Testing Documentation

Epidemic Eagle

Testing Documentation

Prepared for SENG3011

Created by Aaron Shek, Sahibpreet Bassi, Daniel Steyn, Liam Treavors and Mihail Georgiev

Context

Testing our API ensures that the API is correctly implemented according to standards set by the user. Without testing our API, then the client would be dissatisfied with the product, as well as reducing the reputation of the API.

There are two ways our API is tested: manual testing and automated testing.

Manual testing is done to test the user experience via human observation of how data is sent to the user, using tools such as Swagger.

Automated testing is also included as an aspect of our API, to inform our team when an error occurs in the development environment. Another feature of automated testing is the ease of repeated running of tests to reduce human error and labor. In our API, we will be using GitHub Actions as a pipeline as well as another check in Heroku for the website content.

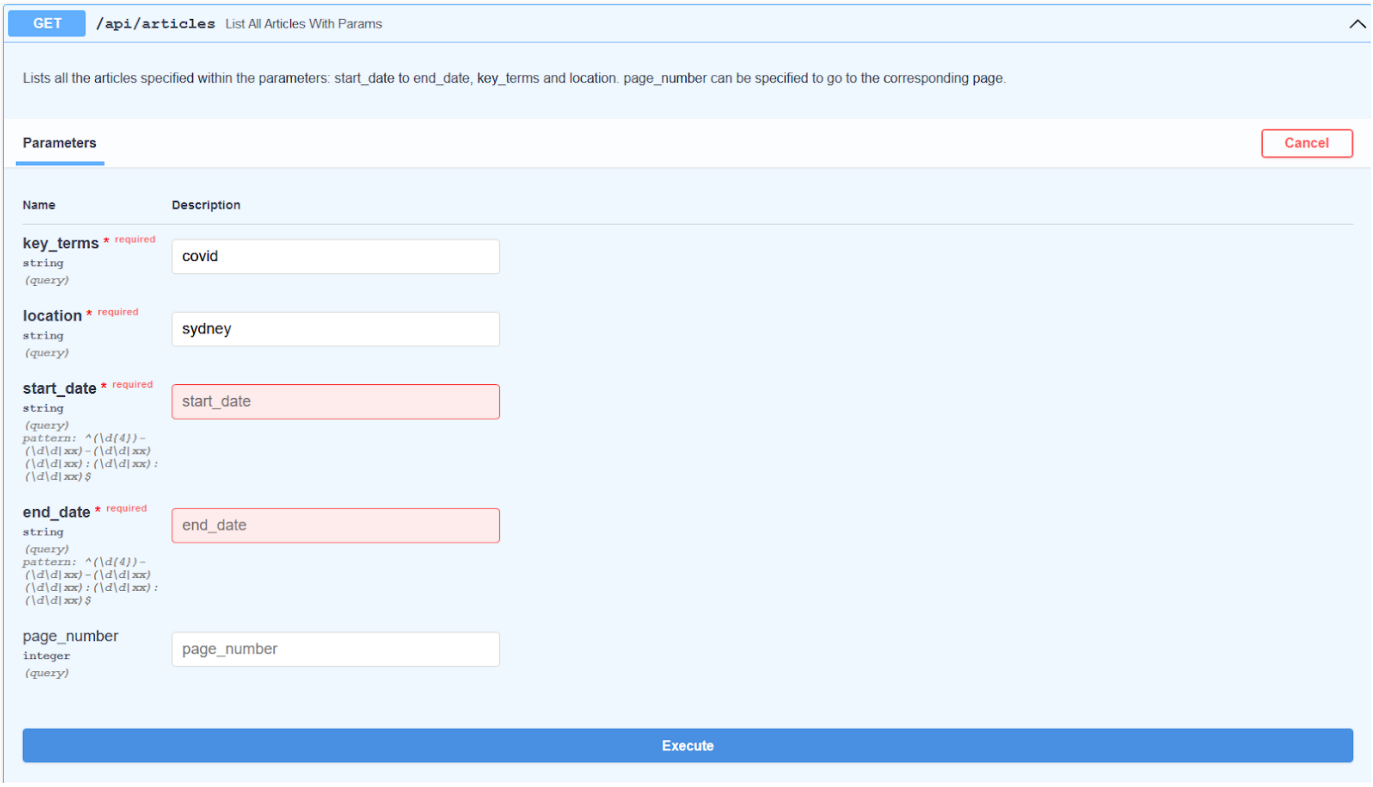
Manual Testing

We will be using OpenAPI (formerly known as Swagger) to test the functionality of our API routes from the user experience. Since we are using FastAPI, the OpenAPI is automatically generated for us, however it will still require manual checking to ensure the behavior of routes is correct.

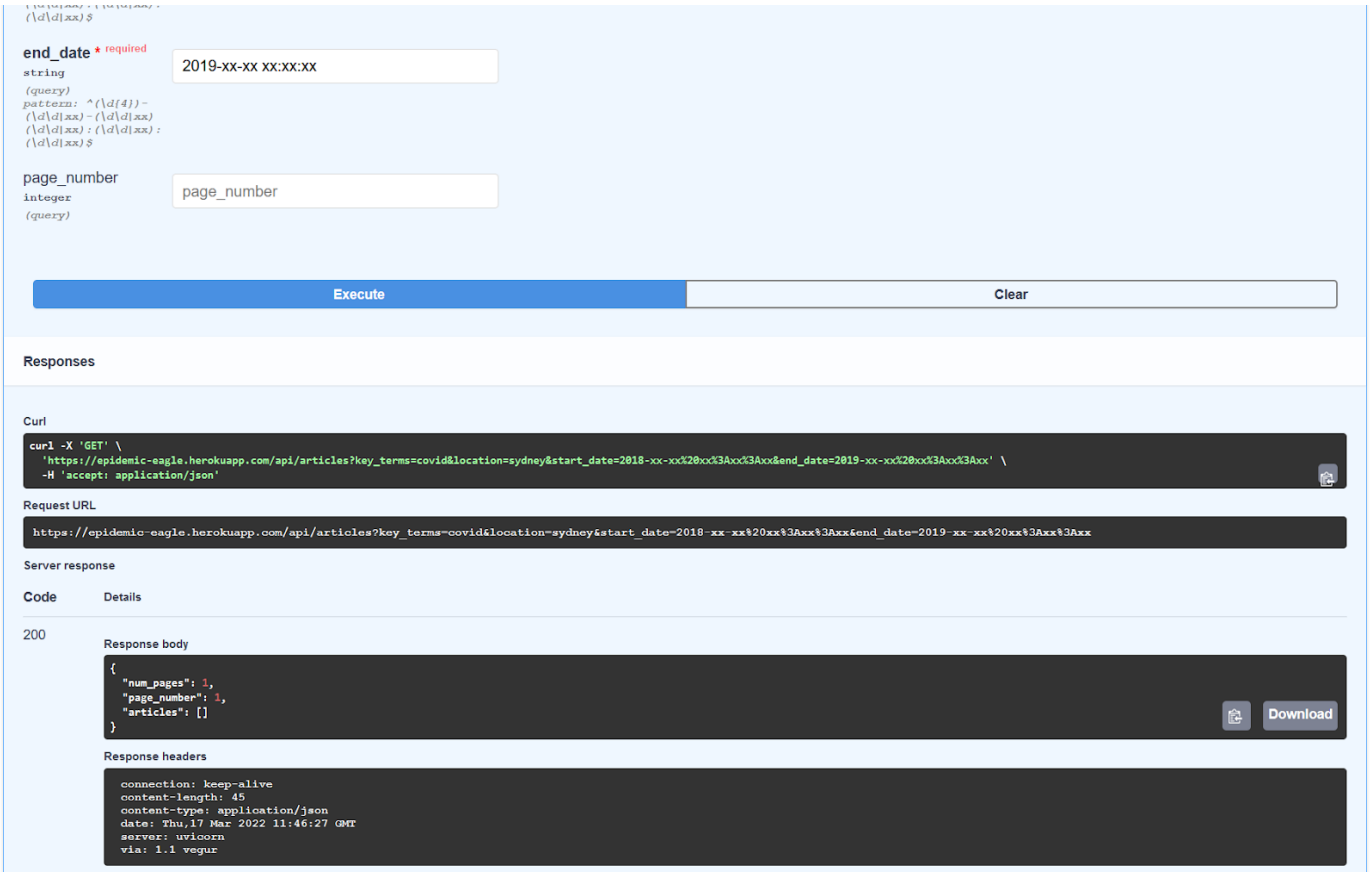
For the hosting of our OpenAPI, it is located at http://epidemic-eagle.herokuapp.com/docs .

