

Design Details

Describe how you intend to develop the API module and provide the ability to run it in Web service mode

Our group will develop a REST API module and a website dedicated to providing reports and articles from WHO Disease Outbreaks. The project and its modules can be broken down into these four stages:

1. Implementation

The entire project will be divided into three sections; the website, the API and the web scraper whose relationships are as indicated in Figure 1.1.

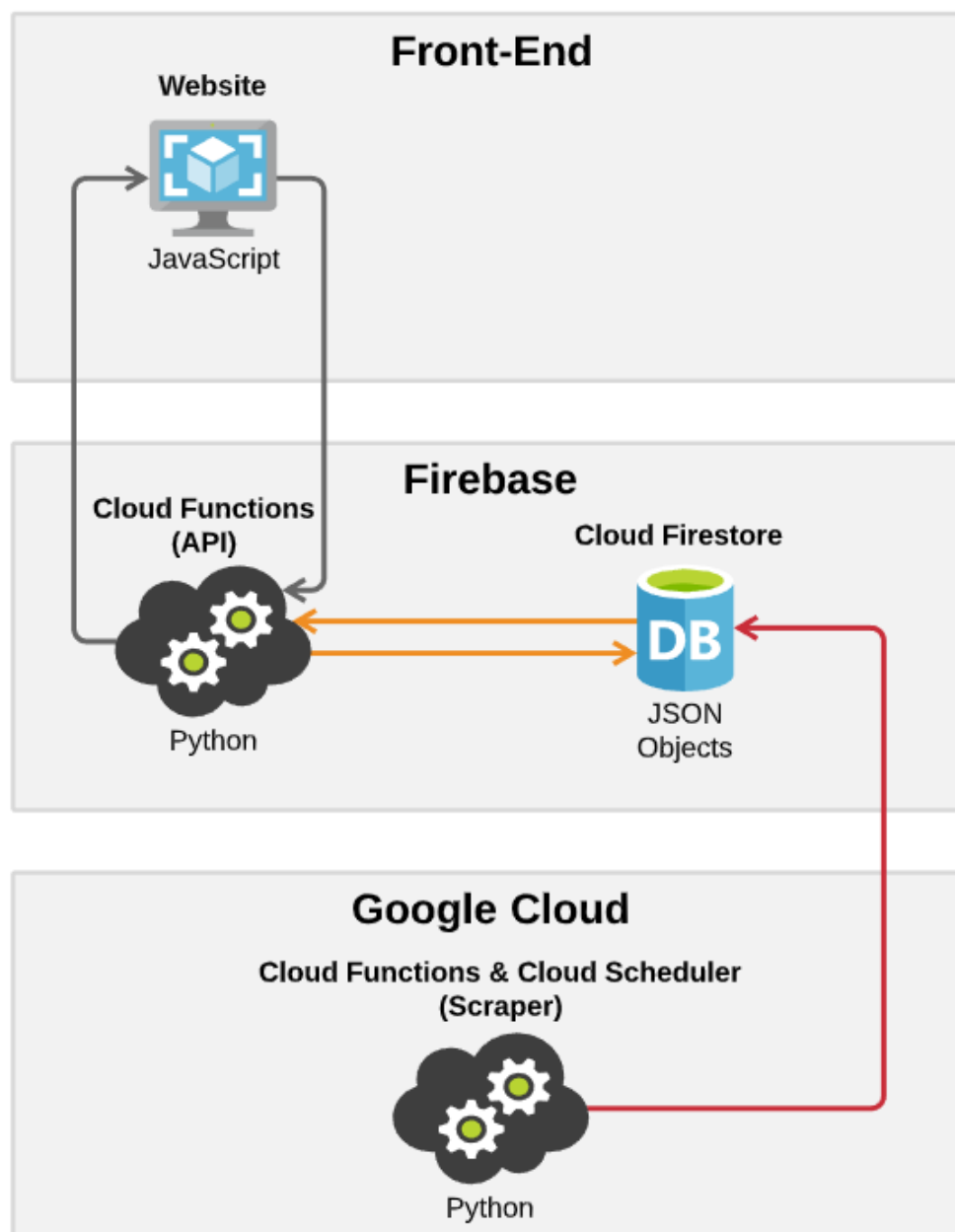


Figure 1.1: Software Architecture Model

Web Scraper

The web scraper will be executed daily as a cron job to ensure the database is kept up to date. This will be achieved by using the Cloud Scheduler to regularly run Cloud Functions from the Google Cloud website. Since the web scraper's capabilities to parse and search for the appropriate reports underpins the project, it will be primarily focused on and continually developed.

API

Similarly the API will be hosted on Firebase through the Cloud Functions console which will utilise Firebase's Python SDK. This will ensure the API can properly access the Firestore database and that its the endpoints are open and ready to receive requests & respond to them accordingly.

Front-end website

Moreover the website will be developed using Firebase's CLI and Firebase's JavaScript SDK to ensure it can be properly hosted on Firebase's website. This website will constantly utilise the database through the API so the user is presented with the latest report or any related reports from WHO.

Each of these components will be developed concurrently and separately within our group, in order to meet each deliverable's deadline.

2. Deployment

Firebase will be used to host the completed API and website we design, as it will automatically come with an SSL connection so users can trust these domains and as developers we will have full control over what sections of code is deployed. Firebase also supports dynamic & static content which is imperative for any website, ensuring reduced latency for a quick service.

API in Web Service Mode

The API will adhere to REST principles as its primary intent is to be usable to anyone in order to gain access to WHO reports & articles for their website. Therefore when it is run in web service mode, the API itself will be hosted on a specific URL, where a client will be able to access its resources only by sending requests through endpoints; a path on a specific URL which the server will then react with a response. This type of communication between the applications will allow information to be transmitted from machine to machine. The process of how a user or developer can communicate with the API is expressed in Figure 1.2.

