In the Testing Documentation report, describe the testing processes used in the development of API, referring to the data and scripts included in Phase_1 folder. This should describe your testing environment and/or tools used, and limitations (e.g. things that are not tested). Describe your testing process i.e. how your team conducts testing using the test data (e.g. in which order) and an overview of test cases, testing data and testing results. Describe the output of testing and what actions you took to improve the test results.

Testing Environment and tool

For the API testing, our team went for two approaches to test, namely Postman and Pytest. We decided to have **Postman** as our client end testing, for simple testing of the responses and the right json information. Postman provided us with a clean environment to call our API, however the testing was in Javascript. We relied more on Pytest as our testing tool as it was much easier to compare and request for API response with the generated output files.

With the addition of **Pytest, it** also helped with the backend of the API, allowing us to make more vigorous test cases and included edge cases and a much more conventional approach to testing our API.

Limitation

The main limitation for testing is the dynamic database that keeps updating every hour. We scripted our web scraper to regularly update the database, hence at times new information may appear in our API response, when we tested before with old static data. To overcome this limitation we decided we used older dates or that articles will not be updated since they are much older and less likely to be updated.

Testing Process

We first test our API on Postman to see if the requests were functional by testing for the correct response code, and keywords in the json response. Since we have only one endpoint, we tested for all possible response cases including 200, 400 and 404.

We then move on the Pytest to test the validation of the API. We first started with checking the validation of the urls, making sure that the wrong inputs of the query parameter will lead to an error message we provide for them. This included date validation and checking for all possible edge cases for the date. Since dates involve the

use of “xx” as placeholder for the datetime format, we had to test the possible cases where the start and end are ambiguous. The remaining parameters location and key_terms can easily be tested with either a fail or success response. The next step was to check if the json responses of the api were correct. This required a file of outputs json that we generated from the scraper and tested with the api calls. A simple equivalent check was all that was needed to pass the test, yet also ensuring that the date of publications were inrange of start and end date, location was valid and the article was relevant to the key term. These tests included our unit tests as we only had one function leading to the api.

Finally, we test the performance of our api by comparing the time of our api. We had test cases ranging on one day - one month.

Overview

Postman

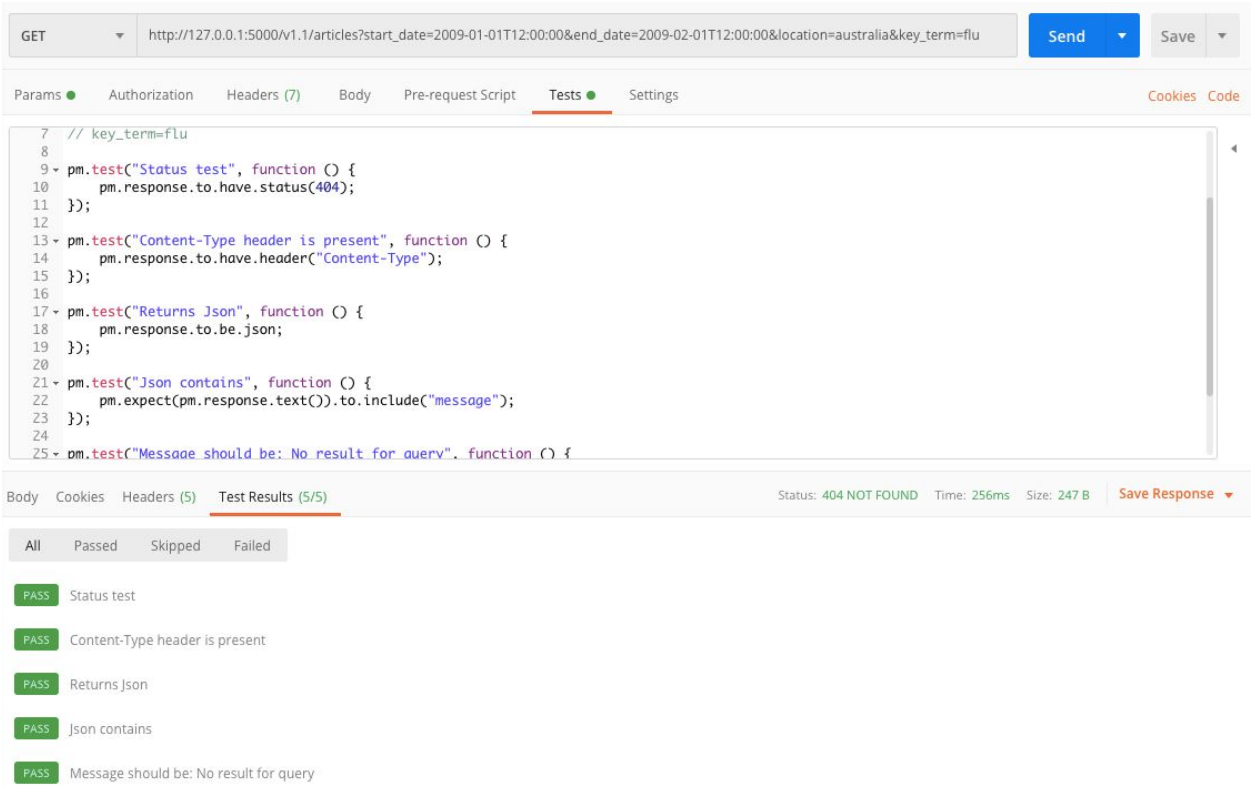

Figure 1.1 Postman Test Environment

This was a setup of our Postman testing. It included a url at the top for calling our api

And an Javascript editor for writing postman test cases. When the button Send is press, it will run the test.

Pytest

```
def test_success():
    PARAMS = {
        "start_date": "2020-01-01T12:00:00",
        "end_date": "2020-02-01T12:00:00",
        "location": "Australia",
        "key_term": "coronavirus"
    }

    r = requests.get(url = URL, params = PARAMS)

    with open('output_response/test_success_output.json') as f:
        expected = json.load(f)

    assert(r.request.method == "GET")
    assert(r.ok)
    assert(r.status_code == 200)
    assert(r.json() == expected)
```

Figure 1.2 Pytest Function

For Pyest, all we had to do was write functions and assert to check that our response and json output was correct

Improvements

Through testing we have found results that needed improving on such as testing with x's when choosing dates and we overcame this problem by having our backend autofill these x's with values that made sense so that end date would always be later than start date. This would ensure that even with strange input with x's, valid results would still be produced unless the user itself put dates that didn't make sense.

Some results would give the wrong response as well which we improved on by changing code in the backend to produce the correct results such as when having x's for years should've given a 400 response for incorrect data entry but it gave a 404 response, so by testing we improved on this to give the correct result.

Also through testing we could find parts of our api not implemented correctly and correct this. For example having our query need key_terms and locations needing parameters during testing when these should be optional. So by finding out from testing we were able to change our code in order to allow for these fields to be optional.