SENG3011

Exception(al) Coders
Final Project Report

Section 1: FluTrackers API

Requirements / Use Cases

Requirements:

The requirements of our API were specified in detail in the project specification document provided. We translate them into the following list.

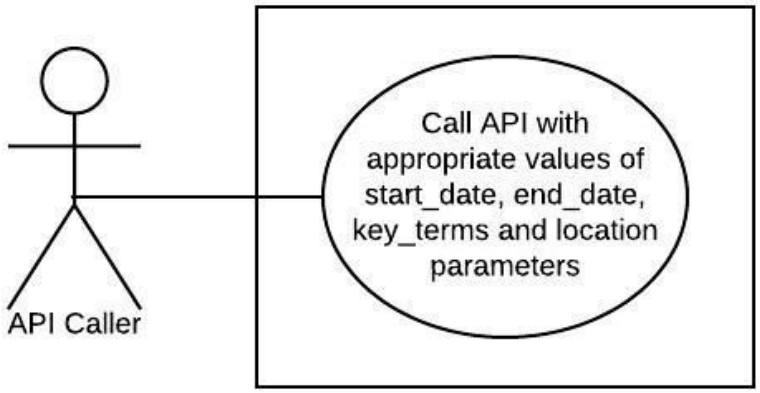
1. Given that a user makes a HTTP GET request to the API with valid values of start date, end date, location, key terms (optional) and a limit for the number of articles they want (optional) , the API should produce an HTTP response, where the response body is a JSON formatted list of article objects posted on FluTrackers occurring between the given start and end dates, at the given location and containing all of the key terms.
    a. Given that a user supplies a start and end date, a location name and a set of key terms, all reports within the range of the start and end date and containing BOTH the location and key terms must be returned.
    b. Given that a user supplies a start and end date, location name and no key terms, all reports within the range of the start and end date and containing that location name in the report text should be returned.
    c. Given that a user supplies a start and end date, key terms and no location, all reports within the start and end date containing those key terms in the report text must be returned.
    d. Given that a user supplies a start and end date, no key-terms and no location, all reports within the matching start and end date must be returned
    e. Given that a user supplies a location name all reports containing a location that is a subregion of the searched location (e.g. a search for New South

Wales should return reports including the term “Sydney”) should be returned.

2. Given that a user makes a HTTP GET request to the API with invalid values of the parameters, the API should produce a response with the appropriate error message in the HTTP response body and an appropriate error code.
    a. If either date is not provided or not given in the yyyy-mm-dd hh:mm:ss format, then provide an error message stating that an invalid date has been provided.
    b. If the given start date is after the end date, provide an error message stating that the start date must come before the end date.
    c. If the limit parameter is negative, then provide an error message stating that the limit on the number of articles must be positive.
3. Each article JSON object returned should include the following information:
    a. URL: The URL of the article.
    b. Date of publication: The date the article was published.
    c. Headline: The headline of the article.
    d. Main text: The main body of the article text.
    e. Reports: A list of JSON report objects for each disease mentioned in the article.
        i. Diseases: An array of possible (possibly unconfirmed) diseases for a particular group showing common symptoms.
        ii. Syndromes: An array of all syndromes mentioned for a particular case.
        iii. Event date: The date on which the case occurred.
        iv. Locations: An array of country and location pairs (as strings) mentioned in the article for the given case. The country is compulsory, and location should contain additional provincial, city or suburb information as a comma separated string.

Use Cases:

Given the standard way in which our API is used, we have the following simple use case diagram.



System Design and Implementation

Parameters and Responses

In accordance with standards for REST, we input information to our module via HTTP GET requests. Parameters are 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 - Zero or more comma separated key search terms. (can be empty)
4. string::location - contains a city/country/continent (can be empty)

The API also accepts the additional optional parameter:

1. string::timezone - string representing a valid timezone
2. Int::limit - represents the number of articles to be returned

