Testing Documentation

Introduction

Beams used two different testing methods to ensure the API functioned as required. Firstly, white box testing was conducted using Jest and secondly black box testing was conducted with bash/python scripts. The unit tests using Jest are integrated into the CI lifecycle to ensure errors are not merged into the production branch. Along with automated GitHub branch checks, the CI tests will prevent merge requests with failed tests to enter the production code base, without wasting time or overhead from developers.

In this document:

- White box testing
- Black box testing
- Logs
- Testing the Sympt App
- Testing the SIR predictions model

Testing with Jest

Jest is a JavaScript Testing Framework with a focus on simplicity. It is an open source library developed and maintained by Facebook.

The reasons that we chose Jest:

- Minimal configuration and setup
- Good and easy to read documentation
- It's an assertion library

To expand on the last point, Jest uses *matchers* to check assertions. It's library is incredibly extensive and was easy to use. Furthermore, there is community support for Jest matchers and Jest in general making it easier to debug and research errors should they arise.

Tests written with Jest:

- `/server/__tests__/formatters.test.ts`
  This is a file of unit tests designed to check input of parameters. Here we test the `formatQueryUrl()` function. This function takes a query as a string: `?startdate=2020-01-0&enddate=2020-02-01&location=china&keyterms=sars` and returns a `URLFormattedTerms` object, shown below. Designed to make database retrieval easy as now we can utilise each parameter individually.

```
URLFormattedTerms = {
      startDate: string;
      endDate: string;
      location: string;
      keyTerms?: string[];
      count?: number;
      page?: number;
}
```

  Tests done include:

  - Testing start and end date given were valid
  - Testing start date is before end date

    - Testing location was recognised if given, otherwise it should be an empty string
    - Testing keyterms were recognised and separated by commas
    - Testing that stray commas before or after keyterms did not result with empty strings being put as keyterms
    - Testing that a negative count or a count greater than 10 produced an error
    - Testing that a negative page number produced an error

- `/server/__tests__/getArticles.test.ts`
  This is a file for testing that the `getArticles()` function returns the correct articles depending on search parameters given.
  Tests done include:
    - Ensuring that count and page are functioning correctly
    - Ensuring articles are retrieved in descending order from most recent to least

Improving Test Results

During the Jest tests, the team discovered that certain parameters could be changed from required to not required. This made our API more flexible, allowing developers to customise the API to their needs.

Moreover, the team discovered that certain API queries were returning less results than expected. This allowed the team to refactor the scraper.

Testing with bash/python

In terms of black box testing, three bash scripts were used to test the results received from the Puppeteer queries. This was done to automate the process of testing and assess the quality of the scrapping algorithms through regex tests.

The tests addressed these issues:

- Key terms appear in retrieved articles
- Locations appear in retrieved articles
- Retrieved articles are within the start and end date

Testing key terms appear in the retrieved articles

The first bash script was used to test if the diseases provided to the server as key terms appeared in the retrieved articles. The script was used to iterate through the list of diseases provided to us (diseaseList.json) and used them as keywords to generate queries to our server. This bash script used a simple command line json parsing tool called ‘jq’ to convert the downloaded text from the server, retrieved through curl, and parse it into a .json file. The .json file (containing the retrieved articles) then undergoes a series of regex tests to ensure that the disease keyword is mentioned in all of the articles scrapped as a result of the query.

The regex tests also return several counts that prove useful in assessing the quality of the results received, those being; the total number of articles received, the total number of articles mentioning the disease keyword, and the total number of mentions of the disease keyword. Given an optimal scraping solution, the total number of articles mentioning the disease keyword should be equal to the total number of articles, and the total number of mentions of the disease keyword should be greater than the total number of articles. This script produces a total of 73 .json files with the parsed query results, one for each disease provided in diseasesList.json.

