TESTING

Deliverable 4

Team QQ

SENG3011

1 CONTENTS

2 Overview

Testing is an important part of building any kind of software. There are various kinds of tests that can be used throughout development, including unit testing, integration testing, end-to-end testing and volume testing, each of which verifies the software at different levels. Tests are often automated and can be run at times when code should be working, such as prior to building. Another useful aspect of testing is that it makes refactoring code safer, as a failed test can indicate a change in the external behaviour of the program, likely demonstrating a destructive change during the refactor.

A feature of many test suites is the reporting of code coverage. Coverage indicates which lines of code have been run within tests and can provide guidance on which areas might need more thorough tests. While a high coverage percentage is desirable, it is not always practical or necessary to test all lines of code. As such, test writing can be just as programmatically expressive as the working code.

During the development process of the ViraLog platform, major changes to the API and scraper were made to allow us to implement all requirements and bring the most useful features to our users. Due to this high fluidity of the state of the API and significant time constraints, it was not feasible for us to write up to date tests for the scraper, the API and the frontend.

Given more time for this project, we would have written tests for the front end and updated our tests on the backend. This would have ensured that our product was as bug-free as possible and delivered the most consistent and reliable information possible to each user. Despite there being no formal code tests, the team have all individually rigorously used the platform during the development process to ensure we minimised the number of bugs in the application.

2.1 Methodology

2.1.1 Types of Testing Used

To rigorously test the system, many types of tests were considered.

2.1.1.1 Unit Testing

Unit testing is the most fundamental level of software verification, ensuring that each individual component behaves as expected. The simplest example of a unit test would be an assertion; however, assertions are not generally used for this kind of testing as they are executed at runtime. Instead, testing frameworks and integration workflows allow for testing prior to creating a build for production.

2.1.1.2 Integration Testing

As a system becomes more complex, it is important to make sure that software components are being combined properly to achieve the expected output. Even if all unit tests pass, that does not guarantee that the components have been correctly incorporated. As such, integration testing is another important aspect of ensuring a functional system.

2.1.1.3 Volume Testing

Volume testing verifies the rigidity of the system when it is under high load. When a server is receiving a high volume of requests, it is important to ensure the maintenance of properties such as data consistency, response time, error handling, and resource management.

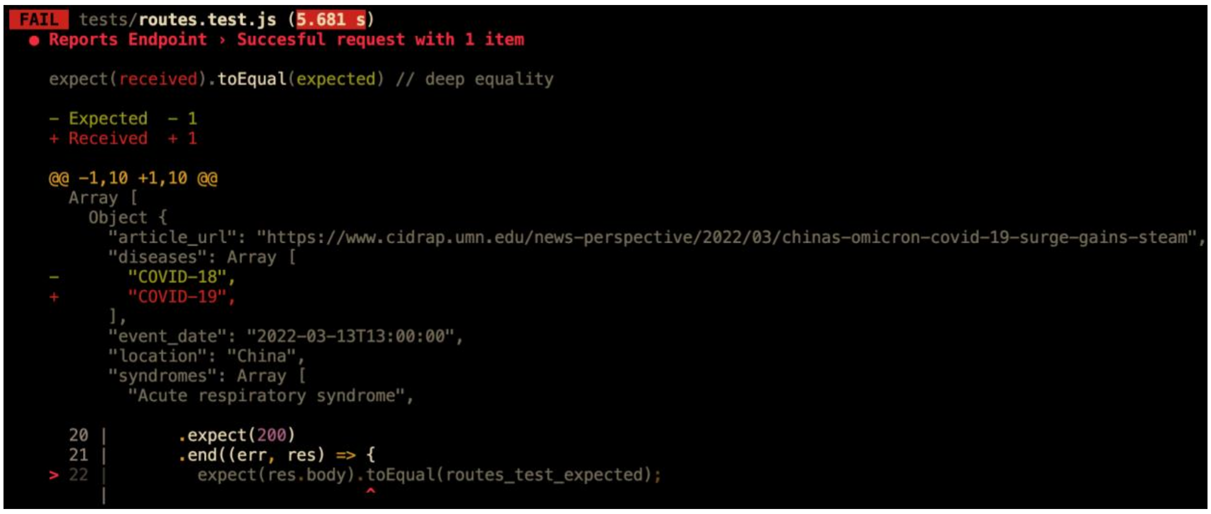
This kind of test is significantly harder to set up than unit tests and integration tests, largely since a high volume of requests must be simulated to accurately imitate a real influx of requests.