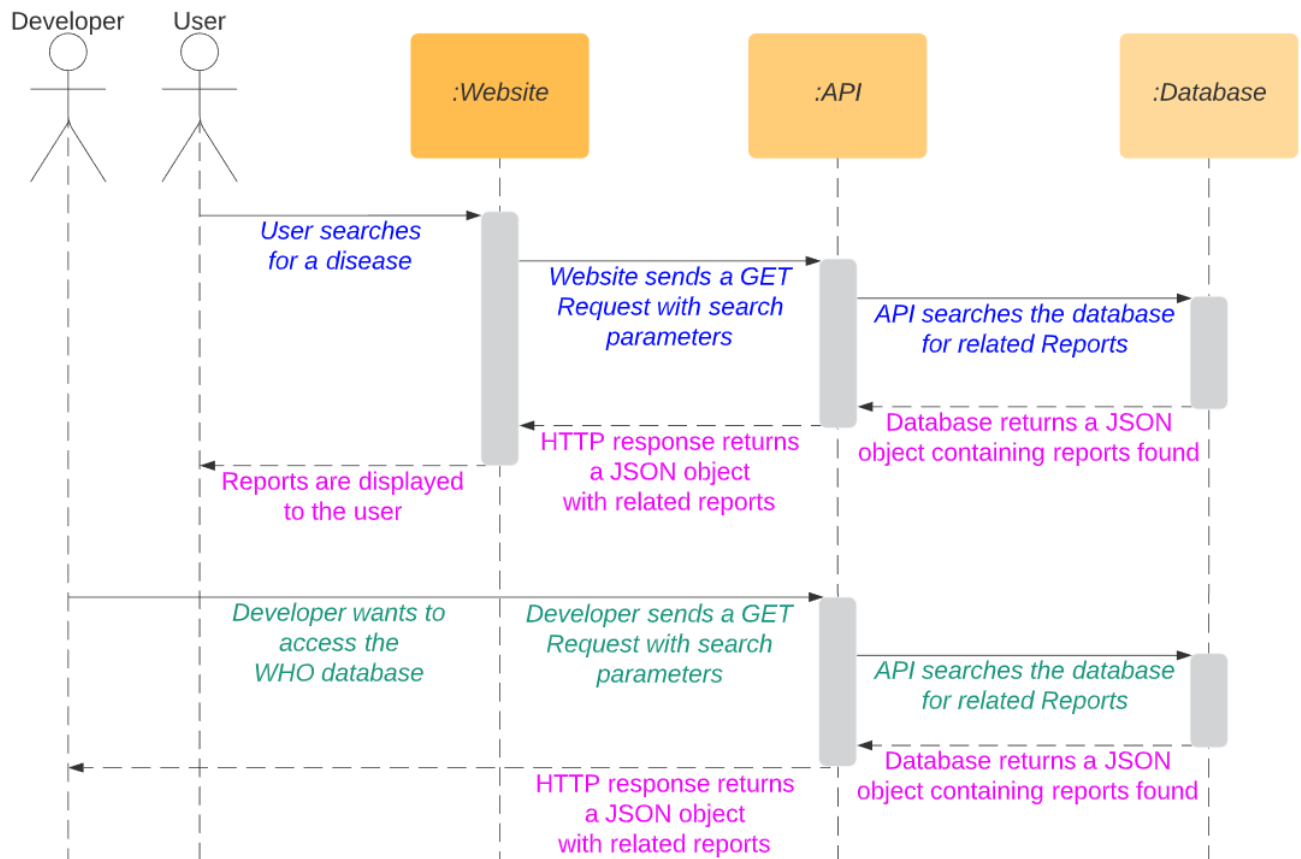


Figure 1.2: Sequence Diagram of a Developer and User

3. Testing

The API, website, scraper and database will thoroughly be tested individually offline and online, which will then undergo the same testing when these components are linked. To test these components offline, mock responses will be sent to the website from the API, the database will have manually inserted information, and the scraper will be tested it can handle any type of web page (SPA & MPA) and it is compatible with all diseases listed in appendix D of "Project Specifications.

4. Documentation

The API will be thoroughly documented during its development, which will be completed using swagger.io; outlining how to use it. Otherwise all changes and versions of each component can be found in our Github repository.

Discuss your current thinking about how parameters can be passed to your module and how results are collected. Show an example of a possible interaction. (e.g.- sample HTTP calls with URL and parameters)

The main purpose of the API is purely to provide the user with related disease reports based on the given search parameters; period of interest, key terms, location and timezone. Any client can interact with the API using a HTTP request. Since the API's purpose is to only ever provide data, it should only need to read the database. Hence our group currently believes the API only needs to be designed with a single GET request. Figure 2.1 Illustrates this process of how a website would interact with the API and Figure 2.2 shows an example of the HTTP responses.

Step	Website Webpage	API Webpage
1. The user enters in search parameters		
2. The user confirms their search	All the search parameters are added to a GET request.	
3. The GET request is sent to the API's endpoint (API/report/)		The API verifies the request is valid, containing all necessary search parameters
4. The API searches for related reports in the Firestore database		The API waits to see if any JSON objects are found
5. A collection of reports are found in the database and a copy is sent back to the API		A 200 HTTP response body is created, with the JSON object containing the reports copied into the payload
6. The response is sent back to the API	The website receives the successful response	The API sends the response to the website.
(Alternative Case 1 at step 4) 4. The query string containing the search parameters are incomplete	The website receives a 400 HTTP status response; the request was malformed and failed.	The API sends a 400 HTTP response to the website.
(Alternative Case 2 at step 5) 5. No reports are found that relate to the search parameters. So The database returns false.	The website receives a 404 HTTP status response; the request was failed.	The API sends a 404 HTTP response to the website.