OpenAPI is a user-friendly and open-source tool to test the JSON responses from our routes.

One benefit of Swagger is that it shows which parameters are required in order to get a successful response from the route.

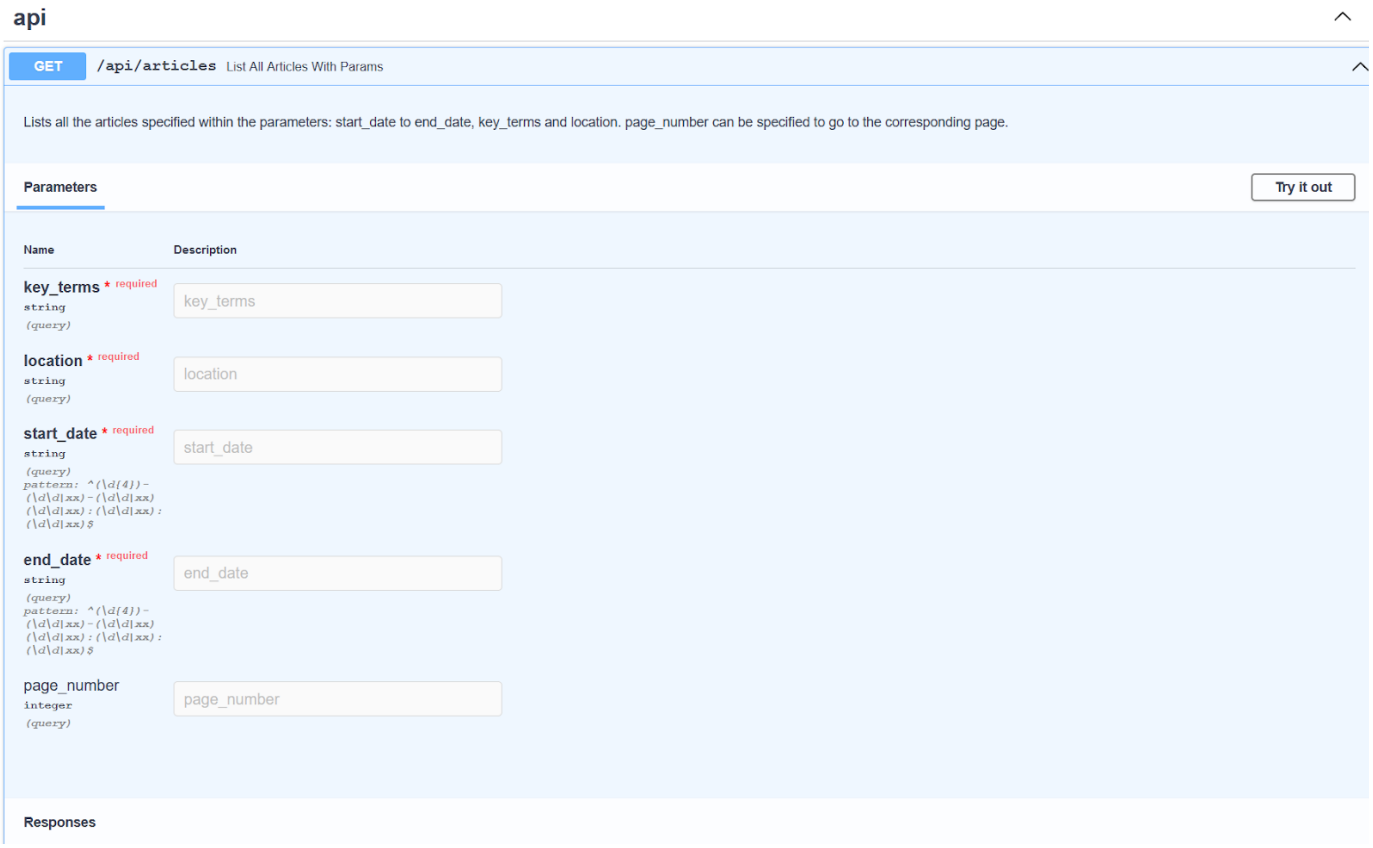


Additionally the response body and headers are also shown, to provide extra information to the client.

