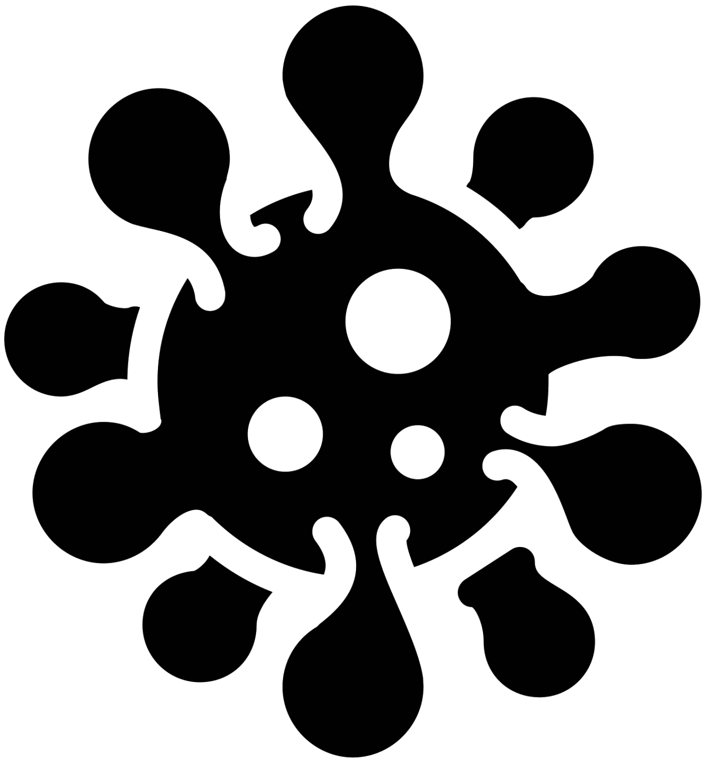
Testing Report



Team Viral

Aimen Hamed - z5310143
Henry Ho - z5312521
Leila Barry - z5206290
Neelam Samachar - z5310063
Shannen Jakosalem - z5164261

Mentor: Nikhil Ahuja
Course: SENG3011

Table of Contents

Introduction

Testing is equally important as code implementation, and our team values an effective and efficient testing strategy in order to validate our APIs correctness.

Within the TestScripts directory of our GitHub repositories we provide:

- Bash Test Script
- Bash Coverage Script
- Postman collections

The scripts within this directory automate and utilise the Jest framework within our application source code, so that we have two entry points to test from. The scripts simply provide an interface and indicator for whether the application currently works or doesn't. For the further explicit failing test cases, they must be run within the application source code as a part of development to get the complete output. The test data which feeds into our testing framework is declared within the application source code under *src/utils/testData.ts*. In the TestScripts README file, a complete documentation is included which describes how to utilise the test scripts.

Postman collections are also found within the TestScripts directory which can be used for manual route testing locally for developers. We have predefined these prior to development, as a means of validation for developers to know what inputs they should take and their expected responses (aligned with the definitions on Swagger).

Test Driven Development

A philosophy that is essential to the development of our backend application is Test-driven development or **TDD**. TDD is a development process that requires a developer to write the test cases based on the requirements before any functional code is written. It is an iterative process that goes back and forth between testing and coding to ensure the code being written is **testable** and **correct** in terms of the business logic.

We have decided to use TDD since writing testable code is valuable to our team as tests are currently the central indicator of the correctness of the code we are writing. As a result, all members aim to implement features via TDD, to avoid painful debugging and promote efficient development.

Types of Testing

To ensure that our software testing is thorough we are using a number of different tests in our project:

1. **Unit tests** - addressing each of the smallest possible units of behaviour to ensure they behave as expected. This includes testing each possible route for expected output, and any edge cases to ensure each case is handled correctly.
2. **Manual route tests** - addressing how the entire application works and ensuring it functions as expected by hitting each route. Currently, these are manually done via Postman, with the Postman collections in the TestScripts folder.
3. **Stylistic tests** - it is important that our coding style and syntax remain consistent within our group to ensure our code is easy to follow. So, we will be making use of stylistic tests that will check our codebase for any inconsistencies or syntax errors.

