SENG3011 Testing Documentation
Bobby Tables

Testing Philosophy **2**
‘Begin with the end’ 2
‘Deliver value early’ 2
‘Fail first’ 2
‘Automate everything’ 3
Testing Types **3**
Test cases **3**
Testing Environments and Tools **4**
Testing Performance **4**
Testing Limitation & Potential Improvement **4**
Scraper Tests 4
Hardcoded Authentication 5
Improved Route Tests 5
Appendix **6**
Appendix A: route test cases 6

Testing Philosophy

Our testing philosophy uses principles and concepts from different software methodologies but it can be simplified into the following:

1. ‘Begin with the end’
2. ‘Deliver value early’
3. ‘Fail first’
4. ‘Automate everything’

‘Begin with the end’

This principle embodies the **Test-Driven Development (TDD) methodology** and the concept of writing tests before writing a line of code. This helps us to be more intentional with how we approach our solution and ensure we are thoroughly meeting our **customer’s requirements**. Adopting a TDD approach maintains our focus on *what* our solution needs to be able to do, without losing momentum in trying to figure out *how.*

To uphold this principle when implementing our **data scraper**, we first studied the end result - that being the information we wanted to extract from the website. By **analysing the sample reports** provided to us by the client and understanding what was required, we were able to take care in implementing our scraper to meet our **client’s needs**.

Before beginning our API implementation, we first focused on formulating our **API documentation**. We explored every possible route of our API in extensive detail to address any errors that may arise. This way, we were able to equip ourselves by preparing **informative response messages and suggestions for resolution.**

‘Deliver value early’

Since we are taking an **agile** approach to this project, our testing philosophy draws upon the agile principle of **executing quick iterations to deliver value early**. In doing so, we are able to prioritise **incremental progress** rather than building a system to perfection that ultimately may not be aligned with what the customer wants. In delivering value early, we are able to be **flexible** and adapt to **changing requirements** and any **unexpected setbacks**. To achieve this from a testing perspective, we set out to **deploy our application** to a cloud-based server before implementing our API. This ensured that our API was working **end to end** even before we had started developing it. We sought to deliver value early in many aspects of this deliverable such as **deploying our documentation with stubs** covering every route before fleshing it out, writing stubs for all our **test cases** before implementing them, and developing a **working data scraper** before fine-tuning it for accuracy. With this approach, we were able to **test early and get feedback** from each other and our mentor before proceeding, allowing us to **resolve any issues** and **pivot** when needed.

‘Fail first’

As engineers, we know that it is important to **break code** to **produce a resilient, fully-functional system**. Therefore, one of our philosophies is to embrace when our tests and builds fail so that we can locate bugs that we may not be aware of. When writing tests initially, we designed them in a way that our code **fails the first time**. This helped us to be certain that the test we were writing was **successfully**

targeting the feature that needed testing. This practice prevented us from writing **non-functional and redundant tests**, thus ensuring our test suite was valuable and extensive.

'Automate everything'

The word 'everything' is an exaggeration but this philosophy encourages us to **automate our workflow** wherever we can. Because this project is extremely restricted by time, we need to be sure we are focusing on what is absolutely necessary without wasting precious time and resources on what is not. To do this, we have to rely on automation. Rather than running tests and deploying our app manually, we have implemented **Github Actions** to do this work for us. **Pushing to our remote repository** triggers our **tests** to run and **pushing to our 'main' branch** runs our **deployment** to Heroku automatically. As a result, we can receive feedback quickly on whether our code is **building successfully** and **immediately take action**, rather than waiting for someone to manually review it. Therefore, this testing philosophy enables us to **value our time** and execute our tasks **efficiently** and **intentionally**.

Although we are not perfect in upholding our testing philosophies due to time restrictions, we can use them as a guideline as we progress through the different deliverables.

Testing Types

Black box testing of each route was undertaken in the design of the testing of the API. This ensures all routes **function as expected from the end user perspective**. The overall process undertaken is outlined below.

1. Write tests for each route
2. Implement routes to pass their tests
3. Run coverage testing on routes and update as necessary

This process is done to **ensure functionality of the API while maximising the efficiency at which each route can be implemented**, which is necessary due to the **time constraints** imposed on the project. Additionally, coverage testing of functions reduces the likelihood of an unexpected edge case disrupting the functionality of the backend, thus lowering the chance of the API breaking for the end user.

Test cases

The table shown in Appendix A dictates the current list of routes being tested and their corresponding test cases.