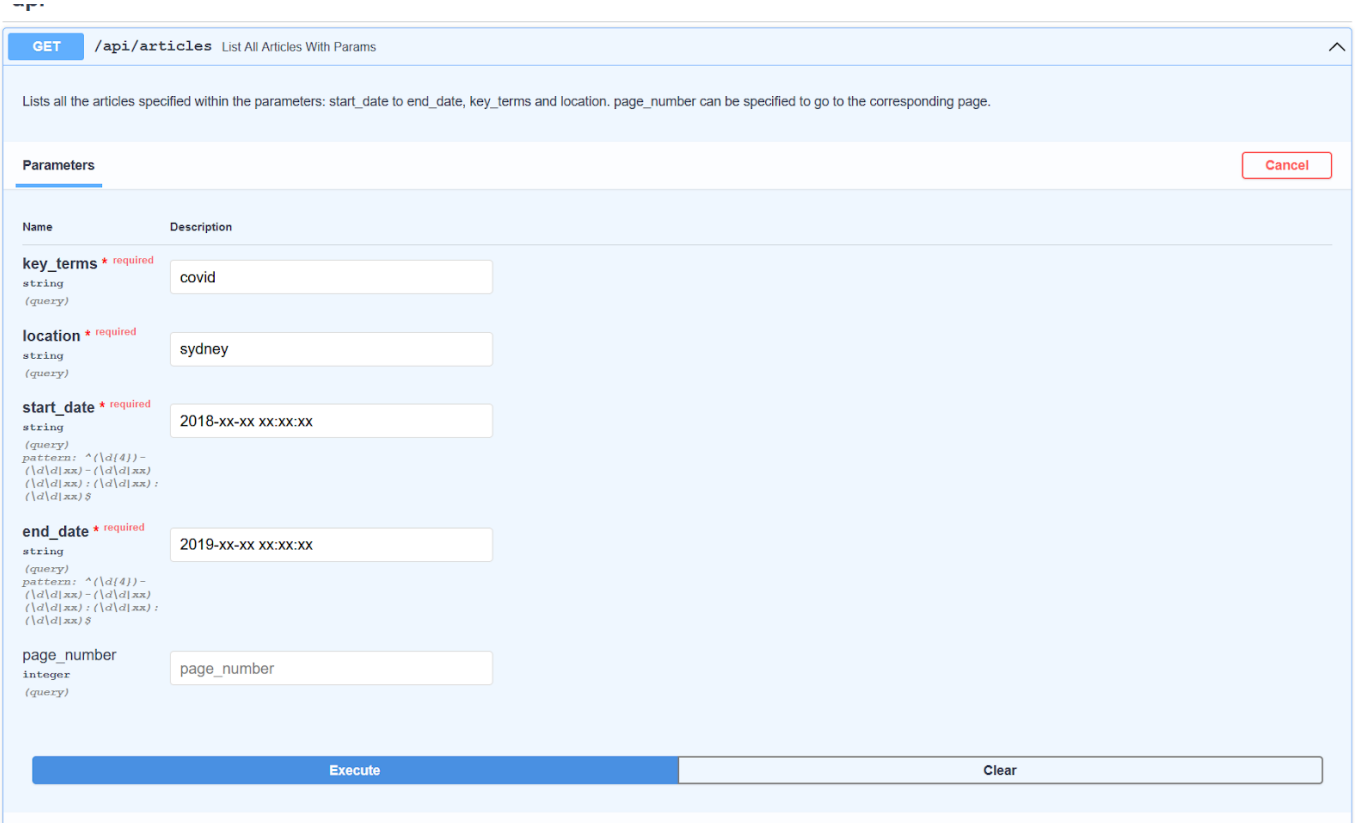


Example OpenAPI route test for /api/articles.

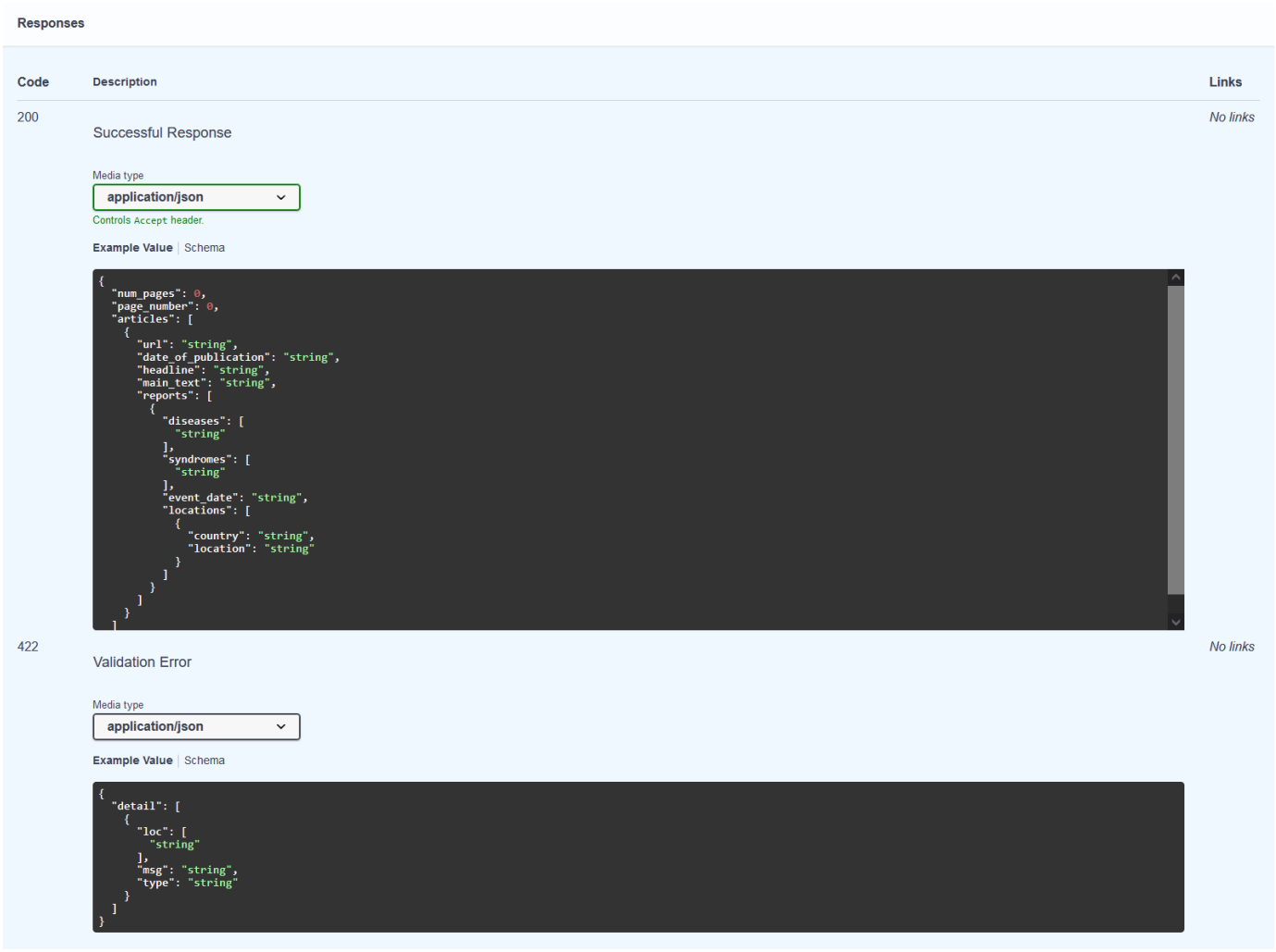
First of all, to generate a GET request we use the url “/api/articles” and pressthe “Try it out“ button.