Figure 2.1: The Process of Communication between the API and a website

Specifically as captured in Figure 2.1, the search parameters will be passed to the API through a query string in the URI of the GET request. This is because the information is not sensitive so this method will ensure reduced latency. Then as it is commonplace to handle a JSON response, all successful API requests will respond to the client with a JSON object, thus the headers will always state the content type as JSON.

HTTP	Example HTTP Request/Response	
GET Request	<pre> parameters = { method: "GET", headers: { "Content-Type: application/json" } } fetch(`\${APIURL}`/report/start_date=2020-17-01&timezone1&end_date=2020-xx-xx&timezone2&key=Coronavirus&location=China, parameters) { </pre>	
200 Response	<pre> { headers: { "Content-Type: application/json" "Status-Code: 200" }, body: { "url": "https://www.who.int/csr/don/17-january-2020-novel-coronavirus-japan-exchina/en/", "date_of_publication": "2020-01-17 xx:xx:xx", "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 (2019-nCoV) from Wuhan, Hubei Province, China. The case-patient is male, between the age of 30-39 years, living in Japan. The case-patient travelled to Wuhan, China in late December and developed fever on 3 January 2020 while staying in Wuhan. He did not visit the Huanan Seafood Wholesale Market or any other live animal markets in Wuhan. He has indicated that he was in close contact with a person with pneumonia. On 6 January, he traveled back to Japan and tested negative for influenza when he visited a local clinic on the same day.", "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"] }] } } </pre>	
400 Response	<pre> { headers: { "Content-Type: application/json" "Status-Code: 400" } } </pre>	
404 Response	<pre> { headers: { "Content-Type: application/json" "Status-Code: 404" } } </pre>	

Figure 2.2: Example HTTP Request & Responses

Present and justify implementation language, development and deployment environment (e.g. Linux, Windows) and specific libraries that you plan to use

The entire tech stack will be critically analysed in order to evaluate the most suitable languages & libraries for each application.

Tech Stack Choice 1: Server vs Serverless

All applications of the project will inevitably be hosted online, prompting research into the benefits of a server or serverless architecture. Our examination into both architectures is elaborated in Figure 3.1.

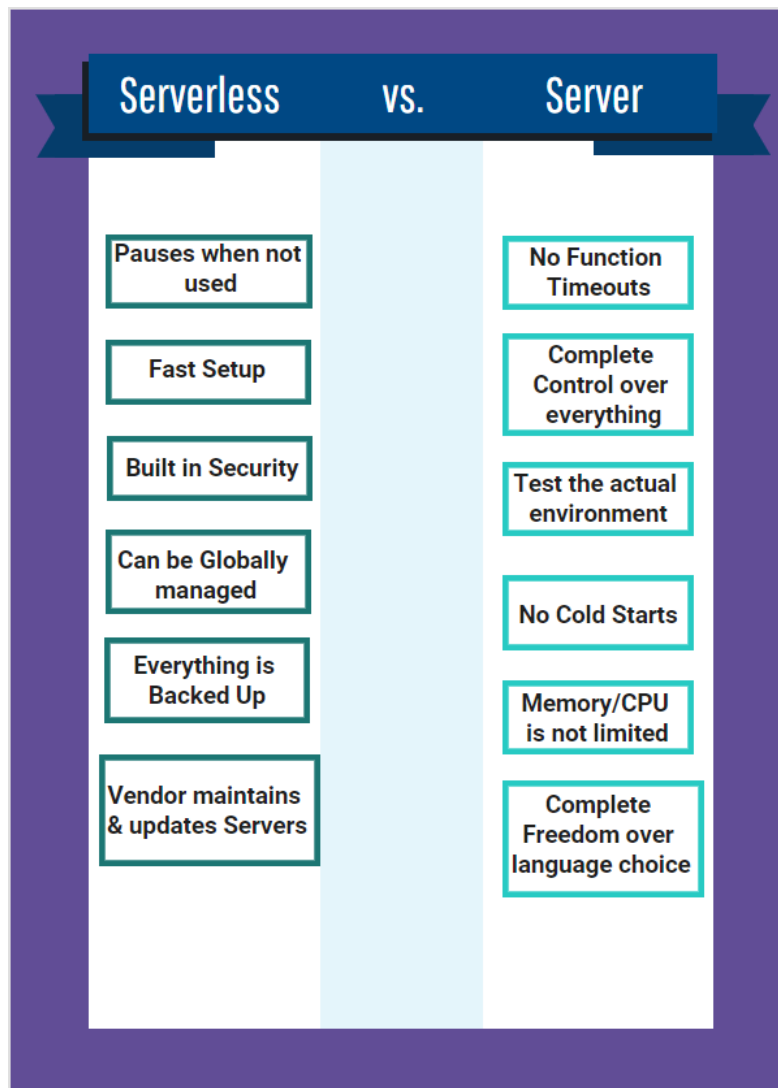


Figure 3.1: Server Comparison

The main consequences of a serverless architecture would not severely influence the application, while the advantages would significantly make development more efficient and less time consuming. Hence for these reasons we will use a serverless architecture.

Note since cold starts present the biggest concern for a serverless architecture, we will be using Python for the API as it has a faster run time than NodeJS. Similarly we will ensure we minimise package usage as it will shorten a cold start time by reducing internal networking latency. The other benefits that Python presents to this project is further explored later.

Tech Stack Choice 2: Serverless Platforms

Since we will be developing using a serverless architecture, three of the main serverless platforms which support online functions have been compared in Figure 3.2.