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&limit=4

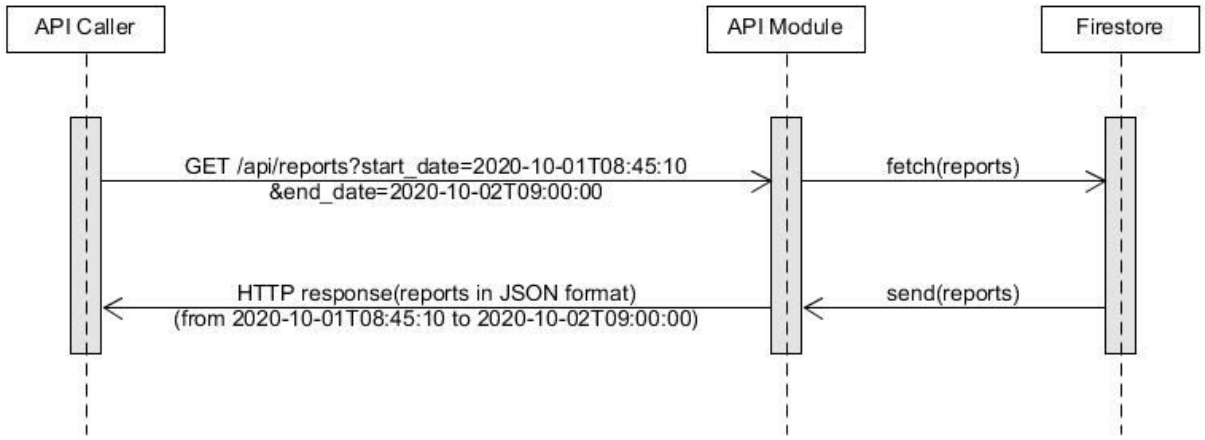
Responses are collected in JSON format sent back in a HTTP response body, for example:


```
{
        "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"
                        ]
                }
        ]
}
```

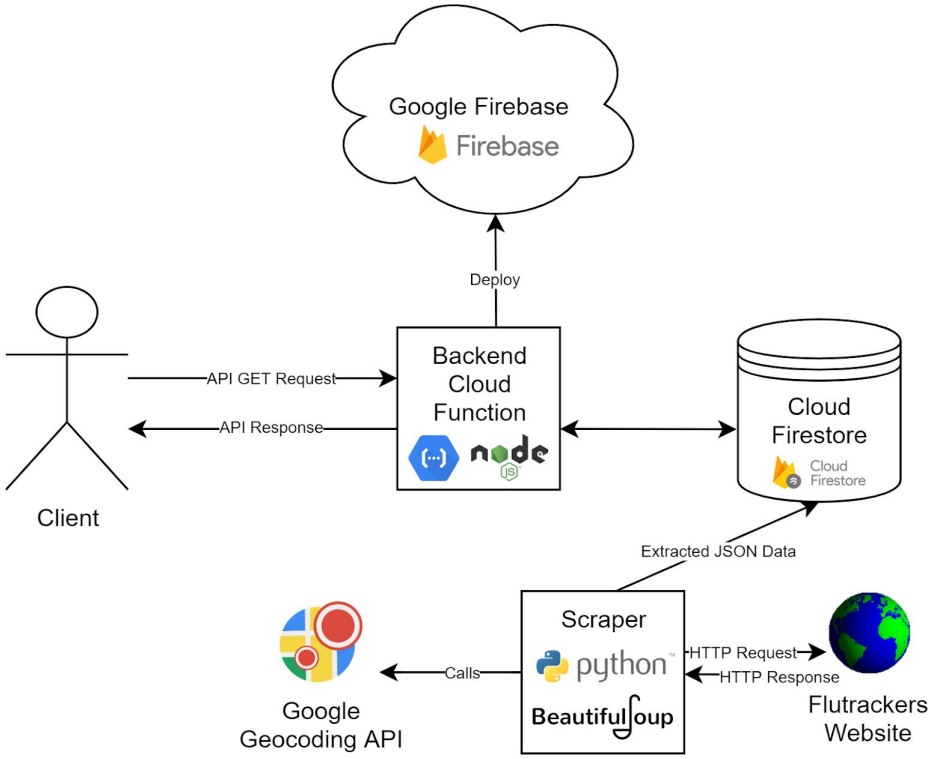
Sequence Diagram

A typical usage of the API is shown below.



Architecture

Below is the diagram showing the key components of the API and their interactions.



