SENG3011 Deliverable 2 - Testing report

Group: FatDonkey

Members: Flynn Zhang, Edwin Tang, Frank Su, Josh Rozario

Testing Environment

In order to thoroughly test our system we developed testing scripts to cover both the internal logic flow of the various methods written for our backend functionality and also the response by our API endpoint to various requests

In testing the back-end functionality, unit testing scripts were developed which focused on verifying the internal logic of the various functions that we developed. This type of testing script was used to ensure that the methods that were written to extract entries from a MongoDB database with specific requirements were able to return the correct entries. These testing scripts allowed us to isolate the behaviour of our backend methods which allowed us to easily pinpoint any logical errors in our code. This allows us to ensure with certainty that the code was functionally correct. We developed tests for our functions which filtered by **location**, **disease** and **region**. (outbreak_disease_test.py, outbreak_location_test.py, etc.)
Sample test for filtering results by disease

```
def test_outbreak_disease_covid():
    res = disease_filter('COVID-19', col)
    for entry in res:
        assert entry['disease'] == 'COVID-19'
```

We also developed HTTP testing scripts for our Flask server to ensure the correctness of API calls. The feature testing scripts verified that the response json and response codes from a request to our API endpoint was as expected. These tests formed the bulk of our testing as they tested the API in its entirety. These tests are in the outbreak_http_test.py file.

Sample test for region parameter

```
def test_outbreak_region_http(url):
    outbreak_param = {
        "location": '',
        "disease": '',
        "start date": '',
        "end date": '',
```

```
        "region": 'Asia',
        "end_index": '',
        "start_index": ''
    }

    resp = requests.get(f"{url}outbreak/",
params=outbreak_param)
    print(resp)
    articles = resp.json()
    assert resp.status_code == 200
    assert articles[0]['region'] == 'Asia'
```

The primary focus of this testing environment was to help us to easily and quickly find bugs and errors in our code which we could then fix to ensure our system is consistent in returning the correct information to the user. We aimed to test every branch of our code (i.e. test every single line of code in our API) so as to achieve a high test coverage.

Our testing scripts are in the \Phase1\TestScripts folder.

We also conducted informal testing through the Swagger UI while we were developing the API as it provided a quick way to test new functions.

While testing of our natural language processor occurred, it was very limited as it had to be carried out manually. This is detailed in the Limitations section of our report.

Test cases/data

In developing our testing cases we made sure to emulate a variety of potential inputs by an end user. Our test cases supplied valid and invalid inputs and checked if the response code and response message/json object was as desired. In the above example, we checked if the response code was 200 and if the region of the first article was Asia. If invalid inputs were supplied such as if the start_index was greater than the end_index, the tests would check that the response code would be 400 and that the error message was the desired one. Our test cases would test the API's ability to filter articles based on location, disease, time period, region, start_index and end_index. Tests that used a combination or all of these filters were also written.

Sample test for incorrect inputs

```
def test_outbreak_start_higher_end(url):
    outbreak_param = {
        "location": 'China',
        "disease": '',
        "start date": '',
        "end date": '',
        "region": '',
        "end_index": '5',
        "start_index": '8'
    }
    resp = requests.get(f"{url}outbreak/",
params=outbreak_param)

    assert resp.status_code == 400
    msg = resp.json()
    assert msg['message'] == 'start_index cannot be
greater than end_index'
```

Our test data was manually written for each test case as they required specific inputs.

Test process

The tests (specifically HTTP tests) required our API to be running on a local server and then the tests could be run from the TestScripts folder by executing the ‘pytest’ command. This was not something we knew from the start of our testing process so our HTTP tests were throwing errors until we realised what the problem was. The tests would run in alphabetic order, of the test filename (pictured below). The order of our test files/test cases did not matter as they were independent of each other. Testing would occur every time we made changes to our API code or implemented a new feature. If the tests did not pass we would make the appropriate changes depending on whether the tests needed to be updated or a bug in our code needed fixing.

```
C:\seng3011-disease-watcher\Phase1\TestScripts>pytest
============================================= test session starts =============================================
platform win32 -- Python 3.8.3, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: C:\seng3011-disease-watcher\Phase1\TestScripts
collected 26 items

outbreak_all_test.py .                                                                                [  3%]
outbreak_disease_test.py ....                                                                         [ 19%]
outbreak_http_test.py ..............                                                                  [ 69%]
outbreak_location_test.py ....                                                                        [ 84%]
outbreak_region_test.py ....                                                                          [100%]
```

Test Results

All tests ultimately passed with zero errors after finalising the implementation of our API.

```
  c:\python38\lib\site-packages\pymongo\srv_resolver.py:72: DeprecationWarning: please use dns.resolver.resolve() instea
d
    results = resolver.query('_mongodb._tcp.' + self.__fqdn, 'SRV',

..\..\..\python38\lib\site-packages\pymongo\srv_resolver.py:57
..\..\..\python38\lib\site-packages\pymongo\srv_resolver.py:57
..\..\..\python38\lib\site-packages\pymongo\srv_resolver.py:57
..\..\..\python38\lib\site-packages\pymongo\srv_resolver.py:57
..\..\..\python38\lib\site-packages\pymongo\srv_resolver.py:57
  c:\python38\lib\site-packages\pymongo\srv_resolver.py:57: DeprecationWarning: please use dns.resolver.resolve() instea
d
    results = resolver.query(self.__fqdn, 'TXT',

-- Docs: https://docs.pytest.org/en/stable/warnings.html
========================================== 26 passed, 15 warnings in 25.36s ==========================================
```

Test method

Our testing scripts were developed with the principle of black-box testing in mind such that the internal logic of our code was irrelevant in comparison to returning correct output. This allowed us to develop the testing cases before we started development allowing us to set concrete requirements for what each method is meant to achieve without interfering with the logic of other methods. This allows us to ensure that any future code written was able to functionally and correctly fulfil its requirements. Black box testing also allowed us to ensure that various methods called in conjunction will not interfere with the internal logic flow and thus insure that any requests made to our API endpoint, regardless of parameters can be correctly processed.

Tools used

All of our testing scripts were developed using Pytest as it can be easily integrated with our API hosted on a Flask server. Additionally the Pytest module allows for easy creation of modular testing cases allowing us to easily test edge cases and other potential erroneous inputs from the user whilst also being more suitable for our agile development approach in comparison to the built-in Python testing module.

Limitations

There were several limitations that we encountered when we were developing our testing scripts. One such limitation was the assumption that data stored and available in our MongoDB database was valid and of consistent format. Additionally, it would have taken the tests too long to check the response articles from each API call due to the number of articles that API calls would return. As a result, we were only able to check the first few articles in each test. In order to be 100% sure of our API responses, it was necessary to manually check the original json file that contained all of the articles that were imported into the database.

Testing of our natural language processor for tagging our data had to be conducted manually as the results could only be verified by human eyes. As a result, it is impossible to verify each of the 8000 articles we tagged and so many articles may have erroneous tags.

Improvements from testing

Aside from fixing the bugs and errors in our code that were discovered from the testing scripts. As were iterated on our backend functionality the testing scripts ensured that the logic of new features did not break the implementation of previous features.