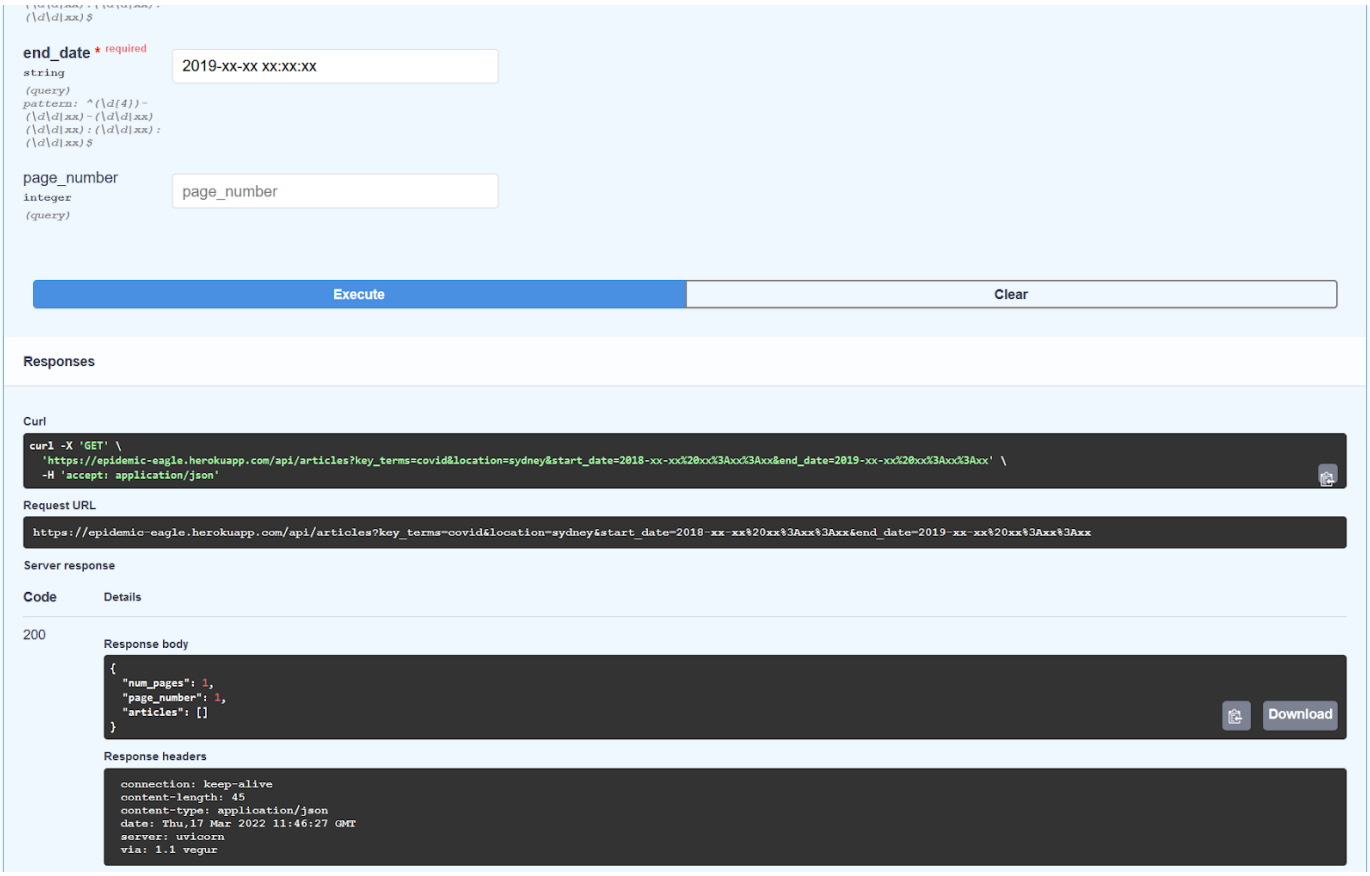
Firstly we enter the required parameters, in the format specified.



There are two types of status codes, a 422 Error for invalid parameters or a 200 successful response with a list of articles with the parameters.



Then we can push theexecutebutton.



In this instance, it responds with apassedresult and a response code200.

By using manual testing, we can get a sense of how a user would interact with our API.

And by using OpenAPI, we can visualize how a client uses our routes and update them if they are not user friendly to enter. It is an effective tool to test user experience and to quickly change and send parameters.

Automated Testing

To ensure that the software we are deploying is usable and functional code, we have implemented a GitHub pipeline. This will run multiple tests every time we push our code to main, and if these some of these tests fail then the code will not be pushed to the website. By using a pipeline,

GitHub Pipeline

First, the pipeline ensures that the Operating System being used when running the tests is Ubuntu. This is to make sure that the tests have consistent results to the web host, since we are using a linux based system. The pipeline then checks that it is in the correct branch, so that the tests run on the correct version of our software.

```
13  # A workflow run is made up of one or more jobs that can run sequentially or in parallel
14  jobs:
15    build:
16      strategy:
17        matrix:
18          python-version: [3.10.2]
19      runs-on: ubuntu-latest
20
```