Furthermore, volume testing is costly, time consuming, and even if an error does occur it can be difficult to replicate and diagnose. It is also difficult to model a realistic influx of requests.

After considering these factors, and the fact that our API will not be under heavy load, our team did not attempt to create volume tests for our API.

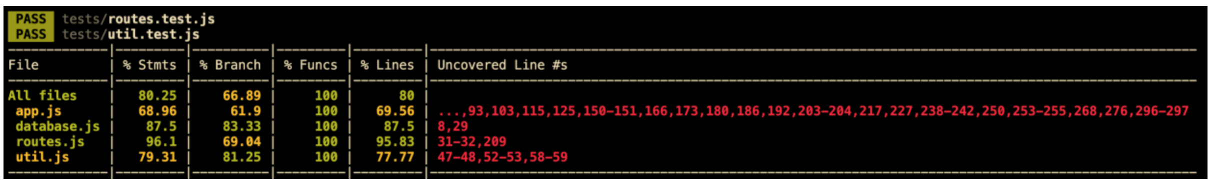
2.1.2 Testing Framework (Jest)

Most languages have frameworks that automate the running of tests. The testing package we are using for this project is ‘Jest’ which is compatible with Node, SuperTest (for HTTP route testing), and even frontend frameworks such as React. Jest provides a useful set of expectation functions, including `toBe`, `toBeCloseTo`, `toEqual`, `toStrictEqual`, `toHaveProperty`, `toMatchSnapshot`, `toThrowError`. If an expectation is not satisfied, the framework provides useful feedback upon what was expected and what the true result.



```
 FAIL  tests/routes.test.js (5.681 s)
  ● Reports Endpoint › Succesful request with 1 item

    expect(received).toEqual(expected) // deep equality

    - Expected  - 1
    + Received  + 1

    @@ -1,10 +1,10 @@
      Array [
        Object {
          "article_url": "https://www.cidrap.umn.edu/news-perspective/2022/03/chinas-omicron-covid-19-surge-gains-steam",
          "diseases": Array [
    -       "COVID-18",
    +       "COVID-19",
          ],
          "event_date": "2022-03-13T13:00:00",
          "location": "China",
          "syndromes": Array [
            "Acute respiratory syndrome",

      20 |         .expect(200)
      21 |         .end((err, res) => {
    > 22 |           expect(res.body).toEqual(routes_test_expected);
         |                            ^
```

Jest also provides coverage information in an easily readable format.



```
PASS  tests/routes.test.js
PASS  tests/util.test.js
------------|---------|----------|---------|---------|---------------------------------------------------------------------------------------------
File        | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
------------|---------|----------|---------|---------|---------------------------------------------------------------------------------------------
All files   |   80.25 |    66.89 |     100 |      80 |
 app.js     |   68.96 |     61.9 |     100 |   69.56 | ...,93,103,115,125,150-151,166,173,180,186,192,203-204,217,227,238-242,250,253-255,268,276,296-297
 database.js|    87.5 |    83.33 |     100 |    87.5 | 8,29
 routes.js  |    96.1 |    69.04 |     100 |   95.83 | 31-32,209
 util.js    |   79.31 |    81.25 |     100 |   77.77 | 47-48,52-53,58-59
------------|---------|----------|---------|---------|---------------------------------------------------------------------------------------------
```

3 Front End Testing

Front end testing can be very challenging due to how a front end application is visually driven, as opposed to being data-driven like an API or web scraper. Being visually driven brings an entirely new set of challenges to the testing process as you must test that all components function correctly on all different devices and screen sizes. Due to how dynamic the site is and how it looks different on different devices, testing that the CSS renders properly and measurements fit the design is near to impossible. The accessibility of the site must also be tested to ensure, for example, that someone using a screen reader can navigate it. Given time constraints and the volume of work that would have been required to fully test the application front end, we were not able to properly write tests with reasonably high coverage.

Testing a front end application can typically be broken down into three stages, unit testing, integration testing and end to end testing. By combining all these techniques, an application can reach quite a high percentage of code coverage. Below are details of how we would have gone about testing the front end of our application.