Testing locations appear in the retrieved articles

The second bash script was used to iterate through a much smaller list of provided diseases, and tested multiple locations for each disease. Similar regex tests were performed on the results received, to ensure that the articles were illustrating events in the tested locations. This script produces a total of 27 .json files with the parsed query results, nine tests for each of the

three diseases tested, three queries testing a country location, three queries testing a state location, and three testing a city location.

Testing retrieved articles were published within start and end date

The third bash script was used to iterate through the same smaller list of provided diseases as the second script (three disease keywords), and changed the start date and end date variables for each disease. Similar regex tests were performed on the results received, to ensure that the articles were published on a date that is equal to, or after, the specified start date, and equal to, or before, the specified end date. This script produces a total of 15 .json files with the parsed query results, five tests for each of the three diseases tested, testing endDate - startDate gaps of one year, six months, one month, one week and one day, whilst the endDate is always the present day.

Logs

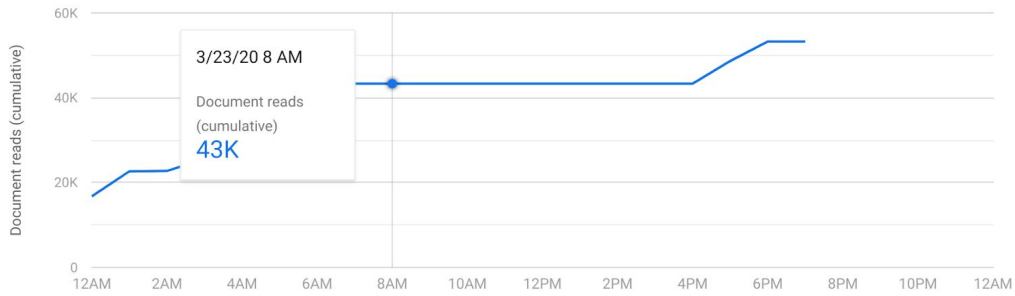
The team introduced two types of logs:

- JSON Snippet for the developer returned with each request
- Firebase log of all reads from the database

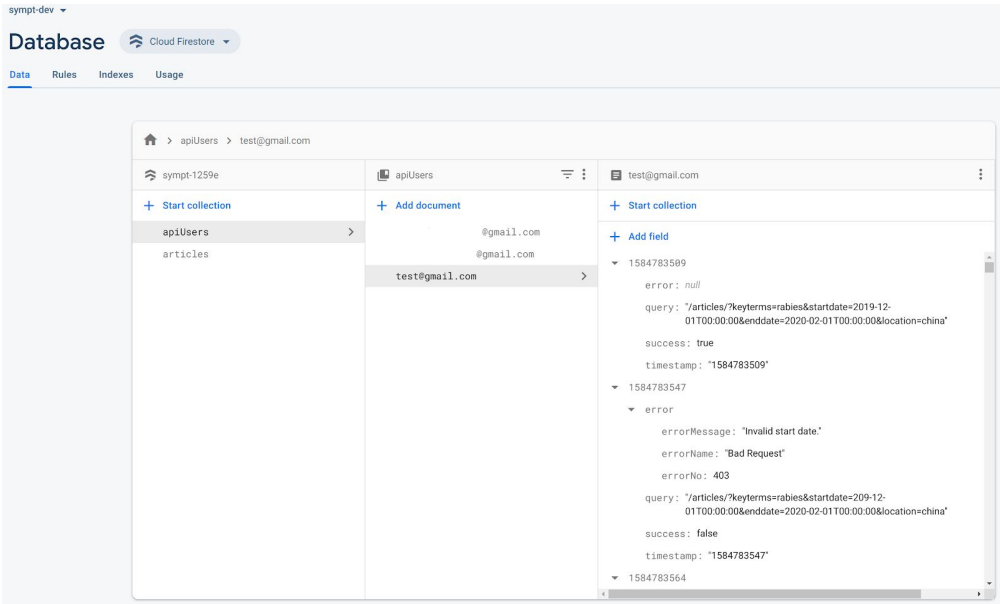
Below is an example metadata object


```
"metadata": {
  "team": "Beams",
  "time_accessed": "2020-03-24T02:31:56",
  "data_source": "ProMed",
  "total_articles": 5
},
```