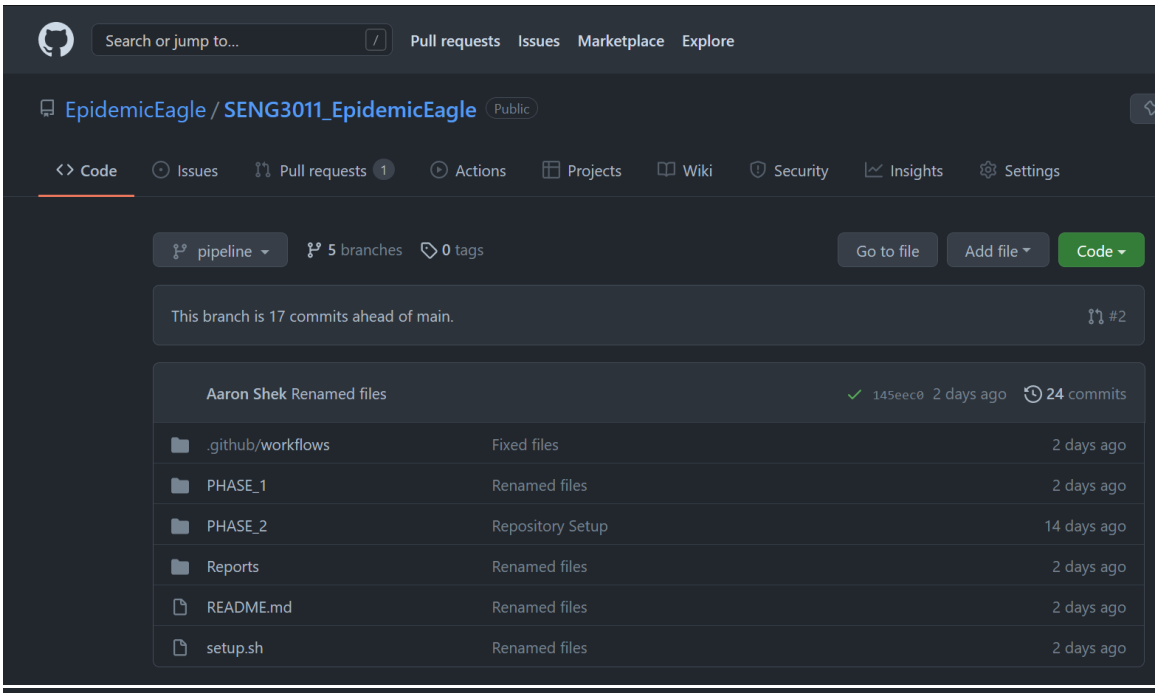
Using Ubuntu, the pipeline then installs python version 3.10.2, our desired version for all tests and software. Different versions provide slightly different functionality, so this also maintains consistency.

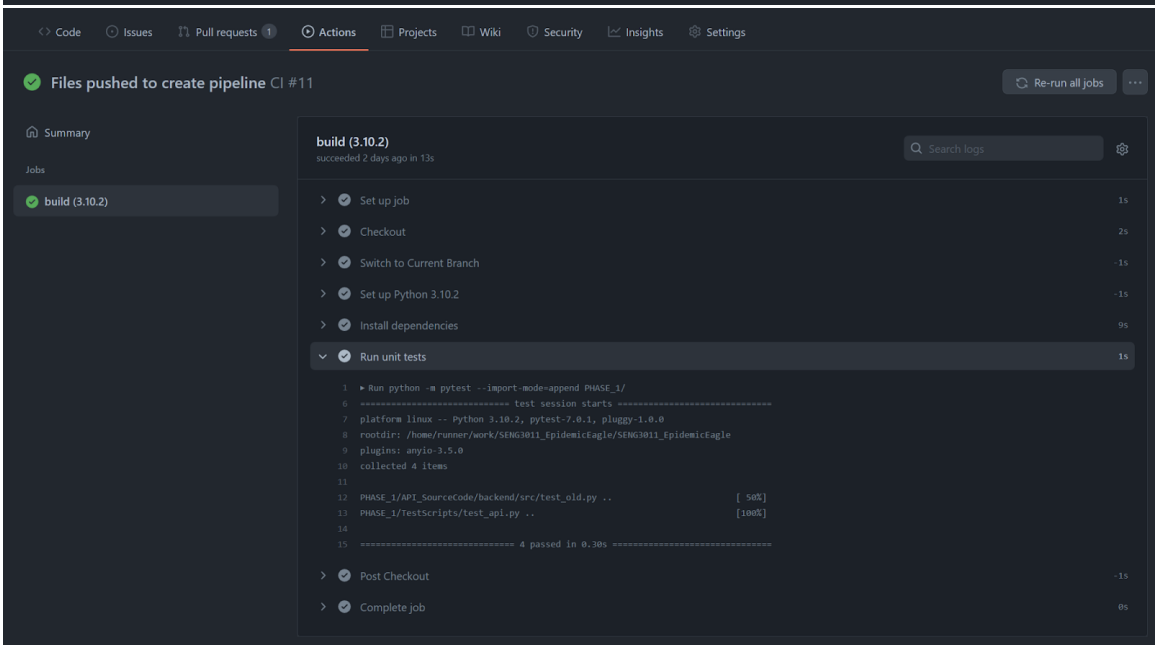
```yaml
    # Steps represent a sequence of tasks that will be executed as part of the job
    steps:
      - name: Checkout
        uses: actions/checkout@v2
        with:
          fetch-depth: 0

      - name: Switch to Current Branch
        run: git checkout ${{ env.BRANCH }}

      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v1
        with:
          python-version: ${{ matrix.python-version }}
```

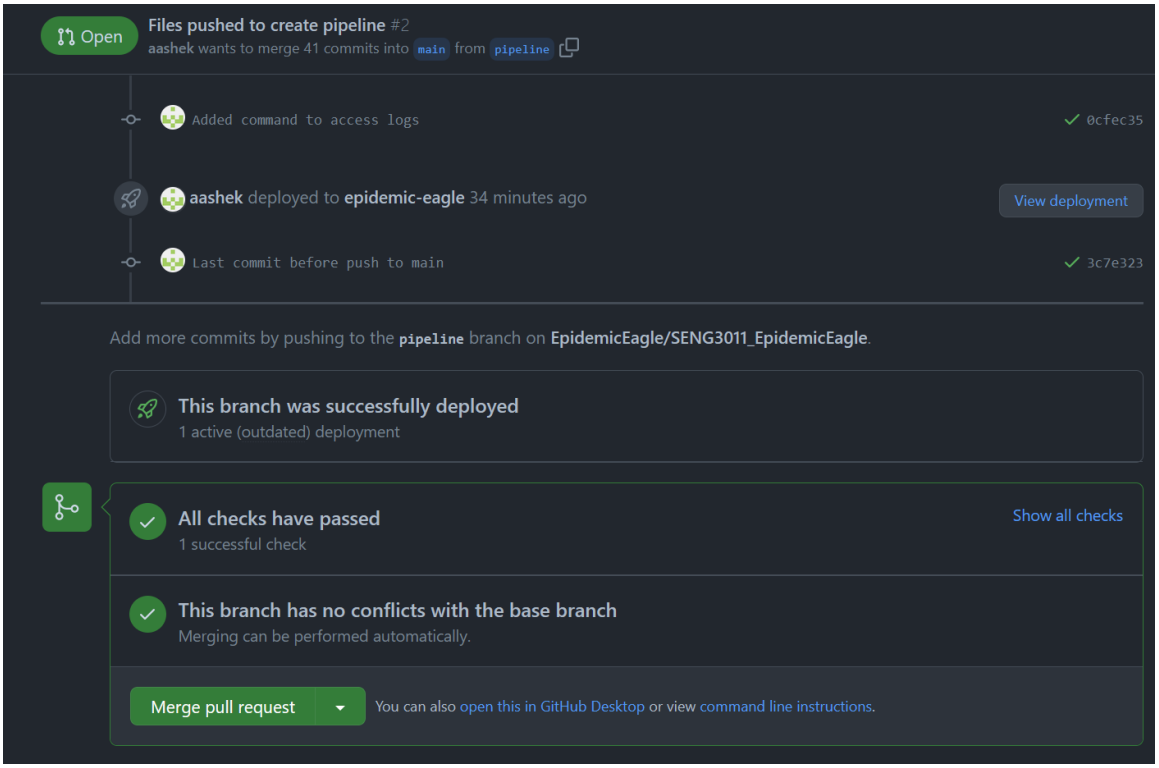
We then install a dependency named pip, which is a helpful tool to manage and download packages or libraries. A list of all of these is found in requirements.txt, and pip is used to make sure the correct version of these are all installed. Pip is also used to install pytest, which is how the tests will finally be run after the setup stage.

```yaml
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
          pip install pytest
      - name: Run unit tests
        run: python -m pytest --import-mode=append PHASE_1/
```

