TESTING

Deliverable 2

Team QQ
SENG3011

1 CONTENTS

2 OVERVIEW

As stated in our deliverable 1 Design Details report, 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, 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.

One common practice is to implement a new test each time a bug is found. This ensures that the bug can be immediately detected if it were to re-emerge.

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.

Testing was very important to us throughout the process of developing our API and scraper. We experienced the advantages outlined above; we were able to utilise the tests we had created to detect bugs and it allowed us to safely refactor our code.

2.1 TESTING PHILOSOPHY

While developing our scraper and API, our team utilised the test-driven development methodology. This meant that during the planning stage for our project, once we had decided on the requirements for our API, we developed an extensive set of test cases to match these requirements. As we were developing, we would repeatedly run the tests against our software which allowed us to keep track of how much progress we had made.

Another advantage of using test-driven development was that it gave each of us a better understanding of the project requirements, we created tests for all possible cases that we could think of, which forced us to learn about what all the endpoints are and what error codes should be returned under certain conditions. Test-driven development allowed us to fully think through our requirements before jumping into writing our code. Looking back, we believe that this approach saved us a lot of time in the long run, test-driven development presented us with the opportunity to fully agree upon our system requirements so that we were all on the same page, rather than having disputes later once code had already been written.

The tests were created slightly after the Swagger requirements had been completed. There were a few times when, as we started writing the tests, we realised that the API (or scraper) should really behave differently than what was outlined in the Swagger, so we would update the Swagger.

2.2 Methodology

2.2.1 Types of Testing Used

In order to rigorously test the system, many types of tests were considered.

2.2.1.1 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. As a part of the test-driven development approach taken by our team, unit tests were vital for us, and they were definitely one of the most common forms of tests that we used (see later in the report for some examples).

2.2.1.2 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. Our team made common use of integration testing through our tests of the API endpoints for all the routes, as well as our testing for the scraper. These components utilised multiple different functions, so we used integration tests to test them.

2.2.1.3 Volume Testing

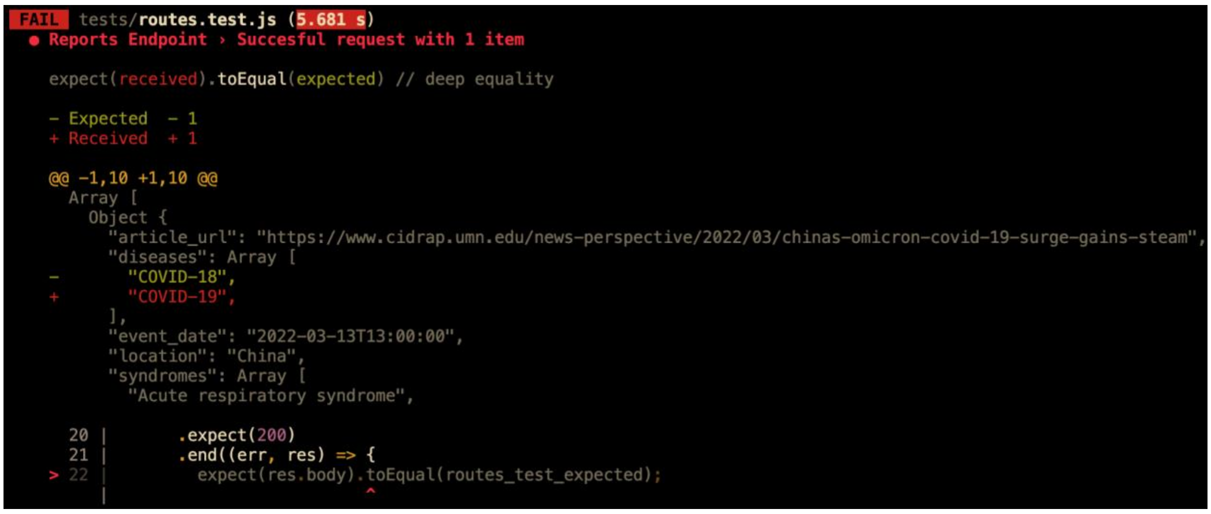
Volume testing verifies the rigidity of the system when it is under high load. When a server is receiving a high volume of requests, it is important to ensure the maintenance of properties such as data consistency, response time, error handling, and resource management.

This kind of test is significantly harder to set up than unit tests and integration tests, largely due to the fact that 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.