3.1 Unit Tests

Unit tests are the foundation for testing the front end of an application and are aimed at ensuring small functions work correctly and data has been correctly collected and rendered from the API. Jest is a good tool for this and was developed with React component testing in mind. Jest allows a developer to render a component in the test and check that the inner HTML values it is displaying are correct.

An example of a unit test for our platform would be ensuring that the aliases for a disease on a particular disease page are correct and match those of the database. Once this has been verified for all diseases, we can check other small pieces of information such as symptoms and that the risk analysis values are being displayed properly.

3.2 Integration Tests

Integration tests sit between small unit tests and full-blown end to end tests. Most commonly they involve DOM simulation and interaction with the site. They are usually aimed at ensuring state management is correct and is being handled appropriately. Given enough time, we would have implemented integration tests using the jsdom library. This is a JavaScript library aimed at simulating the DOM without the need for a client-server. It does this by implementing just enough of a subset of a web browser to be useful for testing. More about jsdom can be found here.

An example of an integration test for our platform could be triggering a watch/unwatch of a disease from the dashboard page and ensuring that its corresponding report frequency graph appears/disappears. We could also test that the active diseases tool functions correctly by counting how many reports each disease has had recently and ensuring that only those which fit within the user-defined parameters are displayed and are updated upon the user adjusting said parameters.

3.3 End to End Tests

End to end testing is not as important as unit tests or integration tests for checking the functionality of the front end. If all unit tests and integration tests pass, then the core functionality of the application should be correct, and an end-to-end test is simply checking the functionality of the glue holding each component together. It can be very time consuming to run tests like this and they can often fail due to inconsistent input and output from a user or inconsistent internet connection.

To conduct end-to-end testing, we would use a framework called Mocha. This allows you to run a mock development server and serve the webpage on a localhost URL while interacting with it using code. This lets you simulate clicking buttons on the page, inputting information, and navigating around the site. Mocha aims to bring together the full flow of the site and simulate a real user using the platform.

An example of this kind of test could be a user first begins on the map page, clicks on a map pin, navigates to the corresponding disease, watches it, moves back to the dashboard page and confirms that the disease is in the “watched” section. Mocha would allow us to ensure there is minimal lag between each of these actions, each of the actions are called correctly and they have the correct functionality.

4 Scraper Testing

Completely testing the scraper's functionality is a task that cannot quite be completed. Due to the nature of a news website, the articles that the scraper grabs will be constantly updated and changing. Hence, if we wanted to compare the results collected by the scraper to example test cases or other data, we would need to pull that from the CIDRAP website too, however, we would be unable to test whether that scraper is working correctly.

Instead, we must focus on the more important task of ensuring the scraper processes articles correctly, instead of whether it collects them from CIDRAP correctly. It should also be noted that due to time constraints, we did not have the chance to write tests for the Geocoding API call portion of the scraper. If we were to test this, however, we would do so by creating a sample article with a location, pass it through the scraper and ensure the latitude and longitude values returned are correct. This ensures consistency between Geocoding API calls and when run periodically ensures there are no issues collecting data from this API.

4.1 Unit Tests

The unit tests written for this scraper are focused on ensuring the `` `findReports` `` function can pull all reports from each article successfully. To do this, we had to set up a `` `testDiseases` `` object, which is used to map disease aliases to disease ids, thus standardising how we represent our diseases in the database. If, for example, this diseases object was empty when passed into `` `findReports` ``, we would never find any reports in any articles.

The testing structure for the scraper is to create a sample article with each required field in it, then an object with each expected report in it, indexed by their location. This makes it easy to then loop through the list of reports returned by `` `findReports` `` and check that each one matches the expected report. If a report matches its expected value then we delete it from the expected report object to ensure we have not received duplicate reports, then to ensure that the expected report object is empty once the loop has finished executing to ensure every report was found.

There is no error checking functionality within the `` `findReports` `` function, as the only place it is called is from within the `` `processArticles` `` function and that contains error checking for the article object.

4.2 Integration Tests

The integration tests for the scraper focus more on how the `processArticle` function functions and ensures it can pass the correct parameters to the `findReports` function. The test cases are very similar to those used in the unit tests, with the difference being how they are used. The integration tests can be broken down into two targets; ensuring that the `processArticle` function will correctly handle invalid articles and ensuring that the `processArticle` function correctly collects disease alias data from the database and translates it into a format the `findReports` function can find.