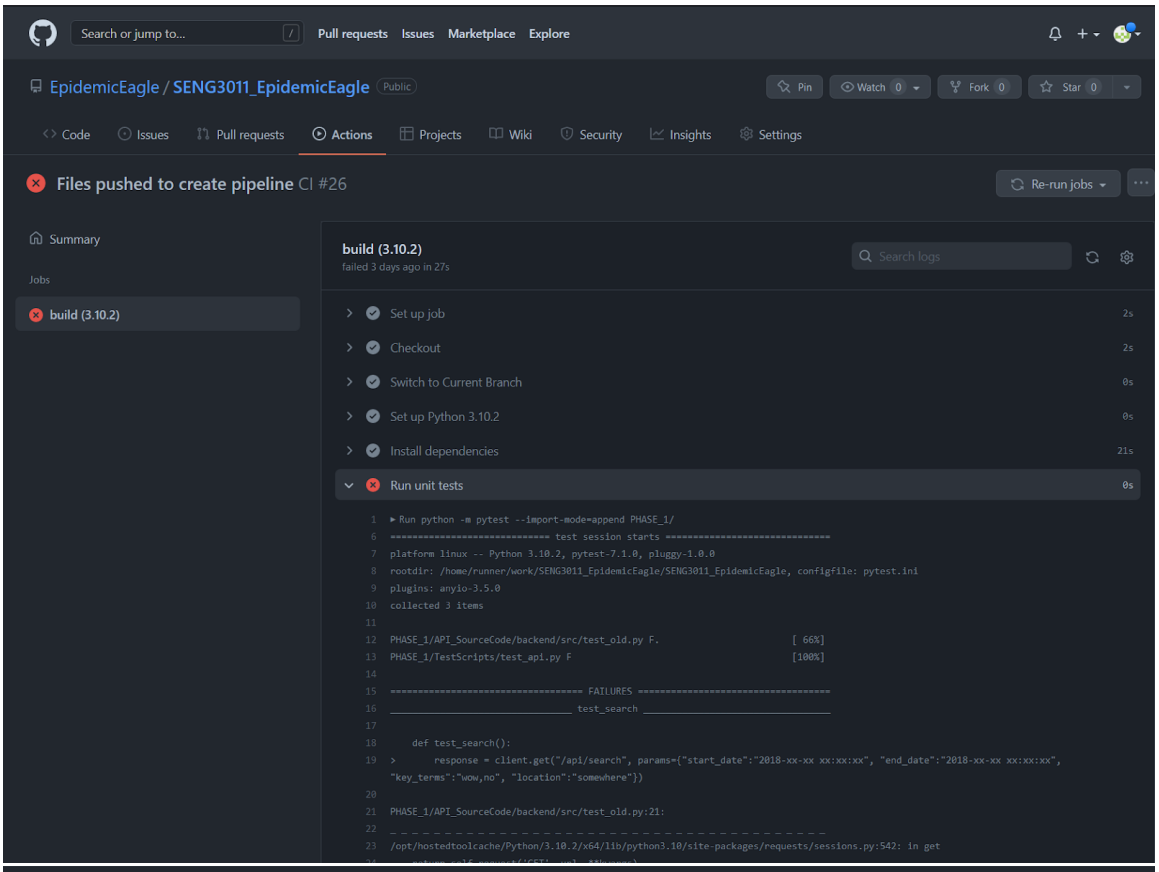
Now that the correct environment is installed, the pytest command is used to run all of the tests found in python files with the prefix ‘test_’. If all of these tests pass, the pipeline will succeed and will show a green tick on GitHub. If even one of these tests fail, then the pipeline will fail and will show a red cross on GitHub, and the code will not be pushed to deployment. The pipeline can be checked to see which test caused the failure, so that a fix can be implemented.


Search or jump to...
Pull requests
Issues
Marketplace
Explore
EpidemicEagle / SENG3011_EpidemicEagle
Public
Code
Issues
Pull requests 1
Actions
Projects
Wiki
Security
Insights
Settings
pipeline
5 branches
0 tags
Go to file
Add file
Code
This branch is 17 commits ahead of main.
#2
Aaron Shek Renamed files
145eec0 2 days ago
24 commits
.github/workflows Fixed files 2 days ago
PHASE_1 Renamed files 2 days ago
PHASE_2 Repository Setup 14 days ago
Reports Renamed files 2 days ago
README.md Renamed files 2 days ago
setup.sh Renamed files 2 days ago


Code
Issues
Pull requests 1
Actions
Projects
Wiki
Security
Insights
Settings
Files pushed to create pipeline CI #11
Re-run all jobs
Summary
Jobs
build (3.10.2)
build (3.10.2)
succeeded 2 days ago in 13s
Search logs
Set up job 1s
Checkout 2s
Switch to Current Branch 1s
Set up Python 3.10.2 1s
Install dependencies 9s
Run unit tests 1s
▶ Run python -m pytest --import-mode=append PHASE_1/
test session starts
platform linux -- Python 3.10.2, pytest-7.0.1, pluggy-1.0.0
rootdir: /home/runner/work/SENG3011_EpidemicEagle/SENG3011_EpidemicEagle
plugins: anyio-3.5.0
collected 4 items
PHASE_1/API_SourceCode/backend/src/test_old.py .. [50%]
PHASE_1/TestScripts/test_api.py .. [100%]
4 passed in 0.30s
Post Checkout 1s
Complete job 0s