Design Choices

Comparison of backends:

	Google Firebase (serverless)	Amazon Lambda (serverless)	Server-based backends (Eg: Python+Flask/ Django, LAMP)
Ease of Use	Very simple set up. The Firebase Admin SDK supports many languages including Python, Node, Java, Go, etc. to write scripts. 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. Admin SDK is available in Node.js, Java, Python and Go	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 *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, the server is hosted locally making stress-testing difficult.
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.	Requires Lambda proxy integration in order to create a REST API	MySQL has a plugin to be used as a REST API, but ordinarily has a different format to our needs.
Documentation	Extensive documentation and video tutorials available.	Good documentation.	Extensive documentation.
Community Support	Good community support.	Good community support.	Good community support.
Pricing	Pricing 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 based on pre-purchased capacity rather than executions.
Monitoring	App development platform includes crash analytics and performance monitoring.	External monitoring frameworks required e.g. Dynatrace, Datadog, Solarwinds.	External monitoring frameworks required.

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 Lambda, 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.

In terms of pricing, assume the data storage costs are negligible and suppose one million calls will be made to our API. Cloud Firestore is priced at $0.18 / 100K document reads, so it will come to $1.80. Amazon Lambda is priced at $0.20 / 1M requests so it will come to $0.20. For the server-based architectures, pricing is based annually and can go into hundreds or thousands of dollars.

Comparison of 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 the 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	Comparatively slow.	Fast due to its simplicity.	Fast as it uses asynchronous system calls.
Ecosystem	Has lots of dependencies.	Confined to jQuery-specific libraries.	Good ecosystem (especially for complex projects).

We chose BeautifulSoup as our web scraping library due the following:

- 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 Cloud 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.

Section 2: MedWatch

Requirements / Use Cases

Requirements

Based on the importance and severity of the COVID-19 pandemic, it was clear that our application would address this issue. Our intention was to aggregate data in such a way that it helps users make informed decisions about their daily lives.

As a result, we focused on an app relating to hospitals, as hospital visitations and medical treatment are among the few reasons that people are still able to leave their houses, and in fact, the rates at which people are using hospitals have expectedly increased due to the pandemic. Our central business objective is to graphically show users the level of strain experienced by each hospital in New South Wales, as well as the local infection rates of coronavirus thus enabling them to make informed decisions about where to go if they need to visit a hospital.

We present a cascading list of requirements, where sub-requirements are dependent on their parents being satisfied, with priorities listed according to the following scale.

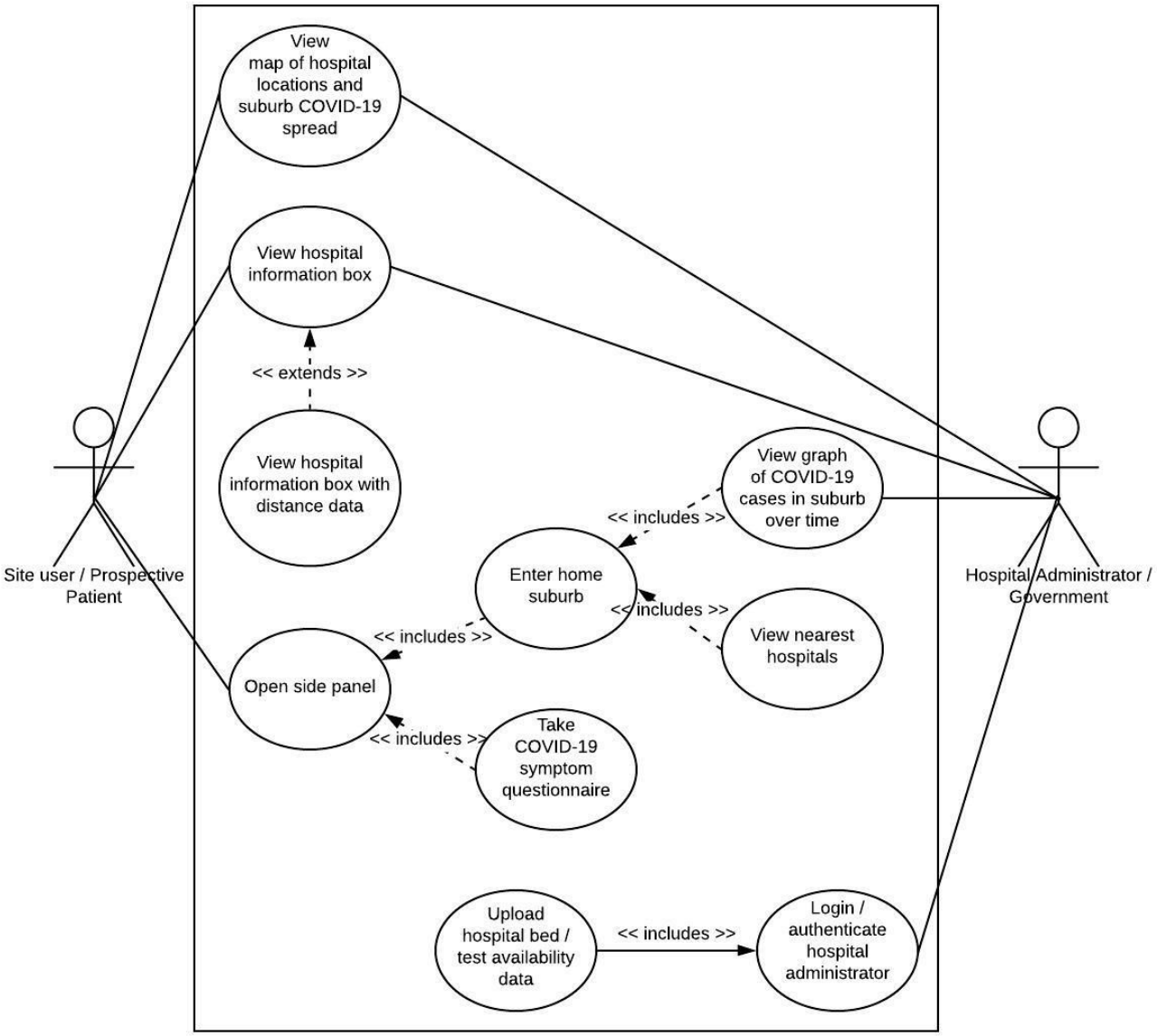
- Priority 1: These features are core features of the application, without which its business objectives will not have been achieved.
- Priority 2: These features should be implemented if there is time, and contribute to the main business objectives.
- Priority 3: These features are extensions and would add value to the application if implemented.
- Priority 4: These features add some value but can be left out.