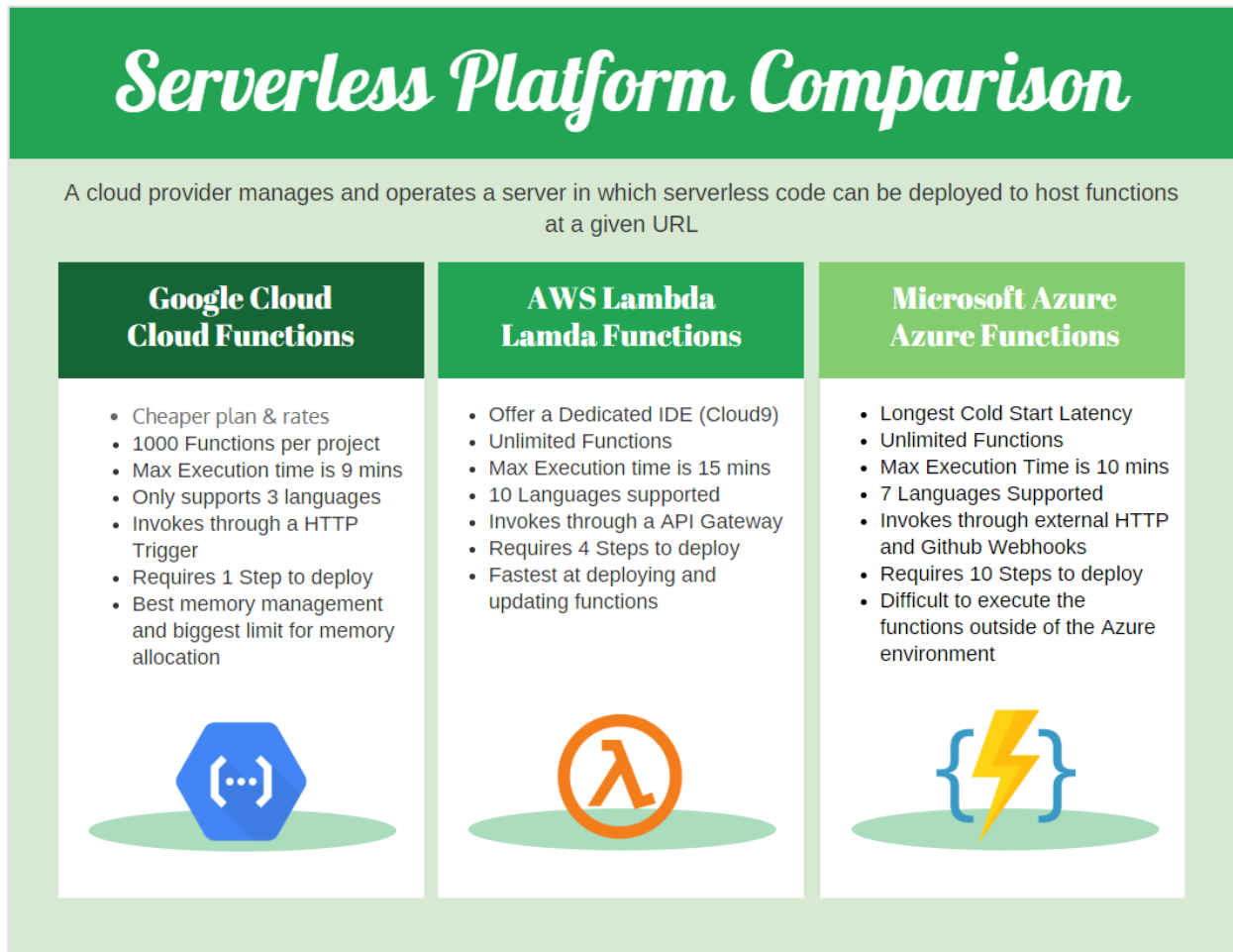


Figure 3.2: Serverless Platforms Comparison

Overall each of the server platforms are similar and offer the same basic capabilities to support a serverless architecture. However Google Cloud in particular provides the most simplistic UI and behaves the most like a typical environment (as it can be replicated with a Virtual environment). Therefore since our group has no experience with any serverless platforms, we will use Google Cloud, and our team is also familiar with the only three languages it supports.

Tech Stack Choice 3: Progressive Web Applications

In addition to the serverless platform a progressive web application will be utilised to hasten the development process. The main applications we assessed are judged in Figure 3.3.


 Progressive Web App Comparison			
Features Available	Firebase	Hood.ie	Deployd
Easy to use UI	✓		
Provides a unique database	✓	✓	
Supports all languages	✓		
Provides a SDK	✓	✓	✓
"No Backend" apps can be developed	✓	✓	
Offers Cloud Functions	✓		
Can host a website & API	✓	✓	✓
Provides "Offline Development" feature		✓	

Figure 3.3: Progressive Web App Comparison

Compared to the alternatives, Firebase will provide the most flexibility to our project without forcing us to use a specific programming language. Through its connection with Google Cloud, it further simplifies the process of developing every application within the limited time frame. Thus Firebase is the most suitable serverless app as every component can be easily hosted and managed all through Firebase.

Tech Stack Choice 4.0: Programming Languages

Firebase supports Python, JavaScript, Go, C# and C++. So each of these languages will be compared as in Figure 3.4 to determine which is most appropriate for our project.

Language	Speed	Extensive Firebase Docs/Community Support	Group Familiarity	Learning Curve	Cloud Function compatible	Asynchronous	Front-End or Back- End?
Python	Moderate	Moderate	6/6	Easy	Yes	Simulated	Both
JavaScript	Fast	Thorough	6/6	Easy	Yes	Yes	Both
Go	Fast	Moderate	0/6	Easy	Yes	Yes	Both
C#	Fast	Minimal	1/6	Difficult	No	Yes	Back-End
C++	Very Fast	Minimal	2/6	Difficult	No	Yes	Back-End

Figure 3.4: Languages Comparison

After the analysis, our group has decided to use JavaScript & Python as we have the most experience and comfortability with these languages which can also be executed as Cloud Functions. Consequently they will ensure we can create a high quality and performing application within the constrictive time frame.