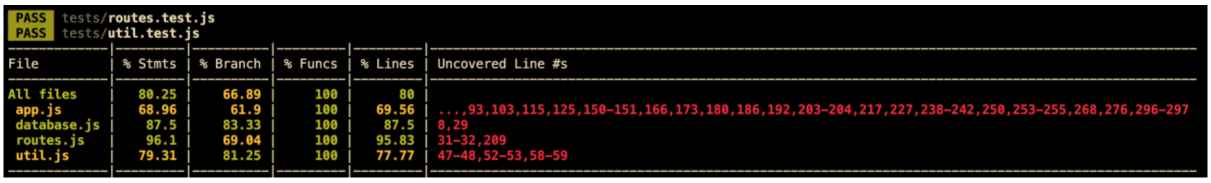
2.2.2 Testing Framework (Jest)

Most languages have frameworks which automates 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, which will be utilised in the next phase of the project. 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 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. This is done through the unit tests and integration tests described below.

There are two main functions involved in the process of digesting an article into reports, these are `processArticle` and `findReports`. The `processArticle` function takes a database connection and an article as its parameters, the `findReports` function takes an article and a list of diseases to map aliases to their `disease_id`.

Each article has a specific structure it must adhere to, which is as follows:

```
{
  article_url:String,
  headline:String,
  date_of_publication:String,
  author:String,
  main_text:String,
  category:String,
}
```

The `processArticle` function handles any inconsistencies between the structure of the articles collected and what their structure should be.

3.1 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.

Some notable unit tests include testing that `findReports` works when the main text parameter contains no reports, but there is one in the headline of the article. The `findReports` function searches for any disease alias within each sentence, delimited by full stops, and when receiving the article it joins the main_text and headline strings together. Additionally, we test for when there are no reports contained in an article and in this case return an empty list of articles.

3.1.1 Sample Unit Tests

Example 1: Testing multiple reports can be found within one article.

Here, the process of testing can be seen clearly where we set out a test article, create the expected results then expect the returned results. Note the for loop at the end removing items from the target object to avoid duplicates and then verifying that the target object is empty by the end.

```
test("test_multiple_reports", async () => {
    let article = {
        headline: "Something around the world",
        date_of_publication: "Mar 15, 2022",
        author: "Alex",
        main_text: "We found a case of legionnaire in Thailand. Also someone may have 
hiv/aids in Australia",
        article_url: "www.a_standard_disease_site.com",
        category: "news"
    }

    let targets = {
        Thailand: {
            article_url: "www.a_standard_disease_site.com",
            disease_id: "legionnaire",
            event_date: "Mar 15, 2022",
            location: "Thailand"
        },
        Australia: {
            article_url: "www.a_standard_disease_site.com",
            disease_id: "hiv/aids",
            event_date: "Mar 15, 2022",
            location: "Australia"
        }
    }
    let processed = await findReports(article, testDiseases);
    expect(processed.length).toBe(2);
    for (let i = 0; i < processed.length; i++) {
        
expect(JSON.stringify(targets[processed[i].location])).toBe(JSON.stringify(processed[i]))
        delete targets[processed[i].location]
    }
    expect(isEmpty(targets)).toBe(true)
})
```

Example 2: Testing using aliases.

This test is targeted at determining whether or not the `findReports` function is capable of coverting disease aliases to their disease id. You can see that a case of "legionella pneumonia" was found in Thailand, and due to this being an alias for the "legionnaire" disease, we receive back that the `disease_id` is "legionnaire".

```
test("test_using_aliases", async () => {
    let article = {
        headline: "Something around the world",
        date_of_publication: "Mar 15, 2022",
        author: "Alex",
        main_text: "We found a case of legionella pneumonia in Thailand.",
        article_url: "www.a_standard_disease_site.com",
        category: "news"
    }
    let targetReport = {
        article_url: "www.a_standard_disease_site.com",
        disease_id: "legionnaire",
        event_date: "Mar 15, 2022",
        location: "Thailand"
    }

    let processed = await findReports(article, testDiseases);
    expect(processed.length).toBe(1)
    expect(JSON.stringify(targetReport)).toBe(JSON.stringify(processed[0]))
})
```

3.2 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 which 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. Refer to the below example to see exactly how this exception catching works.

```
    try {
        let processed = await processArticle(conn, article);
        expect(true).toBe(false);
    } catch (e) {
        expect(e).toBe("Missing field(s) from parameter 'article'");
    }
```

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. In order 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. It should be noted that to reflect the difference between scraper unit tests and integration tests, the final test of each section is the same but for how the data regarding disease aliases is collected.

4 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.

4.1 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.

4.1.1 Example Tests

In each provided example, Jest provides an `expect()` function where you can input a value as an argument, call the `toBe()` function and input what the expected result should be as an argument. If both argument values match, the test will pass.