Requirements list:

1. Display a map of New South Wales with marked hospital locations. (1)
    a. Display graphical representation (e.g. colour coding) of availability of each hospital in terms of available beds. (1)
    b. Display graphically the number of COVID-19 cases in each suburb. (1)
2. On-click of a hospital name or marker, show an information panel containing in-depth hospital view. (1)
    a. Give the best possible estimates of the following statistics: the number of beds available, the number of beds in total, the number of COVID-19 test kits available. (1)
    b. Display a graph of the number of available beds in the hospital over time, as well as a prediction into the future based on local COVID-19 transmission rates. (2)
3. Search bar to search directly for hospitals or locations and bring up overview panel for that hospital. (2)
    a. Drop down menu to suggest hospital names as user types. (4)
4. Implement a form for hospital administrators to fill and provide accurate and current hospital data such as the number of beds and test kits available. (3)
    a. Update website information as forms are filled in by hospitals. (3)
    b. Add authentication so that only verified hospital administrative staff can fill in form. (3)
5. Allow users to enter their home suburb to see suburb-specific information. (3)
    a. Display a graph of COVID-19 cases over time in the user's home suburb. (3)
    b. Display a list of closest hospitals by distance. (4)
6. Implement a coronavirus questionnaire for users to fill out according to their symptoms, and provide advice on what to do based on the result. (3)
7. Display landing page to explain objectives of website and provide a safety disclaimer to users. (4)
8. Implement caching of responses to improve response times (4)