Tech Stack Choice 4.1: Front-End

In evaluating libraries to design the website, we will mainly critique each library based on their flexibility and capabilities as denoted in Figure 3.5.

Library/ Framework	Is it Easy?	Group Familiarity	High Performing?	HTTP Communication	How it Creates Web Pages	Front-End or Back- End?
React	✓	4/6	✓	✗	Virtual DOM	Front-End
Vue	✓	1/6	✓	✗	Virtual DOM	Front-End
Backbone	✓	0/6	✗	✓	Virtual DOM	Front-End
Angular	✗	1/6	✓	✓	Direct DOM	Front-End
Flask	✓	6/6	✗	✓	Components	Both
Django	✓	1/6	✗	✓	Components	Both

Figure 3.5: Front-End Web Development Library Comparison

From our research, JavaScript is inherently designed for Front-End web development, as it works seamlessly with all browsers and capable of making asynchronous calls to the API reducing latency. Therefore a JavaScript library will be chosen, as it offers a wider range of high quality features that can be developed with those libraries compared to those offered in Python. In particular React will be used because of our team's familiarity with it, it is intended for making UIs and it can be easily tested.

Tech Stack Choice 4.2: Web Scraper

Since the web scraper is the most complex component of the project, we will need to use an efficient, powerful and flexible scraping library. Our research of these libraries is displayed in Figure 3.6.

Library	Language	Is it Fast?	Can it Scrape JS?	Is it Easy?	Group Familiarity	No. of Dependencies
Cheerio	JavaScript	✓	✗	✓	1/6	6
Request	JavaScript	✓	✗	✓	1/6	20
Apify SDK	JavaScript	✗	✓	✗	0/6	22
Beautiful Soup	Python	✗	✗	✓	3/6	(Relative) 3
Scrapy	Python	✓	✓	✗	0/6	26
Selenium	JS & Python	✗	✓	✗	0/6	4

Figure 3.6: Scraping Libraries Comparison

After inspecting each web scraping library, BeautifulSoup was found to be limited despite our comfortability with it, so our group will choose the most efficient and effective library. Therefore we will use Python's Scrapy as it is one of the fastest performing libraries and provides us with greater control in managing URLs which will assist in ensuring its Cloud Function counterpart does not time out. We will also use Firebase's Python SDK so that all the scraped reports/articles can be inserted into the Firestore database.

Tech Stack Choice 4.3: Database

The last component of our architecture will be the database, which must be capable of handling JSON formatted data and properly store the data so it can be searched for quickly. Figure 3.7 demonstrates our critique of databases which best suited this project.

Database System	Popularity	Group Familiarity	Learning Curve	Database Model	Storage Limit	Max Document Size
Cloud Firestore	54 th	0/6	Moderate	Document	Unlimited	1MB
Cloud Realtime Database	38 th	0/6	Moderate	Document	Unlimited	1MB
PostgreSQL	4 th	5/6	Difficult	Relational & Document	4TB	1GB
MySQL	2 nd	5/6	Moderate	Relational & Document	2TB	1GB
Oracle	1 st	0/6	Difficult	Relational, Document, Graph, RDF	5GB	1GB
MongoDB	5 th	2/6	Moderate	Document, Search Engine	32TB	16MB
Elasticsearch	7 th	0/6	Difficult	Document, Search Engine	50GB	1GB
Redis	8 th	1/6	Moderate	Graph, Search Engine, Time Series	As big as Machine RAM	512MB

Figure 3.7: Database Library Comparisons

Considering Firebase is currently being predominantly used, the Cloud Firestore database will be chosen as it integrates smoothly. Despite it's integration with other components being hosted by Firebase, the Cloud Firestore database will allow for data to be immediately accessed and eliminates the concern of managing a database on an external platform/server.

Specifically the Cloud Firestore will be used over the Real Time Database as it is more geared towards big data which allows the opportunity for scalability and for an extensive history of reports. Furthermore Cloud Firestore's collection style storage system is suitable as all reports saved in the database will never be updated, allowing us to capitalise on the storage structure to conduct more efficient searches. Hence Cloud Firestore is the most suitable database for our project.

Development Environment

A Virtual Environment will be used when developing all Python based functions, as it best replicates the deployment environment used when the function will be run on Google Cloud and overcome any OS/architecture issues. Hence the all Python packages can be frozen onto a requirements.txt and be copied over to the Cloud Function. The same will be done when developing the website by freezing the dependencies into a packages.json file, to ensure a consistent development environment. Additionally The website will be also coded using the IDE Visual Studio Code since it has in build developer tools allowing changes to occur seamlessly.

Everything will be initially developed on a Linux system, however since every function will be exported to the Google Cloud and the source files for the website will be uploaded to Firebase, the actual architecture is irrelevant as it should work with any platform.

Final Tech Stack

Overall the entire tech stack will operate off of Firebase's serverless architecture, and use a mixture of Python and JavaScript to develop its applications. The full stack for each module are as represented in Figure 3.8.