We will be measuring the success of our tests based on a number of factors:

1. **Line coverage** - to ensure our tests are comprehensive and cover as many possible pathways through the code as possible, we will be measuring the line coverage of our tests. The line coverage refers to the percentage of lines that are executed throughout the tests. We will aim to have greater than 80% of lines executed in tests.
2. **Pass/fail** - to ensure our application behaves as expected, each test will have either a pass or fail result. If a test passes, the code covered by that test behaves as intended in the test case. If a test fails, the code does not behave as intended in that test case and needs to be fixed and re-tested.

For our application to behave reliably as intended, our tests should have both a high percentage of line coverage and should all pass.

Testing Tools and Technologies

We are using an automated testing pipeline to automate the process of reviewing and validating our software. To do this, we require a testing framework and a tool to determine our test coverage. For this, Jest provides the functionality for both automated testing and test coverage – essentially, two tools in one, which removes the learning curve we would have experienced trying to integrate and learn how to use two separate tools. Additionally, Jest is closely integrated with the React testing library, providing support for the CI/CD pipelines we are using, and is revered as a reliable framework for other developers. A unique drawback to using Jest includes its learning curve. However, we were going to face a learning curve regardless of the tool we chose to use and so, this was not a serious issue for us. Thus, Jest was a clear choice for the implementation of automated testing in our CI/CD pipeline, proving to be the correct choice as it has been easy to learn and apply to our test suite.

Other frameworks we considered using were:

1. **Mocha JS** – a JavaScript test framework running on Node.js for asynchronous testing.
2. **Istanbul** – a tool to measure test coverage.

We considered using Mocha JS because it integrates well with Node.js, as well as being a popular choice amongst other developers, with it being compatible with most browsers. The downsides to Mocha JS were that it runs all tests in the same process, which means they share memory and are thus, not isolated from each other. This can cause significant conflicts between tests, further exacerbating Mocha JS' more error-prone nature. Mocha JS also has no code coverage output and so, it would have required us to use another tool, such as Istanbul simultaneously, to ensure our test suite was comprehensive.

Our developers run tests regularly throughout the development process. Typically, this includes after any changes are made to the code prior to any commits and/or pushes, during debugging, and automatically when a pull request is made prior to merging.

To implement these tests when certain actions, such as pull requests, are performed on our GitHub repository, we have decided to use GitHub Actions. GitHub Actions allows for workflow automation including triggering tests on certain actions such as push and pull requests. We chose to use GitHub Actions over alternatives such as CircleCI as we have team members with previous experience using it, as well as the added benefit in the ability to support Node.js and make use of live logs.

We will be using GitHub Actions to test our code upon each pull request to ensure that our application is behaving as intended and to allow our developers a quick insight into the functionality of our code. This minimises the time spent ensuring the correct behaviour of the code, so that we can quickly find and fix bugs when they arise with minimal cognitive overhead due to the information provided by our test suite.

Our Test Cases

The following table contains an outline of all of our test cases. Some of these have not yet been implemented due to the status of our application and will be implemented once their corresponding features are functional. Additionally, more test cases are likely to arise as we begin implementing more of our features and so, our testing suite will be regularly updated throughout the remainder of the project.

Table of current test cases and outline test cases for next deliverables.

Unit Tests and Manual Route Tests		**Method: Automated: Jest, Manual: Postman**	
Route	**Test case**	**Expected behaviour**	**Pass/fail**
articles/dump	No articles in the database.	Throw a HTTP 500 internal server error.	Pass.
	Articles in database.	Resolve and return a list of all articles.	Pass.
reports/dump	No reports in the database.	Throw a HTTP 500 internal server error.	Pass.
	Reports in database.	Resolve and return a list of all reports.	Pass.
article/:articleId	Article does not exist.	Throw a HTTP 404 resource not found error.	Pass.
	Article does exist.	Resolve and return the expected article.	Pass.
report/:reportId	Report does not exist.	Throw a HTTP 404 resource not found error.	Pass.
	Report does exist.	Resolve and return the expected report.	Pass.
search	Start time is after end time.	Throw a HTTP 400 bad request error.	Pass
	Failed to retrieve reports	Throw a HTTP 500 internal server error.	Pass
	List of articles matching criteria	Resolve and return a list of all articles or empty which match criteria.	Pass
users/register	Email already in use.	Throw a HTTP 400 bad request error with log explaining the email is already in use.	D3
	Missing credentials.	Throw a HTTP 400 bad request error.	D3