Use Cases

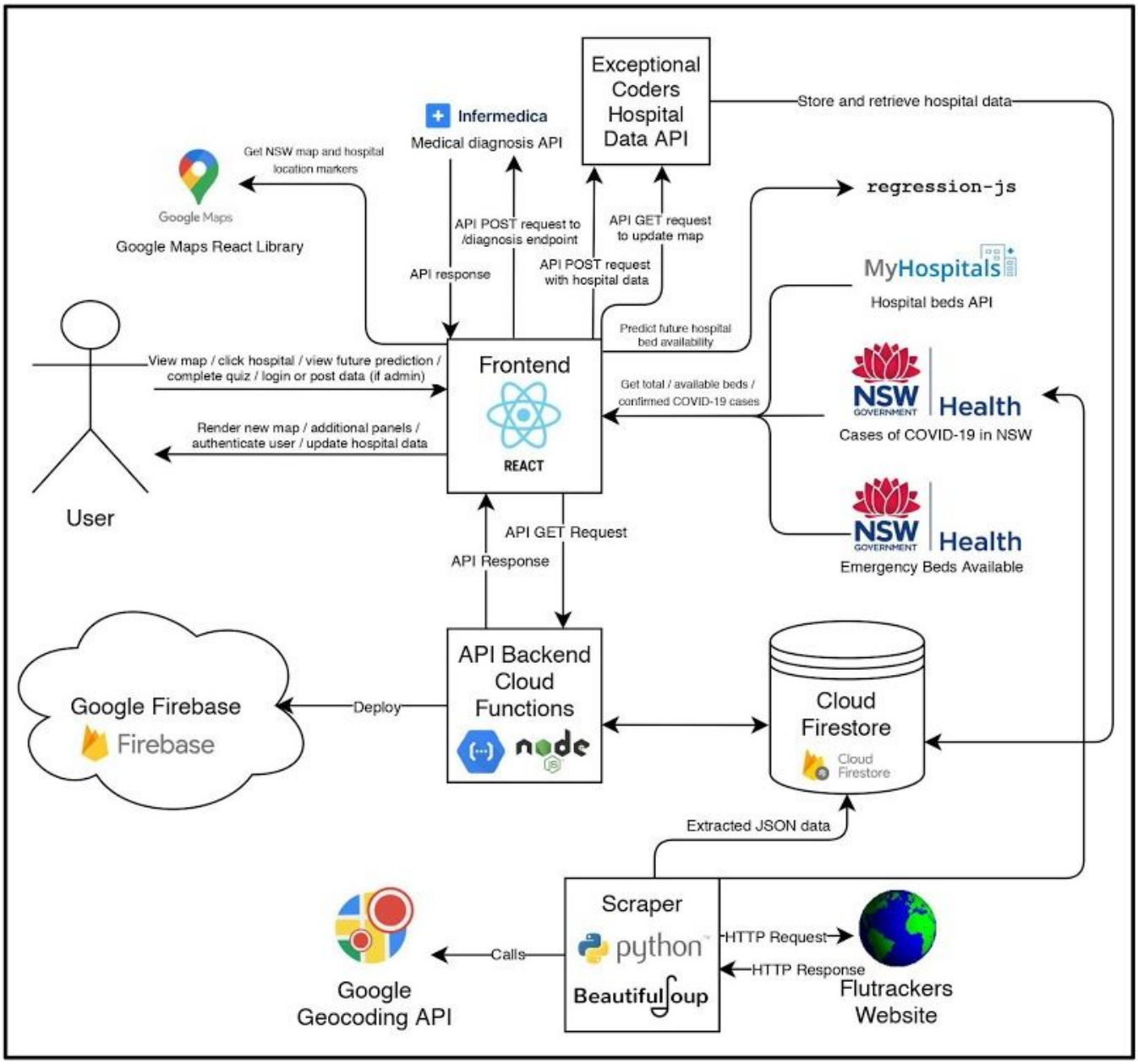
We present the following use case diagram representing how users will interact with our application.



System Design and Implementation

Architecture

Below is an architectural diagram showing the complete technical stack we use and the interaction of components.



Design Choices

Comparison of frontends:

	React	Angular	Vue
Performance	Benchmark project study shows all to have similarly good performance.	Benchmark project study shows all to have similarly good performance.	Benchmark project study shows all to have similarly good performance.
Deployment	Create-react-app creates a fully configured React app.	Most difficult to deploy, but can use the Angular CLI.	Easy to deploy with the Vue CLI.
Single-page application	More complex to develop single-page applications - need external libraries to do some things like form validation.	Angular is specifically built to develop single-page applications. Uses routing and form validation.	More complex to develop single-page applications - need external libraries to do some things like form validation.
Difficulty to learn	Everything is JavaScript, but we would need to learn JSX syntax. Stable framework.	Uses TypeScript so components, modules and syntax look different to JavaScript and could be difficult to learn. Updates their framework every six months.	Closest to HTML and JavaScript basics, so might be easiest to learn. Downside is that it allows poor code, which could make debugging difficult.
Experience	Team has experience in React.	Team does not have experience in Angular.	Team does not have experience in Vue.
Industry Usage (as an indication of popularity)	High	High	Relatively low.
Community Support	Large community - most common problems are answered on StackOverflow.	Large community.	Slightly smaller community.
Documentation	Thorough documentation and video tutorials.	Good documentation.	Documentation is heavy on examples.