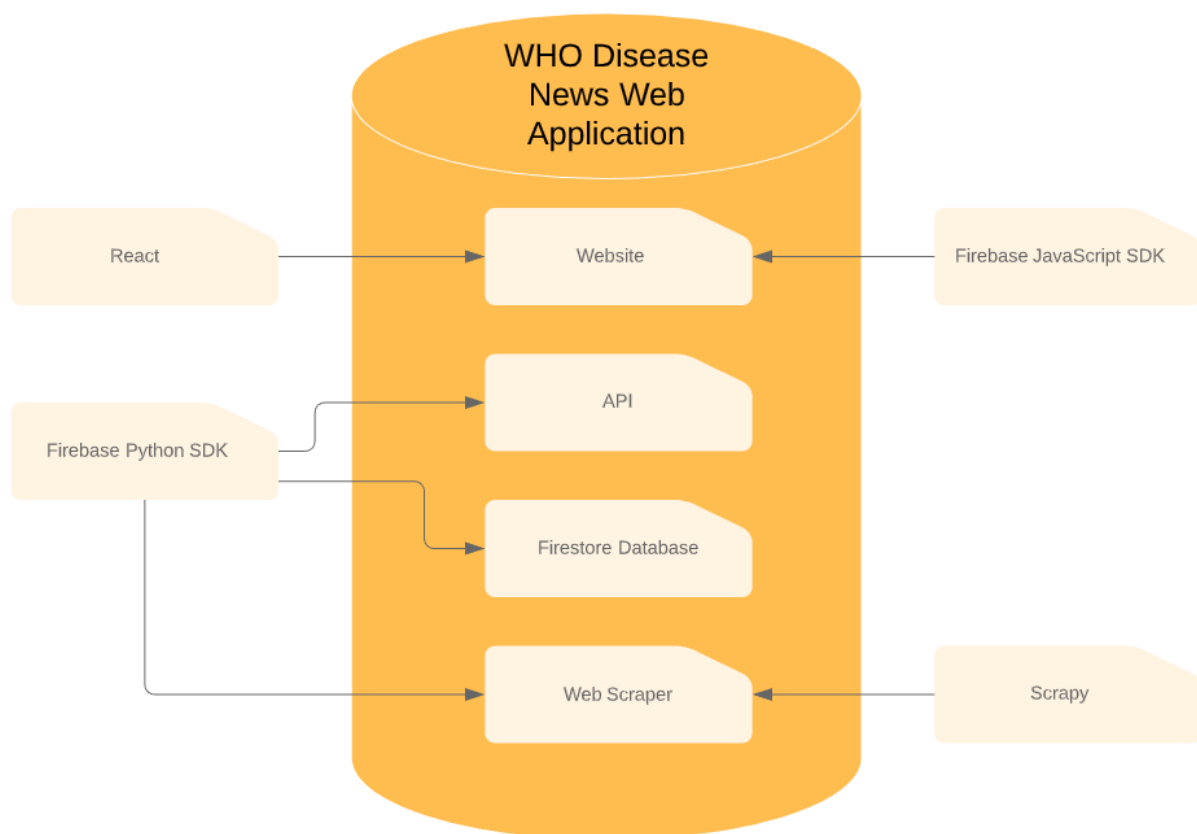


Figure 3.8: Final Tech Stack

Describe final architecture of your API, justify the choice of implementation, challenges addressed and shortcomings

API's Final Architecture

Throughout the development of the API, it became clear that different libraries needed to be utilised than initially thought which has led to differences in final architecture compared to the initially thought architecture (Figure 3.8).

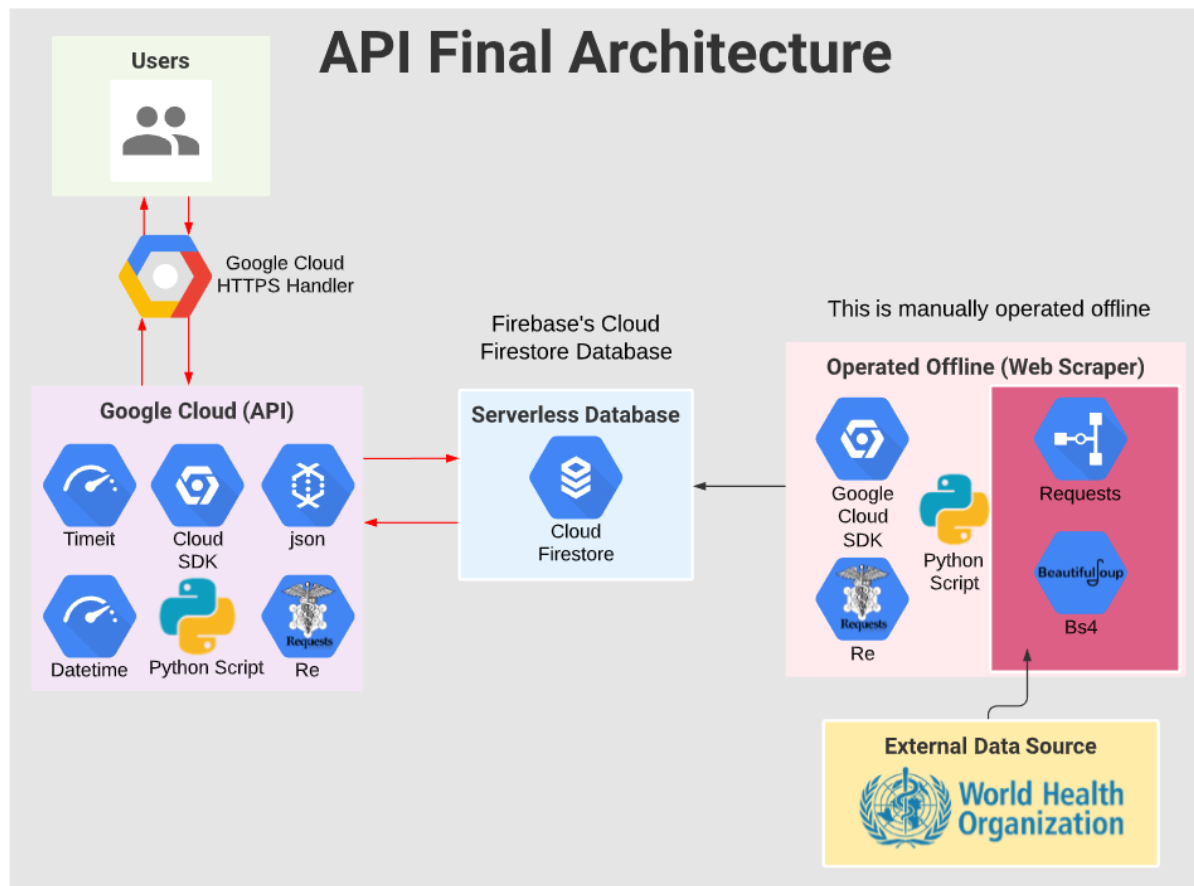


Figure 4.1: API's Final Architecture

Final Implementation Changes

Primarily the API does adhere to the initially planned architecture, because Google Cloud provided thorough performance statistics allowing optimisation and is a platform to securely host the API.

