Test Planning Document

January 21, 2024

1 Requirements

This test planning document will focus on testing the following 3 requirements described in LO1:

- **R1** - *The software should only allow correctly structured and formulated data into the database.*
- **R2** - *The software should retrieve the correct URLs (`video IDs`) that match the user's spoiler filtering options.*
- **R3** - *The software should censor the offending videos before the web browser can display them to the user.*

2 R1

I have identified the database as the most important component out of all the components in the software. This is because the database is the only component that stores data, and each other component relies on successful reads from (and sometimes writes to) it. Any issues with the database, from security to performance, will have a knock-on effect on the rest of the software.

For this reason, I will be utilising *Test-Driven Development* when creating the schema, routes, and other supplementary files for the database. This allows me to create a theoretical model for the database first, and then test it to ensure it works as intended. This will also allow me to test the database in isolation, without having to worry about the other components. It also allows me to add more functionality to the database and ensure the core tenets still work as intended. Additionally, the performance requirements mentioned in LO1 surrounding the database can be implemented here. These are very likely to always pass at first, but as the routes in the database grow, and more actors access the database constantly, we can ensure the high priority performance requirements are met.

The actual individual (unit) tests in question are quite simple. Each set will cover every parameter combination that constitutes a malformed entry (expecting a fail) alongside a single valid entry (expecting a pass). We will also implement the rest of the typical CRUD operations tests to make sure the database is working as intended.

There is little scaffolding required for the unit tests - they will utilise the already created CRUD operations, and if descriptive enough will highlight which specific operation failed. The performance tests will require more scaffolding, namely on the server side to log performance metrics on request. These may need to be stored to compare performance over time, but this is not a requirement for LO1.

Beginning this testing at the start (*TDD*) minimises the risk of the database "suddenly" failing, since if we re-run these tests after every new feature, we can pause development and fix it. I cannot foresee any risks with this approach as

long as the development process executes these tests constantly to ensure base functionality.

3 R2

These tests must be implemented after we have established confidence with the database (and with it, **R1**). This is due to it's heavy reliance on the database; to begin these tests we must populate the database with lots of example media, spoilers, and urls. This can hopefully be automated with systems that can generate enough content to adequately cover every combination. We could adapt this system to generate the tests too, but it will probably be less effort to manually insert data (while still automating all combinations with our known desired values).

This is because the extension may also require non-automated user tests, to see how "accessible" it is for a real user. Having real world data the user can pick between eases this test subject and allows us to simulate a real environment this software may be running in. Our testing approach will be combinatorial testing. This is faster than simulating (unit-test like) interactions with the browser extension, since at it's core we only want to focus on the queries sent and the responses received. We may add this unit testing in (interacting with the extension and then checking the query it sends) but this is outside the scope of the requirement. The only major scaffolding required for this test would be a system to generate all combinations of query from the database; in short, our required scaffolding is the functional database itself. The extension already stores the returned URLs, so it is trivial to add a debug logging system to return these queries to the test. The biggest risk to this test comes from setting up the aforementioned "real world environment". If we go with the manual setup we are open to human error, and if we decide to automate not only is this strenuous on development time but may also need to be tested itself. Having tests for scaffolding to create tests smells of bad design, so I will be opting for the first option and taking as much care as possible to create a good test environment.

4 R3

This test could be automated by checking the contents of the DOM after the extension has run.

We would take a "snapshot" after the software has ran and analyse this HTML to ensure our changes have been correctly implemented.

This requirement needs a lot of scaffolding. Not only do we need the correct URLs in the extension (either by satisfying **R2** or manual input), we will need software that can analyse the DOM page. We could also use a human checker to check the page, but this is not automated and is discouraged as it takes away from development time. We will also need the means to get information on how

long the DOM took to load, and how long the extension took to run. This is to ensure we are meeting the performance requirements ("*before the web browser can display them to the user*").

R3 has the most risks and problems associated, first and foremost being the DOM analysis. This is a very complex task, and we may need to use a third-party library to help us which is not ideal, as we are relying on a third party to ensure our software works. We could also use a human checker, but this is not automated and is discouraged as it takes away from development time. We could also use a third-party library to check the performance metrics, but these may incur their own time penalties.