These tests were constructed by **analysing the input parameters** for each route and considering what possible malformed requests can be made. From this, each one was **assigned an appropriate HTTP status code and error message** detailing the problem within the request. This approach of outlining the expected errors beforehand ensures the implementation is able to handle foreseeable errors arising from bad requests.

Testing Environments and Tools

As mentioned in the design details report, the testing is being done through the use of Pytest. This is primarily due to **familiarity, coverage testing functionality and wide variety of documentation** and support available for the library. Pytest-cov will be used to test the coverage of the server due to its compatibility with pytest.

Since the backend and the scraper is being developed asynchronously, it can be difficult to define expected outputs of the routes as the **reports generated by the scraper will differ in each iteration**. To avoid this problem, a **separate testing database has been established** to mimic the behavior of the actual database except without the presence of a scraper to update articles. Instead the database is filled with sample articles for the tests to use. As such, the testing of the backend is done in a separate environment with its own sample database.

Testing Performance

The performance of the testing environment varies depending on the user. This is due to the fact that connecting to the **database and querying for results can take a bit of time**. Since there are multiple tests to run on multiple routes, this can take quite some time before the entire test suite has finished running. While a majority of the time, the tests typically **take about 10 seconds** to run, there have been instances where the tests can take **upwards of a minute**. The cause in this discrepancy has not yet been identified but aside from this setback, the **tests are all passing and performing smoothly**.

The coverage of the server is currently at 100% for helper files and 89% for the main file. The lower percentage is due to the fact that the main file also contains some code to display the auto-generated Swagger documentation which is manually checked.

API_SourceCode/API/helpers.py	54	0	0	100%
API_SourceCode/API/main.py	74	8	0	89%

Testing Limitation & Potential Improvement

The table below outlines **limitations** of our testing system that we have currently identified and also explored **potential future improvements** that we can consider.

Scraper Tests

At the moment, there are no tests in place to determine the **efficiency** of the scraper. There are also no tests covering the **accuracy** and **correctness** of the data extracted by the **scraper**. This was due to the fact that we are still finetuning it for accuracy, but despite that, it is important to test as there are many unexpected **web scraping challenges** that may arise. Some examples are:

- If the **website structure changes** and our API no longer scrapes data correctly, yet is still storing things in our database.
- If our scraper malfunctions by only scanning some of the data, resulting in an **incomplete report**.
- If the **wrong data is fetched** and is being stored under the wrong heading or classification.

Thus, it is important that we have tests in place to cover this. A potential solution could be running a **scraper accuracy test** periodically throughout the day. This test would run the current scraper across an article that was previously scanned and stored in the database to **investigate inconsistencies**. We could also compile a list of articles and check the scraper's results against this to monitor **correctness**.

Hardcoded Authentication

To run our testing database, the developer has to **manually hardcode** a string for authentication. This presents multiple **security issues** as this could **accidentally leak** if they forget to remove it from the commit. Furthermore, it is **tedious** to have to find the file and edit it. Some solutions for this could be to have the developer pass their credentials as **command line arguments**, thus removing the need to store it in the code, or utilising **Github Secrets** and storing it in there to speed up our workflow.

Improved Route Tests

There are no checks to automatically see if the **documentation schema** corresponds to the actual format outputted by the route. Currently, our route testing is limited to checking if the returned **response code is correct**. To improve our **route testing coverage**, we could write tests that extract the currently hosted documentation's example values and compare results.

Appendix

Appendix A: route test cases

Route	Test functions	Cases (Status code)
/article	test_article	• Missing input parameters (400) • Missing default parameters (200) • Invalid offset (422) • Invalid limit (422) • Invalid period of interest format (400) • Invalid date range (400) • Invalid version header (422) • Valid parameters (200)
/article/{articleId}/content	test_article_content	• Valid articleId (200) • Invalid articleId (404) • Valid version header (200) • Invalid version header (422)
/article/{articleId}/response	test_article_response	• Valid articleId (200) • Invalid articleId (404) • Valid version header (200) • Invalid version header (400)
/article/{articleId}/assessment	test_article_assessment	• Valid articleId (200) • Invalid articleId (404) • Valid version header (200) • Invalid version header (400)
/article/{articleId}/source	test_article_source	• Valid articleId (200) • Invalid articleId (404) • Valid version header (200) • Invalid version header (400)
/article/{articleId}/advice	test_article_advice	• Valid articleId (200) • Invalid articleId (404) • Valid version header (200) • Invalid version header (400)
/article/{articleId}/response	test_article_response	• Valid articleId (200) • Invalid articleId (404) • Valid version header (200) • Invalid version header (400)