Although as specific functionality was needed, new libraries were added to the architecture: `timeit`, `json`, `re`, `datetime`, `requests` and `beautiful soup`. These new libraries were essential to the web scraping process and accurate documentation for the API logs. The other main difference is the use Google Cloud's Python SDK instead of Firebase's Python SDK. This is because since everything is accomplished on a single Cloud Function, the specific Firebase tools are unnecessary, while Google Cloud offers their own SDK for accessibility to Firestore could be accessible. The reasoning regarding the use of Beautiful Soup is explored in the challenges we overcame during development section. The web scraper will also be operated locally offline instead of using a cron job on a Cloud Function. This is because all previously posted reports are not updated and WHO only publishing a new report every few days, it is unnecessary to constantly pay to host a constantly running web scraper. Therefore there is now 1 less component of our API that is vulnerable.

API Cost Analysis (Above & Beyond)

The overall cost of operating our API on a serverless platform has been evaluated in Figure 4.2 as the API will read and write 1 log to the Firebase's Firestore database (which is negligible) and the web scraper will only write to the database. Since there are 2712 total WHO reports dating back to 1996, the 20K per day free write limit will never be exceeded (if it was used daily), so using the web scraper is free.

Comparatively the API itself is a costly process as it is purely dependent upon the amount of requests made per day. Figure 4.2 indicates how for approximately 1 request takes up 250 reads, which means the API can handle up to 200 requests per day for free as $50,000 / 250 = 200$ requests. While at an extreme case of 3 million reads, 12,000 requests can be made per day. Examining both cases highlights that the cost of writing a new log for each request even in an extreme case, still falls under the free 20K threshold, so we will never have to worry about the cost of our API writing. Similarly with a request count of 250/day, or 12000/day, the total monthly invocation would reach 7750/month or 372,000/month which is far below the free Cloud Function invocation limit of 2 million, so we do not have to account of these fees.

The reason that the read count is so high per request is, because the Firestore database is queried for every location stored in the database. This process is essential despite its expensiveness as it ensures the accuracy of our API and underpins the geolocation. Subsequently the API's daily cost could range from \$3.51 to \$5.31, where the additional fees are from using the server in Tokyo and the CPU usage of operating the Cloud Function. These

costs are particularly expensive as we use the a 2.4Ghz CPU to optimise the performance. A slower CPU option could be chosen to naturally lower cost at the consequence of performance.

Thus at the extreme costs of \$5.31 per day or \$165 per month, the API can be somewhat expensive, although since the maximum amount of reads & writes per day can be limited to a budget and the Cloud Functions do not remain functional when not used, the costs of the API can easily be managed to a low price point. Also since a new user is provided with a free \$365 budget which lasts a year, the true cost our group bares is nothing.

API Average Read Cost			
Per Request	Daily	Extreme Per Day	Total
250/50K	23K/50K	3M/50K	
\$0	\$0	\$1.80	\$1.80

API Average Write Cost			
Per Request	Daily	Extreme Per Day	Total
1/20K	92/20K	12K/20K	
\$0	\$0	\$0	Free


Overall Average API Cost	
Normal Daily Usage	\$0 + \$3.51
Extreme Daily usage	\$1.80 + \$3.51
Cost Per Day	\$3.51 to \$5.31 

Figure 4.2: Overall API Cost

Challenges Overcame During Development

The main issues when developing our API can be categorised under: issues with Cloud Functions, Firebase's Cloud Firestore and Web Scraping.

Cloud Functions (Difficulty: Hard)

As previously stated, Google Cloud's Cloud Functions only support JavaScript, Python and Go, with JavaScript having the most documentation and it is the most popular language used by Cloud Function developers. The consequences of choosing Python became evident when coding the API, as issues occurred between the locally hosted version and the actual deployed version: certain functions were unusable (flask's `make_response`) and code had to be moved around in order for the function to operate as intended.

Therefore the main solutions were to choose another serverless platform (such as AWS Lambda), use JavaScript or just debug the problems. AWS Lambda was not chosen as its documentation and examples are not as intuitive as Google Cloud, thus more time would be needed to understand how to develop use Lambda Functions. Additionally JavaScript was not used because despite there being more documentation for it, Cloud Functions only support Node.js 8 (released 2017) and a beta version of Node.js 10 (released in 2019), while Python 3.7.1 (released in 2018) provided the most stable version. Our group decided not to use Node.js 8 because it is no longer being supported, meaning using the newer packages may be difficult and using older packages would not provide the benefits of the latest version. Similarly Node.js 10 was not used because our team wanted to avoid any issues related to it being still in its beta compatibility status with Cloud Functions. Thus our group continued to use Python as it was released later than Node.js 8 making it a more optimised and stable language, it still supports the latest versions of libraries and this version by default encodes text in UTF-8 instead of ASCII.

So our group just decided to debug the issues through multiple deployments of the Cloud Function which was not an issue as there were only minor things two address and all issues related to Cloud Functions were able to be resolved.