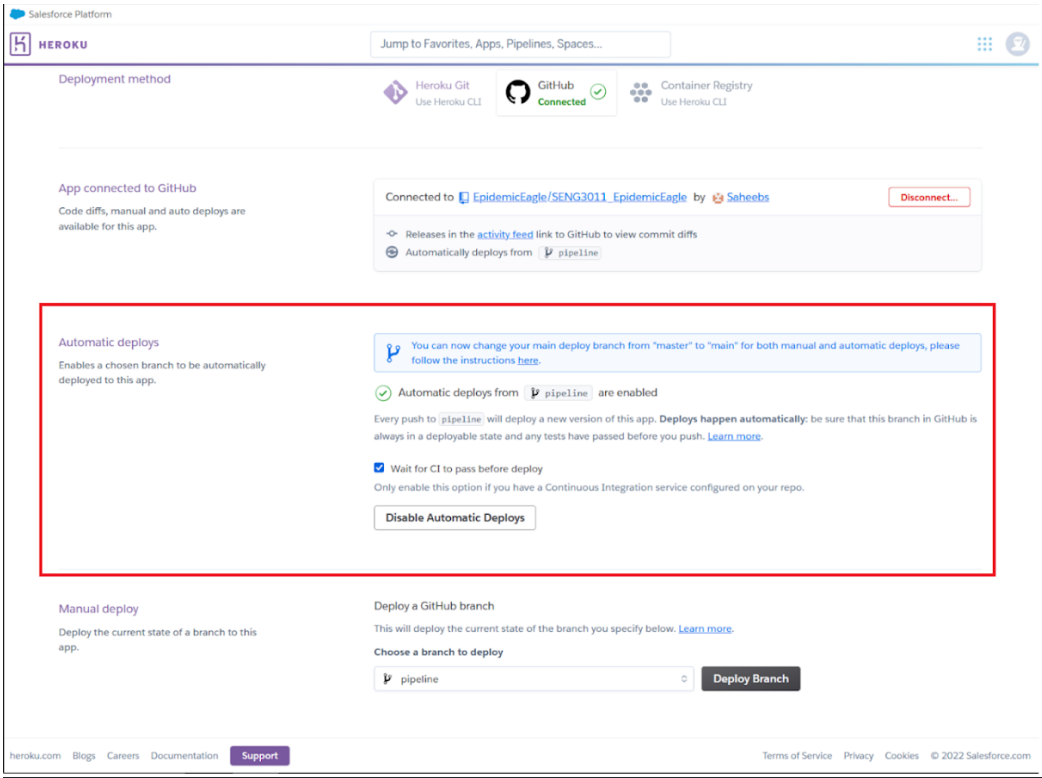
Successful Pipeline

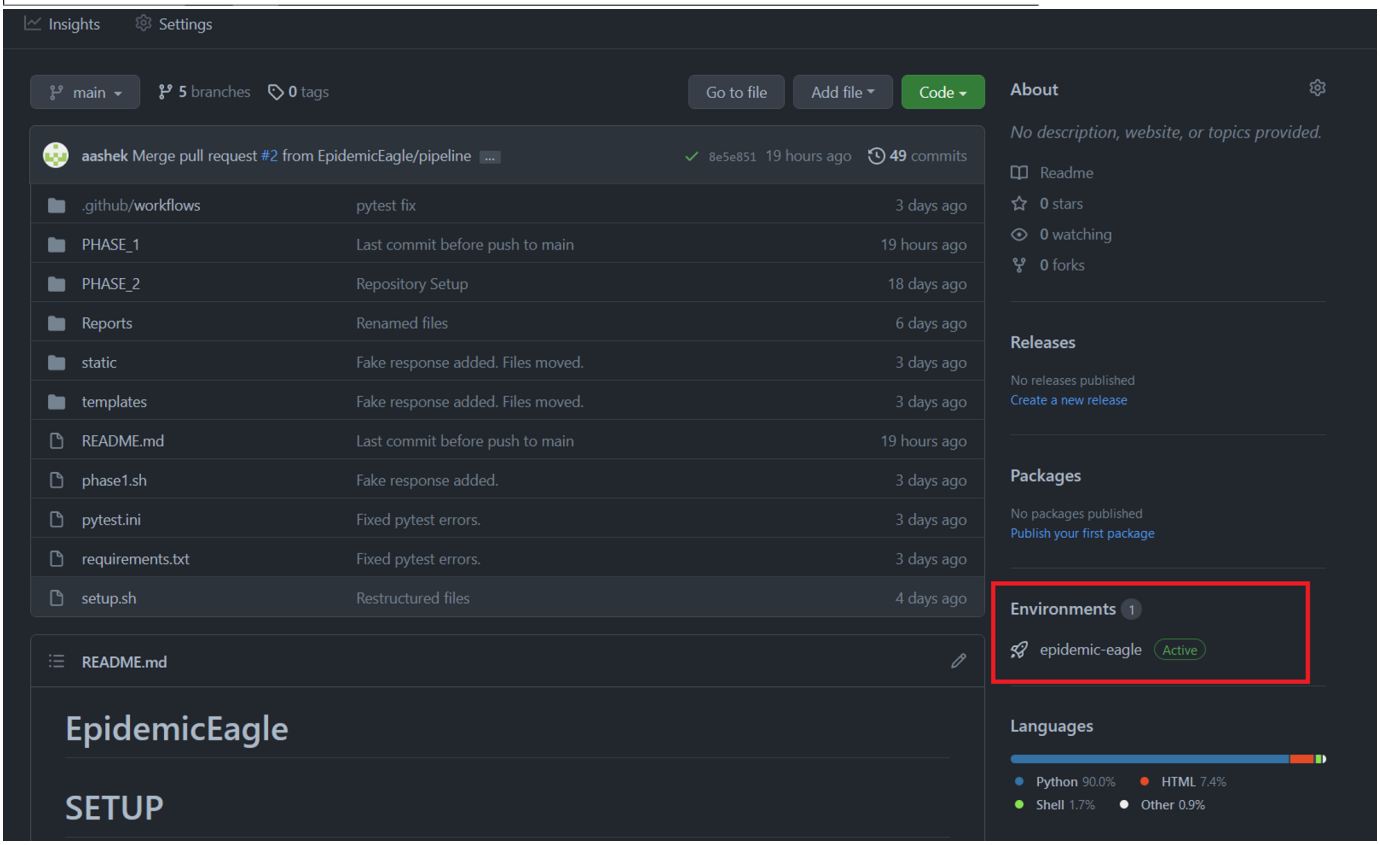


Unsuccessful Pipeline

Automatic Deployment

When the GitHub pipeline is successful, then Heroku will automatically restart and update the website with the new code. This was established to reduce manual labor required to restart the server due to new code changes.





Overview of Tests

Testing helps provide proof of correctness for software. Although our team has in-depth knowledge about setting an environment for automatic and manual testing, we ran out of time to write testing functions for our API. However some sample ones are provided below.