	Valid credentials.	Register a new user.	D3
users/login	Non-existent email.	Throw a HTTP 400 bad request error with log explaining there is no user with this email.	D3
	Incorrect password.	Throw a HTTP 400 request error with log explaining password is incorrect.	D3
	Correct email and password.	Log in the user and generate a token.	D3
users/logout	Log out user.	Log out the user and dismiss the token.	D3
users/:userId	User does not exist.	Throw a HTTP 404 resource not found error.	D3
	User does exist.	Resolve and return the expected user.	D3
users/bookmark-article	Non-existent user.	Throw a HTTP 400 bad request error with log explaining the user does not exist.	D3
	Non-existent article.	Throw a HTTP 400 bad request error with log explaining the article does not exist.	D3
	Article already bookmarked by user.	Throw a HTTP 400 bad request error with log explaining the article is already bookmarked.	D3
	Valid credentials and not already bookmarked.	Add the article to the users bookmarked articles list.	D3
users/dashboard	Non-existent user.	Throw a HTTP 400 bad request error with log explaining the user does not exist.	D3
	Non-existent dashboard.	Throw a HTTP 400 bad request error with log explaining the dashboard does not exist.	D3
	Dashboard already bookmarked.	Throw a HTTP 400 bad request error with log explaining the dashboard is already bookmarked.	D3
	Valid credentials and	Add the dashboard to the users	D3

	not already bookmarked.	bookmarked dashboards list.	
Stylistic tests		**Method: Lint**	
Our developers run lint and make the required adjustments prior to pushing code. Lint is also run through our automated GitHub Actions pipeline and is required to pass prior to merging a branch with master.			
Total Line Coverage:		83.05% (98/118)	

Product Performance Against Our Testing Suite

With our strategy of TDD and extensive test suite, our code passes with 100% of the test cases and 100% lint standard. This is a great achievement, although it is expected as our CI pipelines won't allow any code with failing tests or lint to be pushed into the deployed branch (Master). From this we have validated that our testing suite and strategy are effective and have proven that the correctness of our product is aligned with our requirements.

We justify the correctness of our output from our API routes via manual testing on Postman. We have set up Postman collections in our TestScripts directory so that developers can manually hit the API endpoints with preset inputs and validate their code. Prior to all code being written, this collection was made so that if the expected output isn't achieved we can validate that our code has bugs. This further follows our TDD approach, and has proven to be effective as our endpoints produce the expected outputs which we have defined in both our Swagger and the Postman collections.

The trailing component of our product's performance that isn't at the standard that the team expected is our overall code coverage. As a result of TDD, it is expected that our code coverage should be between 90-100%. However, we acknowledge that while our intended goal is to have all development done using TDD, it is not the reality with a strict deadline. From this our current code coverage comes at:

Produced via coverage script found in PHASE_1/TestScripts

```
======================= Coverage summary ===============================
Statements   : 82.94% ( 107/129 )
Branches     : 88.88% ( 8/9 )
Functions    : 43.33% ( 13/30 )
Lines        : 83.05% ( 98/118 )
========================================================================
```

We expect this to be significantly improved once we finalise our implementation and tools for system testing, as system tests would allow coverage of our entire architecture.

Future Improvements For Our Testing Suite

One of the main vulnerabilities we have identified in our testing strategy is the lack of automated end to end (E2E) testing. We acknowledge this flaw in our testing suite and are investigating how we would be tackling this in future deliverables. However, for this deliverable it is out of the scope, due to the amount of development going on. Our initial thoughts on this strategy would be to develop system tests along with unit test cases, which would simulate a server where we can hit each of the API routes with test input and expected output. The specifics of the tools which will be used are still in discovery, however we are hoping to utilise one that is well integrated with our existing Express and Jest code.

One other challenge that comes with having simulated system tests, is how to automate their environments with a pipeline. Due to the dependency of a database, system tests can only be run locally on a developer machine, as we want to avoid them running with the production database and instead on a temporary one on the local machine. Running on a production database puts at risk the live data of our application with the potential unstable changes during development, hence the reasoning behind the need of a temporary local date store. As a result of this, it is a major challenge on how we automate this and requires further discovery from the team in future deliverables.

Conclusion

In conclusion, we have put considerable thought into our testing strategy and tools as we value the importance of testing in validating the correctness and reliability of our implementation. Having outlined the core aspects of our project's testing strategy here, this report will be a vital resource throughout the development of our project, reflecting on our choices and ensuring a cohesive implementation and development practice overall.