Below is a snippet from the Firebase Database Usage Page



We can also track individual users and their requests via firestore database

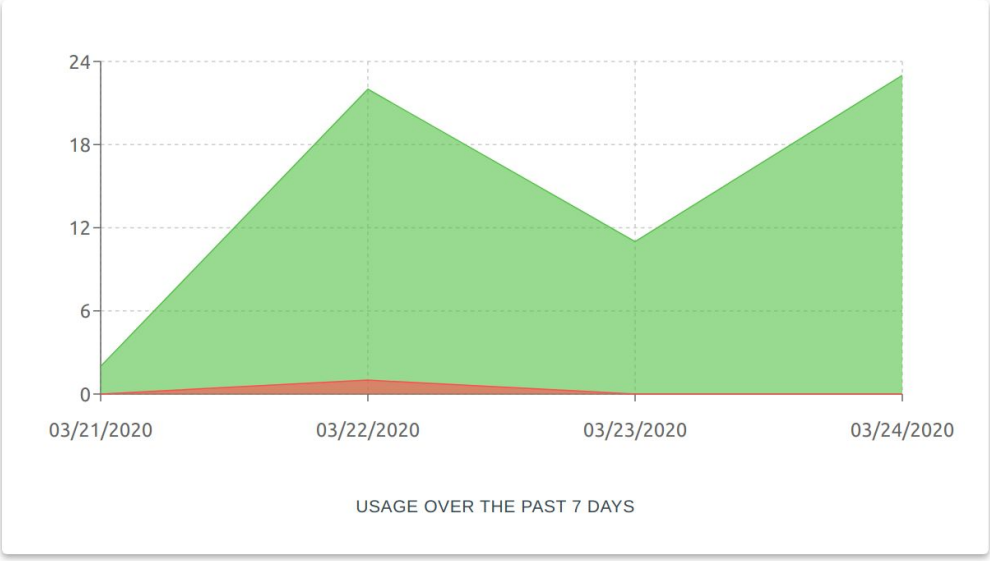


In addition to this, we have made the logs of each developer available to them via the API Dashboard shown below so that they can track their own API usage including the time the query was made, the query string, the error result and the response.

Key:

- Successful Queries (Green)
- Unsuccessful Queries (Red)

Usage Analysis



Logs

Time	Query	Error	Response
March 24, 2020 12:32 PM	?startdate=2000-06-02T00%3A00%3A00&enddate=2020-06-02T00%3A00%3A00&keyterms=rabies&count=5&page=0	None	SUCCESSFUL
March 24, 2020 12:32 PM	?startdate=2000-06-02T00%3A00%3A00&enddate=2020-06-02T00%3A00%3A00&keyterms=2019%20nCoV&count=5&page=0	None	SUCCESSFUL

Testing the Sympt App

The testing of this was done manually by quality assurance team members. Individuals went through screens on the app and manually ensured that, for example, News Feed and Promed Mail Articles feed were relevant and within the time frame which was selected.

Testing the SIR predictions model

The testing for the SIR predictions model was done manually, with COVID-19 as the disease example used in testing. Due to the small subset of information and case numbers that we had to test on, it was not worthwhile to make a testing bash/python script to test for us. Consequently, and also due to the limited information and case numbers we had with COVID-19 (two months worth of daily case numbers), we ran our SIR implementation once for each day in the past. This was done to ensure that the functions mapped with the SIR model are continuous and returned reliable data, with no obvious outliers. This testing process was executed again for the functions mapped with the social distancing contact rate, ensuring that the increase/decrease of daily cases matched with the most recent case data available from Australian government sources.