All in all, there was no clear winner of the three most common frontend frameworks, so we chose React primarily due to our team's prior experience with it.

Choice of map library:

Google Maps was the clear choice in order to render the map that is the basis of our application, and overlay it with markers and other graphics. Google Maps has a library for React, which allows easy integration for our app.

Choice of modelling tool:

Regarding the way in which we modelled the spread of coronavirus, we had to choose between a regression tool or a more powerful epidemic modelling tool. The regression option had the advantage that it could be run entirely in the frontend, whereas for the epidemic-specific tool, it would have been too bulky to run in the frontend, requiring us to add a backend server.

Ordinarily, the epidemic predicting model would obviously provide much more reasonable predictions of the spread of coronavirus, however in this case, our suburb-specific data for coronavirus transmissions was such that it only provided the total number of cases that had been found in a suburb since the beginning of the pandemic. So, if for example, Kensington had 8 cases in our data, but all 8 of those cases were in January and had all recovered fully by now, they wouldn't be relevant to the disease spread model.

For the scope of this project, we decided to not assume how long it might take for a person to recover and thus calculate the number of cases from the beginning of the pandemic. Thus, we chose to use the regression tool because we felt that the advantages of using the high-powered model were undercut by a lack of accurate data. We found that the regression tool is also more resilient to poor infection data, as it factors in official hospital statistics as well judges the availability of hospitals, and simply extrapolates this trend over time.

Choice of datasets:

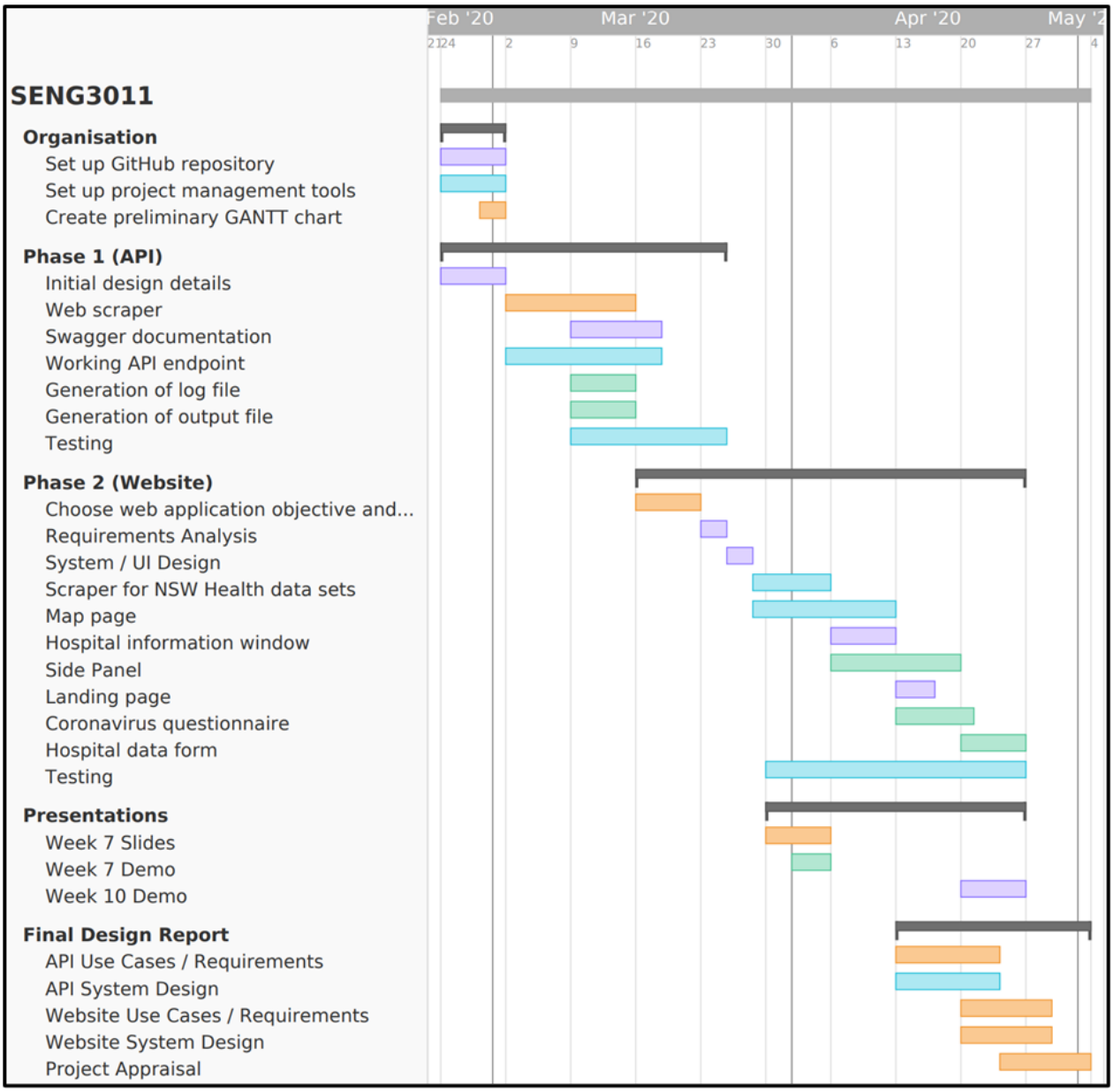
Although aggregate statistics are regularly published, it is difficult to find publicly available data for local cases of COVID-19, especially data that is updated regularly enough to be useful. We found that there was no such data available on a national level, but fortunately, New South Wales in particular was maintaining a table of cases by suburb. The availability of this data from NSW Health was the key reason for our decision to limit the map to just NSW.