Example 1: Testing a correctly formatted date is recognised to be correct. This test ensures that 'true' is returned as the date is valid.

```
test("test_correct_time_format_1", async () => {
    let timestamp = "2022-03-13T13:00:00";
    expect(util.timeFormatCorrect(timestamp)).toBe(true);
});
```

Example 2: Testing an incorrectly formatted date is recognised to be incorrect (as the 'day' field is '1' not '01'). This test ensures that 'false' is returned as the date is invalid.

```
test("test_invalid_time_format_3", async () => {
    let timestamp = "2022-03-1T13:00:00";
    expect(util.timeFormatCorrect(timestamp)).toBe(false);
});
```

Example 3: Testing that a parameter which is Null (doesn't exist) is correctly identified to be missing. This test ensures that the '`period_of_interest_start`' parameter is returned as being incorrectly formatted.

```
test("test_findNull_function_3", async() => {
    let object = {
        "period_of_interest_end": "2022-03-18T19:00:00",
        "key_terms": "ebola,COVID-19",
        "location": "Bhutan"
    };
    let keys = ["period_of_interest_start", "period_of_interest_end",
"key_terms", "location"];

    expect(util.findNull(object, keys)).toBe("period_of_interest_start");
});
```

Example 4: Testing that the type of each parameter should be a single string. In this case the 'key_terms' parameter is a list of strings rather than a comma-separated list of diseases within a single string. This test ensures that the 'key_terms' parameter is returned as being incorrectly formatted.

```
test("test_findNotString_function_2", async() => {
    let object = {
        "period_of_interest_start": "2022-03-13T19:00:00",
        "period_of_interest_end": "2022-03-18T19:00:00",
        "key_terms": ["ebola", "COVID-19"],
        "location": "Bhutan"
    };
    let keys = ["period_of_interest_start", "period_of_interest_end",
"key_terms", "location"];

    expect(util.findNotString(object, keys)).toBe("key_terms");
});
```

4.2 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.

4.2.1 Example Tests

In the following example, SuperTest provides the `request` function (second line) and the methods on the resulting object. Two methods of note are the `.end` and `done` methods, which provides a callback to be run after the request has been sent. This allows the content of the response to be checked. `done` is called to tear down the internal state of the test for the superagent. The purpose of each function can be seen as a comment within the code snippet below:

Example 1: Testing that the API correctly returns a value based on the input parameters provided. Note that this test will always pass as the two periods of interest are one second apart, so scraping new articles/reports will not influence the output of this API between the two provided periods of interest.

```
test("Succesful request with 1 item", (done) => {
    request(app)
      .get("/reports")  // the route being used
      .set("Content-Type", "application/json")  // content type HTTP header
      .query({  // the input parameters and their respective values
        period_of_interest_start: "2022-03-13T13:00:00",
        period_of_interest_end: "2022-03-13T13:00:01",
        key_terms: "COVID-19",
        location: "China",
      })
      .expect(200)  // the expected HTTP response type
      .end((err, res) => {
        expect(res.body);
        expect(res.body).toEqual(routes_test_expected);  // testing that
'res.body' is equal to 'routes_test_expected'
        done();
      });

  });
```

See next page…

The value of the variable `routes_test_expected` can be seen below, and is what the output when calling the /reports route should look like when using the API:

```
const routes_test_expected = [
  {
    "diseases": [
      "COVID-19"
    ],
    "event_date": "2022-03-13T13:00:00",
    "location": "China",
    "syndromes": [
      "Acute respiratory syndrome",
      "fever",
      "cough",
      "fatigue",
      "shortness of breath",
      "vomiting",
      "loss of taste",
      "loss of smell"
    ],
    "url": "https://www.cidrap.umn.edu/news-perspective/2022/03/chinas-omicron-
covid-19-surge-gains-steam",
  }
];
```

Example 2: Testing the API correctly detects that an incorrectly formatted date was passed in, raises a **400** error and the correct response message is returned.

```
test("Date in wrong format", (done) => {
    request(app)
      .get("/reports")
      .set("Content-Type", "application/json")
      .query({
        period_of_interest_start: "2020-03-13T12:21:21",
        period_of_interest_end: "2022-03-13X17:21:21",
        key_terms: "COVID-19",
        location: "China",
      })
      .expect(400)
      .end((err, res) => {
        expect(res.body["message"]).toEqual("Invalid timestamp for
'period_of_interest_end', must be in format 'yyyy-MM-ddTHH:mm:ss'");
        done();
      });
  });
```