Testing Documentation

Group - 1 Group 2 Group 3 Group 4

Joel Ivory (z5162834)
Hoshang Mehta (z5255679)
Lachlan Sutton (z5308261)
Jiaqi Zhu (z5261703)
William Wan (z5311691)

Table of Contents

Overarching Testing Philosophy

Our team chose not to use full test-driven development (TDD), and instead opted for a more moderate version of TDD, in that our software was largely developed outside of pre-written test cases, and augmented later to work with these test cases. This choice was made mainly due to the relatively short development timespan. While full TDD does generally lead to higher quality and faster development over a large amount of time, there is a considerable overhead to the setup of these tests that would have prevented the team from producing a working solution in time. Additionally, delaying the writing of tests allowed us to change the structure and output of the API to better suit potential users without having to rewrite the testing suite to facilitate this change.

In an attempt to maintain some of the benefits of TDD, our testing was instead written alongside the development of the implementation. Through this moderate TDD, the team was able to ensure the correctness of the API while preventing the halting of development that comes with the writing of tests beforehand.

Testing Overview

As the API consists mainly of only one endpoint, testing is relatively simple, and unit testing will provide most if not all testing information. Testing is run through pytest and pytest-django, a plugin for pytest that provides tools for testing on our chosen web framework, Django.

Each endpoint of our solution has its own unique testing file, which contains the entirety of the testing suite for that endpoint. The files are named according to the format “test_[endpoint].py”. For example, an endpoint /search/only/one/ will have a test file “test_search_only_one.py” which will contain every test that relates directly to /search/only/one/. There are three of these in the current version of the software, being: “test_docs.py”, “test_index.py”, and “test_search.py”.

Test Case Generation

Test cases are chosen through a ‘divide-and-conquer’ method, in which possible outputs are divided into sections of possible output, and then further refined into specific test cases, so that every possibility is considered in the test cases. For example, our /search/ endpoint broken down once is as follows:

Error		Non-Error		
Invalid Method	Invalid Input	0 results	1 result	Many results

This is further broken down in the below section, ‘Chosen Test Cases’.

Chosen Test Cases

As above, the test cases are broken into nested categories that specify the area that each case tests for.

/>Exists:

This test simply tests that the index page is a valid page for HTTP GET requests. This is the only test for the index page as it is not a part of the underlying API.

/docs/>Exists:

This test simply tests that the /docs/ page is a valid page for HTTP GET requests. This is the only test for /docs/ as it is not a part of the underlying API.

/search/>Error>Invalid Method:

In this test, all invalid HTTP methods (all but GET) are sent to the endpoint, and each request must return a ‘405 Method Not Allowed’ error. This test is important as it will prevent confusion on which method to use for this endpoint

/search/>Error>Invalid Input>Date:

In this test, an invalid start date string is given to the /search/ endpoint in a GET request. The API must then return a ‘400 Bad Request’ error. The same test is then performed with an invalid end date, and the API must produce the same result.

/search/>Non-Error>0 results>Simple:

In this test, a simple valid request containing a date range in which there is no data is given to the /search/ endpoint in a GET request. The API must then return a ‘200 OK’ response with an empty “articles” array.

/search/>Non-Error>0 results>Wrong Order:

In this test, a valid request containing a date range in which the start date is after the end date is given to the /search/ endpoint in a GET request. The API must then return a ‘200 OK’ response with an empty “articles” array.

/search/>Non-Error>0 results>Equal Time:

In this test, a valid request containing a date range in which the start date is equal to the end date is given to the /search/ endpoint in a GET request. The API must then return a ‘200 OK’ response with an empty “articles” array.

/search/>Non-Error>0 results>No Location:

In this test, a valid request containing a location that does not exist is given to the /search/ endpoint in a GET request. The API must then return a ‘200 OK’ response with an empty “articles” array.

/search/>Non-Error>0 results>No Key Terms:

In this test, a valid request containing a key term that will not be in any article is given to the /search/ endpoint in a GET request. The API must then return a ‘200 OK’ response with an empty “articles” array.

/search/>Non-Error>1 results>Old Format:

In this test, a valid request containing a date range that will return ‘old format’ articles (Articles using a naming format that is no longer used) is given to the /search/ endpoint in a GET request. The API must then return a ‘200 OK’ response with an “articles” array containing 1 article corresponding to https://www.who.int/emergencies/disease-outbreak-news/item/2003_03_7a-en. This test is necessary to ensure that the scraper is correctly identifying key values, and that the database is correctly being filtered.

/search/>Non-Error>1 results>New Format:

In this test, a valid request containing a date range that will return ‘new format’ articles (Articles using a naming format that is currently being used) is given to the /search/ endpoint in a GET request. The API must then return a ‘200 OK’ response with an “articles” array containing 1 article corresponding to https://www.who.int/emergencies/disease-outbreak-news/item/cholera-benin. This test is necessary to ensure that the scraper is correctly identifying key values, and that the database is correctly being filtered.

/search/>Non-Error>1 results>Different Locations:

In this test, a valid request containing a date range that contains two articles corresponding to different locations is given to the /search/ endpoint in a GET request twice, once for each location.. The API must then return a ‘200 OK’ response with an “articles” array containing 1 article each time, corresponding to the location that was queried. This test is necessary to ensure that the database is correctly being filtered by location.

/search/>Non-Error>Many results:

In this test, a valid request containing a date range that contains many articles, only some of which correspond to a chosen location is given to the /search/ endpoint in a GET request. The API must then return a ‘200 OK’ response with an “articles” array containing **only** the articles that are from this chosen location. This test is necessary to ensure that the database is correctly being filtered by location, and can filter multiple articles over large ranges.

Improvements

The chosen testing philosophy (moderate test-driven development), while time-efficient in the short-term, is not sustainable or time-efficient in the long-term. In the case that development is continued, the testing philosophy should change to test-driven development to ensure correctness during development, to prevent mistakes and potential unnecessary rewrites of large sections of the API.

Additionally, more specific tests could be written in future, as the API structure and output format should be more concrete and understood by the team after further development, allowing more granular tests to be written.

Performance

At the time of writing, 13 out of 14 tests pass. “/search/>Non-Error>1 results>Old Format” does not pass due to a discrepancy in the headline of the article. This indicates a fairly robust API that conforms to the team’s expectations of outputs.