Also, surprisingly, we found that there is not much data that exists to describe individual hospitals' resources, beyond a total bed count (which is given in the MyHospitals API). NSW Health maintains live trackers of the number of emergency beds available in each hospital, which we used to inform our predictions of how busy the hospitals were. Eventually, due to the guesswork involved in predicting the number of available beds for each hospital, we updated our design to allow hospital administrators to log in to our application and manually upload the official and current numbers of available beds and available COVID-19 tests.

Section 3: Team Organisation and Appraisal

Division of Responsibilities

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).



Project Appraisal

Achievements

Main map page:

The primary achievement of the project is an interactive map of New South Wales, that marks out hospitals, and for each hospital, shows their approximate availability with a colour grading scheme. This is combined with a graphical view of the spread of the coronavirus in each suburb, by overlaying suburbs with red circles sized proportionately to the number of cases in that area. At a first glance, this visual display makes it very clear to users which suburbs have greater transmission of COVID-19, and gives the user an ordinal impression of the relative risk-levels of going to a hospital based on the availability of the hospital and the risk of the surrounding area of the overall safety of the hospital.

Users can then make their minds up about hospitals to visit based on their own health concerns and private situation: for example, someone who is immunocompromised but lives in an area shown to be "high-risk" and where hospitals are running low on beds might choose to make a long drive to a relatively safer area, however the average person may still choose to go to their local practice.

The website also contributes to hospitals not becoming overwhelmed. Given that users can see which hospitals are under more strain, it will naturally distribute traffic to other hospitals, thus spreading out the total load of patients.

Hospital availability prediction:

Our website implements a prediction feature, where clicking on a hospital shows the number of beds available over time as a graph, and is extrapolated into the future using regression tools. Ideally, this serves as an accurate prediction of how the pandemic will impact hospital resources into the future, and this provides major benefits to hospitals to order supplies and prepare for future demand. It also helps patients with elective surgeries or hospital plans in the next few weeks perhaps reschedule those plans or contact other hospitals based on their own risk levels.

Users who have a hospital in mind for which they wish to look up information are able to use the search bar, which implements a drop-down menu to suggest hospitals based on the user's typing.

Localised side-panel:

Another key component of the application is the side-panel. Users are able to specify a location by suburb, and the panel view will bring up the closest hospitals by distance, giving users an alternative way to find the most conveniently located hospitals with high availability.

The side-panel also includes a questionnaire, which implements a CDC recommended flowchart for users to fill out to determine what their course of action should be based on any symptoms they are showing. This contributes to our site becoming a centralised hub for any people feeling ill to investigate what they should do next and which hospitals they can visit.

Issues

Data availability:

The main issue for our project was the amount of data available for us to use to give a precise picture of hospital availability and COVID-19 transmission. In terms of the first aim of our project: to give users the ability to make informed decisions about hospitals to visit based on availabilities and local COVID-19 transmissions, there were a number of data sources that were lacking that would have allowed us to give a much more wholistic and accurate image of the situation. These included:

- Data for the number of available emergency / ICU beds, hospital resources (e.g. ventilators), hospital staff, personal protective equipment available at each hospital at a given time - each of these data points provide an insight into how overrun a

hospital is, and enable us to give advice to hospitals to order in more resources if we feel they are at risk of becoming overwhelmed.

