Test Report - F21DG

Callum Stewart, Saad Badshah, Abigail Rivera, Daniel Scott, Bruce Wilson, Sebastian Zebrowski

1 Preface

1.1 Introduction

Contained within this report is a documentation of the testing strategy employed whilst developing this application. Thorough testing of system components, through both automated and manual techniques, has been utilized all along development to ensure the delivery of a robust, reliable software system that achieves the system requirements set forth in the Design Specification.

For the most part, this document is comprised of component level Unit Tests which verify that a given component works correctly with a mix of expected and unexpected behaviour patterns. Although it is hard, if not impossible, to anticipate every edge case that could occur, reasonable thought has been given towards likely behaviours that can be expected in day-to-day usage of the application.

Unfortunately, Integration Testing had to be done manually by developers close to the end of the project's development cycle due to issues with testing frameworks and last-minute snags which required urgent attention to address. The steps taken within these manual Integration tests, are however, documented within this report.

1.2 Explanation of Testing Strategy

As mentioned above, a combination of both manual & automated testing was rigorously employed throughout the course of development to ensure the final deliverable was up-to-scratch and met the requirements outlined in the Design Specification.

To perform our automated testing for our JavaScript components, the **Jest** framework was employed. Jest is a unit-testing framework developed & maintained by Meta (Facebook's parent organization), and as such came complete with extensive documentation and examples to reference from. Jest allows for configuration and tweaks to be made to it via an optional **jest: {...}** parameter in a project's **package.json** file. This came in especially useful when attempting to mock SessionStorage and DOMElement behaviours which, by default, aren't supported & aren't trivial features to implement.

Due to an unfamiliarity with the Jest framework, and the difficulty incurred when attempting to mock specific behaviours of Browser components and classes, there was an element of manual JavaScript testing involved as well. This manual testing was performed by writing out a series of objectives & example inputs that a class would be expected to deal with and then simulating user behaviour to assess the robustness of the specific functionality being tested (i.e. Uploading Files). The console in the Browser would be monitored for print statements and any warnings/errors that were thrown in this process.

Because of an unfamiliarity with GitLab and time-pressure stemming from upcoming deadlines, a CI/CD pipeline was not established for this project, which in turn resulted in a requirement for the Python code to be manually tested without a framework. Elements of this testing strategy have been preserved under the **tests** folder in the source code, specifically being used for generating .csv test data dynamically and then supplying it to the STFT logic to validate the analysis being performed.

2 Testing

2.1 JavaScript Automated Testing

This section details the automated test suites developed for the JavaScript portion of the application.

If elected, these tests can be ran by any end-user with access to the source-code by entering the following command in a terminal launched from the directory: **npm test**. This command runs all the test suites that Jest can find for the project and displays how many were successful and any errors or warnings that occur.

2.1.1 FILEIO.JS

The following functionality has been tested for the File Uploading functionality of the application.

1. Verification of file prefix: This test checks that the **fileTypeIsSupported(fileName)** method works as expected. The first test supplies a valid String filename, ending with a .csv prefix, and expects that the method will return ’true’.

   The next test supplies the same method with an invalid file name, *someInvalidFile*.**json** and expects the method to return ’false’. This is because the file extension is not supported, hence we don’t want it to return true for an unknown/unsupported file type.

2. Verification of csv input to JavaScript Array: The first test creates a valid String of example csv test data (emulating real behaviour, as the *FileReader* object of **parseFile()** method reads the csv as text.) A variable containing the expected output is then created below, but this time as a JavaScript array, and not encoded as a String object. The test then expects that, upon calling the **csvToArray(csvData)** method with the supplied, valid csv string data, the method will return an object matching in structure to the expectedOutput variable created earlier.

   The following test then verifies that, upon being supplied with invalid, non-numerical csv data, an exception is thrown with the message **Non-numerical data discovered in csv file.**

   Finally, the last test in this test suite verifies that csv data containing exponential values (formatted with e’s and -’s) is successfully read and parsed by the class. Identical to the original test for parsing csv data, an String representation of some example csv test data is created along with an expected output as a variable containing a JavaScript array. The test calls the **csvToArray(csvData)** method and expects to receive an object with a structure matching the expectedOutput variable created before.

That concludes the automated testing for the FileIO module. You’ll notice that there is no testing for ’reasonable’ or ’sensible’ data being uploaded to the application; the user should be able to experiment with the application however they see fit, and there should be as few guard rails as possible to prevent their exploration of these algorithms.

2.1.2 SESSIONSTORAGE.JS

This section was a little trickier to test owing to the fact that the test requires a Browser’s sessionStorage functionality in order to function correctly. As this test is running offline, absent a web browser, the sessionStorage functionality was initially very finicky and tricky to emulate/mock. Eventually a helpful library was discovered to emulate the sessionStorage functionality that integrated with Jest seamlessly; **mock-local-storage**. As the name implies, this library mocks the local (& and session) storage functionality of a web browser in the Jest framework. After the introduction of this library, testing was straightforward.

Referenced within the tests of this helper class are a couple of constants required solely for testing: **testKeyForS-essionStorage** and **validStringValue**. Both of these are relatively straightforward to comprehend: the first is the key which will be used to store the value of the object in session storage, the latter being the value to be stored.

The following is a record of how the functionality of this class was tested:

1. Saving string to sessionStorage should succeed: As the name implies, this test verifies that upon saving an value to sessionStorage with this helper class with a given key, the data is, infact, stored in the browser's sessionStorage.

2. Saving JSON Stringified object should succeed and return valid string: This test ensures that an example JavaScript array, preserved and converted into a JSON String, can firstly be stored in sessionStorage, and secondly can then be retrieved without any corruption to the format of the JSON String.

3. Removing string saved in sessionStorage should succeed: This test removes the value associated with the given key in SessionStorage and verifies that upon requesting the value of the key afterwards, without rewriting a new value to the key, a *null* is returned.

4. Getting data from nonexistant key should return nothing: This test verifies that upon attempting to retrieve a value for a key which does not exist in sessionStorage (emulating behaviour of mis-typed parameter in code), the behaviour is safely handled and only a *null* reference is returned.

2.2 JavaScript Manual Testing

Text goes here...

2.3 Python Manual Testing

Text goes here...

3 Conclusion

This concludes the test report for the application. As has been demonstrated above, a robust testing strategy has been utilized to ensure individual components were as expected. Suitable, reasonable edge cases have been chosen to test with, as has realistic/expected behaviour and the combination of the two has lead to a robust final product.

With more foresight, and a few less hiccups throughout development (specifically little niggles with PyoDide and SessionStorage), our Integration testing strategy would've relied less on manual testing by developers and could've been automated by either Jest or a python testing framework to further ensure end-to-end robustness of the application. It also would've been more desirable to identify more browsers/browser versions which support our application, however with many browsers electing to make auto-updating an automatic, turn-off feature (i.e. Google Chrome) it was believed that most users will either be using a browser, or have access to one, which supports the application.