To test the error checking functionality of the `processArticle` function, we attempt to pass in an article object which is missing various parameters. When the `processArticle` function receives an object without all the required parameters, an exception is thrown. In the tests, we expect this exception and fail the test if it is not thrown. Jest has some issues with the built-in exception expectation testing, in that to the `expect` function, you must pass in a function, not the result of a function. Due to this increased level of abstraction, sometimes exceptions are not recognised. To work around this in our error checking tests, we found a method that acts almost as a proof by contradiction. Within a try block, we pass in the bad article to `processArticle` and after that we have a Jest `expect` where we expect true to be false. If calling `processArticle` does not raise an exception, then we continue in the `try` section and attempt to expect that true is equal to false. As `processArticle` successfully raises an exception, we can continue to the `catch` block and ensure that the error message of said exception is correct.

These integration tests expand upon the unit tests by ensuring the article processor can correctly collect disease alias information from the database and parse it into a digestible format. This is all done within the `processArticle` function, with the resulting disease information being passed into the `findReport` function. To test this, we create a new database connection at the beginning of each test, then pass that into the `processArticle` function. After that, the process for checking results is the same as the unit tests.

5 API Server Testing

Due to the simple nature of the API server, it was possible to create a comprehensive test suite to thoroughly validate the function of its components. The two kinds of tests used were unit tests for utility functions and integration tests of routes to ensure that the system returns the expected results. Each of these sets of tests used the Jest package for safe and parallel testing with useful expectation functions.

Given how much the API was being changed during the development of the platform, we did not have the time or resources to write consistent and high coverage tests of each new feature. We did however write comprehensive tests of the core features of the API, with most new features simply extending the core functionality. We have detailed our testing strategies below.

5.1 Unit Tests

Unit testing is the most fundamental level of software verification, ensuring that each individual component behaves as expected. There were several unit tests written for the API which involved testing utility functions. These utility functions included ensuring dates passed in were of the correct format (‘`yyyy-MM-ddTHH:mm:ss`’), all required input parameters existed in the query string and were not null, and the type of all parameters were strings (as opposed to being null, boolean or integers for example).

The methodology behind writing these unit tests included looking at valid inputs and trying to slightly modify them to replicate possible human behaviour where the inputs would become invalid. This was especially true for testing invalid dates, as common errors may include inputting an invalid month, day, hour, minute or second field, or forgetting to add the 'T' at the middle of the date string. Adding a variety of these tests proved useful as an error existed in the regex string where it would accept a 'HH' (hour) value of any number below 30, instead of capping it at 24. This may seem inconsequential; however it could've caused issues regarding the incorrect returning of articles/reports between two given dates.

5.2 HTTP TESTS

HTTP tests were implemented using the framework provided by Jest and in conjunction with the 'SuperTest' package. The aim for the HTTP tests was to ensure that the API provides the expected responses when an endpoint is called using a variety of valid and invalid parameters. By checking that the appropriate status codes, responses, and/or errors are being served, the quality and functionality of the API can be confirmed. These tests also provide another point of safety for continuous development, allowing unintended impacts on behaviour to be immediately detected.

These tests could be considered unit tests as each test evaluates a single scenario of API usage. They could also be considered integration tests as they cover the whole API codebase, as opposed to the unit tests of the utility functions which only cover single functions. We implemented a range of both unit and integration HTTP tests. Regardless of their definition, these tests are black-box and only the external interface that is provided to the user is observed.

The SuperTest package for Node, which is an extension of the JavaScript 'SuperAgent' HTTP client, provides a '`superagent`' to handle the binding and management of the application. SuperTest can be used as a standalone testing library however we used it with Jest, as it is the framework being used to test other components of the project.

The data used for the test was initially manually developed dummy data, however as the functionality of the API evolved, real results that were manually verified to be correct were added as test cases. One limitation of HTTP testing includes a lack of existing tests to replicate '500 Internal Server' errors, as if we were aware of a '500' error it would become a priority to be fixed. Another limitation of testing our API is as time moves forward, the number of possible outputs our API can provide will continue to increase because new articles are constantly being scraped into the database. Therefore, the output our API produces when processing recently scraped articles/reports will remain untested unless new tests are consistently written.