- Data for the number of coronavirus tests available at each hospital - this would allow us to advise patients who are feeling coronavirus symptoms to visit hospitals with more tests, thereby also reducing the queues for tests at hospital.
- The data and location of each reported coronavirus infection, and which hospital the patient is being taken to (if any) - this information would enable us to perform much more sophisticated modelling for the disease, since at the moment, although a suburb might be shown to have had 10 cases, our predictions for transmission should depend heavily on whether those cases occurred two days ago or two weeks ago. These predictions would be very valuable for us to make further predictions for the levels of hospital strain in the areas that are deemed to be heading towards high rates of infection, which in turn would help hospitals make plans.

Disease Model:

At the moment, our modelling of hospital strain is based purely on a regression model, and although regression with enough data gives a decent prediction for the kind of growth we are seeing in cases, or the strain on hospital beds, it is weaker than epidemic-specific modelling tools that require infections by date in order to make predictions.

FluTrackers API:

The initial requirement for us to use the results from our own API in our project was difficult to fulfill given the nature of our project.

Our website was designed in order to help users form their day-to-day decisions, and as such, it explores statistics on a local level as opposed to a national or international scale. Although our API had the functionality to search by local suburbs, cities, states or countries, for the few reports about Australia that were on FluTrackers, they rarely included such specific geographical information, usually showing cities at most.

For our initial demonstration, we decided to include an additional page that displayed a list of all recent world disease outbreaks, as well as a search bar with some filters. Although it did serve to remove some of the abstraction of the summary statistics for the users, it largely did not integrate well with the main purpose of the project, which was the map page.

Desired Skills

Understanding of stack choices: It would have been helpful to have deeper knowledge of how to evaluate a particular technology stack for our purposes, i.e. to know what features to look for when building different projects. Although our mentor was very helpful in terms of our choice of architecture, it would be useful to have slides that break down commonly used programs and tools in industry in terms of their strengths and weaknesses given contexts.

Future Improvements

More accurate data: Given the recency of the coronavirus pandemic, new tools and data are being released by the day. A recent development has been a University of Sydney dashboard that shows the number of cases in New South Wales by postcode. This data is likely more accurate and more up-to-date than the current NSW Health data we are using (which is updated semi-regularly as opposed to live). Also, tracking cases by postcode instead of suburb allows us to perform better modelling for predicted transmission. Additionally, closer collaboration with hospitals and governments would allow us to use internal data that is more accurate and current.

Improved prediction logic: Currently, we make predictions regarding the spread of coronavirus and its impact on hospital availability using regression analysis on past data. However, in the context of a virus, there are further modelling tools that exist to predict the spread of an epidemic, factoring in the extent to which people are socially distancing. In future, we would look at adding a backend capable of performing such algorithms in order

to make better predictions about future strain on hospitals. This would require data that tracks COVID-19 cases by suburb and also over time.

Integration of more hospital information: Beyond information about available beds and testing kits, more collaboration with hospitals in future could lead to more comprehensive and centralised hospital data for users to be able to track. For example, hospitals or clinics could upload the business of GPs, or availability of vaccines so that prospective patients can quickly determine if a hospital has what they need.

Visual disease spread: Since we are already using modelling tools to predict the impact of COVID-19 on hospital availability, we could use similar modelling to graphically show the predicted spread of coronavirus over the next one or two weeks. This would give a more holistic view of how we are predicting hospital strain, as we would be visually relating it to the epidemic.

Increased geographical scale: Our initial decision to limit the website to New South Wales was based on a lack of hospital and coronavirus data from other states. However, given our adaptation to ask hospitals directly for the relevant information, it would be possible to extend the website to all of Australia or internationally without the need for location specific health APIs.

Beyond coronavirus: After the conclusion of this pandemic, we envisage that our application will still have considerable value as a tool to let people know about their local hospitals and how available they are. Furthermore, if we have access to health data about cases of other diseases (e.g. influenza) by suburb, we would be able to continue using our prediction tools to inform patients of the current hotspots and the hospitals most under strain from the epidemic. Likewise, hospitals could use our application to track the spread of diseases and plan to stock up on medications / share with other hospitals accordingly.