Unit Testing

Unit testing is one of the most important parts of testing. By ensuring that each small part of our software is able to work independently on a consistent basis, we can be more confident that they will work together in our system to achieve the desired result. If we were simply to test the system all at once, it would be difficult to pinpoint any errors that may occur. Unit testing means that we know exactly what parts of the software are breaking, saving time and resources when searching for problems.

Parameter Testing

One of the tests is a parameter test, meaning we test the parameters and whether they are the right type. This is necessary as we cannot guarantee that the users will put in the correct parameters. If they manage to put incorrect parameters in, our software needs to be able to return an error to the user, with an explanation of what caused the error, helping protect our code from breaking.

For example, a call to /api/articles/"1" would return a 422 Error for Invalid Type.

A helper function is described below.

```
def arg_test(url):
    params={"start_date":"2018-xx-xx xx:xx:xx", "end_date":"2019-xx-xx xx:xx:xx", "key_terms":"covid, measles", "location":"Shanghai"}

    response = client.get(url, params =  {"start_date":"2018-xx-xx xx:xx:xx", "end_date":"2019-xx-xx xx:xx:xx", "key_terms":"covid, measles"})
    assert response.status_code == 422

    response = client.get(url, params =  {"start_date":"2018-xx-xx xx:xx:xx", "end_date":"2019-xx-xx xx:xx:xx",  "location":"Shanghai"})
    assert response.status_code == 422

    response = client.get(url, params =  {"end_date":"2019-xx-xx xx:xx:xx", "key_terms":"covid, measles", "location":"Shanghai"})
    assert response.status_code == 422

    response = client.get(url, params =  {"start_date":"2018-xx-xx xx:xx:xx", "key_terms":"covid, measles", "location":"Shanghai"})
    assert response.status_code == 422

    response = client.get(url, params = {**params, 'start_date': 'two days from now'})
    assert response.status_code == 422

    response = client.get(url, params = {**params, 'end_date': 'today'})
    assert response.status_code == 422

    response = client.get(url, params = {**params, 'start_date': '2019-20-20'})
    assert response.status_code == 422

    response = client.get(url, params = {**params, 'end_date': '2019-20-20'})
    assert response.status_code == 422
```

Then this helper function is called for the three routes with these required parameters.

```
29  def test_wrong_args():
30      arg_test("/api/search")
31      arg_test("/api/articles")
32      arg_test("/api/reports")
33
```

Acceptance Testing

Another type of testing the acceptance tests, which ensures that correct parameters input achieve the desired output. This means that when users are calling the API, they find all the information they are looking for, be it multiple reports on a common disease over the past few years or a single report on a new disease in the last month. By comprehensively testing with correct parameters with results we know to be true, we ensure that our users are satisfied, thus accepting that the API works.

Limitations of Testing

Some parts of our API cannot be tested, due to the variance in results. This is namely when new ProMed articles are released, as they won't be in our test suite, and the new articles may change the results of previous tests. Thus, it is difficult to test new material that ProMed releases.

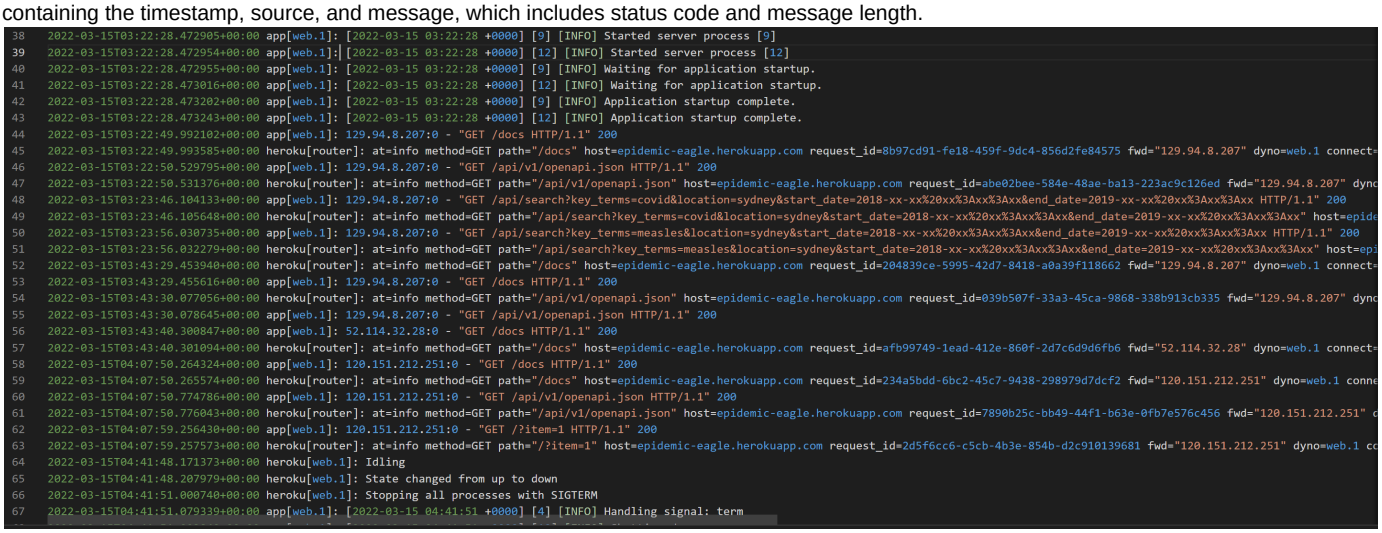
Web Scraper (Scrapy)

Currently we cannot run automated tests on our web scraper. It would be too difficult to test daily results, since we do not know how many posts were created on a single day. We can do some limited manual testing by checking our database each day to see if new entries are being made, though this is not guaranteed to be correct as we won't be able to ensure that all reports and articles are being generated. Manual testing also takes lots of time, which is inefficient and often unviable as the team already has limited time to work on the project.

Log Files

Logs are an extremely helpful feedback tool when deploying an API. They can keep track of requests made to the API and store information about each request.

Heroku offers an inbuilt logging function whenever calls are made to the API it is hosting, saving the logs in Heroku. Using the Heroku CLI, we can read these logs and store them in a text file. By typing in "Heroku logs" into the CLI, Heroku will display the 100 most recent logs. A "-n 200" tag after the command will display the 200 most recent logs and can be changed to the desired number of logs. Heroku logs store information containing the timestamp, source, and message, which includes status code and message length.



Benefit of Logs

Logs are useful as they contain runtime information for users. By analyzing this information, we can determine how to optimize our software so that users have a more enjoyable experience. This could be anything from how much traffic we are receiving at certain times, how fast each request is being processed, which requests are being called the most and if processes are returning errors.

Analyzing the logs will allow us to see what routes are called the most, and allows us to further improve the usability of those routes. It can also show us what routes are not being used, and determines whether or not the functionality of a route is viable to a client.