Cloud Functions – Cold Starts (Difficulty: Hard/Impossible) (Above & Beyond)

The major drawback of using a serverless platform is the greater possibility of experiencing a cold start. Since completely preventing a cold start from occurring is impossible, the only solutions possible are to minimise its chance of occurring or to minimise its impact on latency. Since a Cloud Function's environment could stay active/usable for after 5 hours of its last invocation, we could simply call the function every 5 hours. However this solution is not feasible as it still does not completely address the issue, it requires the cost of regularly keeping the API hot and it is impossible to assume when the API may be used. Therefore our group has minimised the cold start latency timing to 1000 ms so that when it does occur, the

requestor does not incur the cost of the cold start (these figures can be further explored in the API Testing document). Specifically to minimise the duration of a cold start, our group minimised the amount of crashes that occurred, incorporated lazy initialisation and used the latest libraries. Cold starts will occur more frequently if the server is to crash, as a cold start occurs after the server is just hosted/available to use. Hence our group thoroughly tested our code and address all possible edge cases (which can be further explored in the API Testing documentation). Similarly through lazy initialisation, less code is kept in the global scope, which means when a cold start does occur, everything in the global scope does not need to be reinitialised until the function that has specific dependencies is actually called. The minimisation of dependencies was also enforced as the less libraries/packages to import the lower the latency. Additionally every single Cloud Function on Google's servers share a dependency cache where all libraries & packages are temporarily stored. This means if a previously invoked function's environment is recycled, using the latest and most popular libraries has a greater chance of these libraries being stored in the dependency cache which can then minimise the import time of a library. This reason further substantiates the why our group choose Python over JavaScript for the development of our Cloud Functions.

Consequently despite being unable to directly stop cold starts, our group was able to mitigate its affects on a user's experience through code refactoring and smart dependency usage.

Cloud Functions - CORS (Difficulty: Easy)

When concurrently developing our front end website, we discovered the API did not support Cross-Origin Resource Sharing, so receiving a response from the API was impossible. The solutions available to us were to have our front end host on the same domain, us an Extensible Service Proxy or just set the headers ourselves. Since this API is intended for public use, only allowing our website to have CORS privileges does not solve the issue. Similarly ESP is predominately used to monitor & control the authentication process of requests coming into the API, which is unnecessary for our simple API. This would also only complicate the process of designing the API which would only increase latency and the time required to develop the API. Specifically we allow any website from any origin to fetch data from our API while the other headers are to handle pre-flight requests. Thus our group chose the simple approach of implementing the headers ourselves which permit CORS as seen in figure 4.3 which easily and quickly solved the problem.

```
Headers = {  
    'Content-Type': 'application/json',  
    'Access-Control-Allow-Origin': '*',  
    'Access-Control-Allow-Headers': 'Content-Type',  
    'Access-Control-Allow-Methods': 'GET'  
}
```

Figure 4.3: CORS Headers

Web Scraping (Difficulty: Easy)

Web scraping itself presents challenges regarding the keyword extraction for summarising reports, although using Scrapy presented a great difficulty.

Since we had to learn Scrapy from start, it presented a challenge in order to get basic functioning web scraper. However only after completing this web scraper did we realise Scrapy could not be hosted on Google Cloud as it has its own unique environment and would need to be hosted on Scrapinghub, a specialised website for Scrapy crawlers. Continuing to use Scrapy would then only cause more problems as multiple platforms had to be managed and time would be required to address the learning curve. Therefore the only logical solution was to use BeautifulSoup, a library most of our group are familiar with and the loss of performance was not a detriment as each WHO reports are generated on a static web page, which only requires a simple crawler.

Furthermore during the web scraping process, our group had difficulties of discerning which how to properly recognise the different names of a diseases & symptoms and understanding that if a report mentions a country or state it affects all cities in that area. Since our web scraper only used string matching, a more complex method could be implemented such as keyword extraction or the usage of other API. However implementing a specialised keyword extraction would require the development of an AI and in general be beyond the scope of what our group could implement in the current time frame. Similarly using APIs would only increase our dependencies on external sources which only increase risk of failure and latency, so there is a greater chance that our API being unfunctional.

Hence our group utilised the Firestore database, to store every single city, disease & symptom (with their alternative names) so that our string matching process would be further optimised. This solution would be faster than reading from a file as the Firestore interactions have been optimised for Google Cloud Functions.

So using BeautifulSoup and the Firestore database resolved all our web scraping problems.

Firebase's Cloud Firestore (Difficulty: Medium)

Firestore presented various issues as its simplistic implementation does not support the JSON datatype and its limited complexity of queries resulted in long wait times to search & filter the database.

The obvious solution of using dedicated library such as MySQL for the database would only eliminate the advantages of a serverless platform which is all inherently connected. While alternatively using the Realtime Database which despite allowing data to be stored in JSON, it is only offered from a single region and supports basic queries, which would only increase latency and doesn't solve the issues.

Consequently our solution was to just rebuild the information extracted from the database into a JSON object, which overall doesn't significantly impact the performance of the API. Moreover when making queries, into the database, we have additional attributes for each report/article so that these attributes could be queried for instead, which significantly reduced the latency. An example of a query which caused this issue was searching for all cities in a state, or all cities in a country as every single city in these areas had to be added to a list. Thus using our solution, a query could be made to all the reports/articles to find if any of them contained information on a country. Note we are aware of Firestore's ability to make custom indexes which should fix this problem, however we were unable to get it working, so we used this method instead.

So using this straightforward approach to the Firestore resolved all of our database hurdles.

Conclusion

Overall all of the challenges we encountered shaped the final architecture of the API which forced optimisation has allowed us to design a polished and efficient final product.

Future Improvements

Major improvements could be integrated into the web scraping process as outlined in the challenges section. However the most impactful change could be implemented in the API's interaction with the Firestore database. With the average request equalling 250 reads, Firestore's free reading threshold can easily be surpassed. Consequently in a future version, the API could pursue and refine alternative solutions to storing constantly read data in the database which would both serve in lowering the overall cost of hosting the API and reducing latency times. Otherwise other serverless platforms which specialise in the transfer of data could be explored.