Test Plan Report for open source JavaScript utility library

COMP.SE.200-2021-2022-1 Software Testing

Prepared by:

Lauri Viitanen (K98617)

Topi Huttunen (252791)

5.12.2021

Github repository:

https://github.com/Latemus/COMP.SE.200-2021-2022-1/tree/main

Table of Contents

Definitions, acronyms and abbreviations

- CI = Continuous Integration
- Jest = JavaScript test framework
- Travis CI = Continuous integration framework
- Coveralls = Test coverage reporting tool
- null = null is a primitive value in JavaScript that represents the intentional absence of any object value.
- undefined = undefined is the value of an uninitialized variable or object property in JavaScript
- NaN = The global NaN property is a value representing Not-A-Number in JavaScript

1. INTRODUCTION

This document is a test report for an open source JavaScript utility library. The library can be found from https://github.com/otula/COMP.SE.200-2021-2022-1. The purpose of the document is to describe the testing plan of the software library to the level where a third-party tester without prior knowledge of the testable software library would be able to test it following the test plan.

For the testing we have chosen to use Jest library for two different reasons, first we have some limited prior experience with it already, and second this allows us to further deepen our knowledge with the said testing frameworks. In addition during our initial research on JavaScript testing frameworks we have conducted that Jest works better on ES6 module structure than for example using MochaJS with Nyc coverage library.

Additional information about the target utility library from the assignment of this assignment:

> *The library is meant to be used in a front end application built with React. React promotes functional programming and composition over inheritance.*
>
> *The application is an E-commerce store selling food products from various small producers. Users can search products by category, price, producer and*

various other criteria. Products can be added to a shopping cart. Shopping cart automatically updates and shows the total price. Checkout process is handled with a third-party solution.

The food producers can add their products via a previously created portal. The producers can leave some fields blank if they do not want to specify some attributes like category or calories. It has been decided that the store front end is responsible for handling these missing values. Front end is also responsible for making sure that the product descriptions look similar i.e. the first word of a sentence starts with an upper-case letter and that prices are shown with two decimal accuracy.

2. OBJECTIVES

Objective of the test plan is to have as wide of a coverage of the interfacing functionalities of the system based on three factor:

- objective coverage of the test
- prioritization of the functions by their level of difficulty of implementation
- functionalities working with numbers

The first one tries to achieve as wide as possible percentual coverage of the tests run by testing functions which use other functions of the interface. This can be considered to be a sort of integration testing of the different units within the program even though it’s a type of unit testing of the said function.

The second tries to achieve the test coverage to functionalities which are determined to be harder to implement code-wise thus having a higher chance of including errors.

The third one is prioritizing functions working with numbers due to the application being a front-end of an E-commerce platform selling various products. Even though the basic math operations being easy to implement they are highly critical in terms of the product, thus these are prioritized. The prioritization chosen also defines what is left out of the scope, in other words, the functions that do not fall into the criteria selected.

3. SCOPE

The scope of this test plan is to implement adequate unit test coverage for 10 most prioritized methods from the library's src folder. As this library contains only utility methods without any business logic or abstract interfaces, no integration tests will be implemented. In addition no system or acceptance tests can be performed, as there are no system requirements or documentation that could be referenced. We will exclude everything inside path src/.internals as they are not in the scope of this test assignment.

3.1 General

The chosen source files are listed on table 1. below, with justification why the said file is tested and the interfacing functions the said functions either use for provide information to:

Function name	justification for testing	interfacing functions
add.js	critical math function	createMathOperation
divide.js	critical math function	createMathOperation
difference.js	uses two internal and one interfacing function	baseDifference baseFlatten isArrayLikeObject
countBy.js	uses one internal and one interfacing function	baseAssignValue reduce
compact.js	secures the sanitizing of the data. Critical for preventing usage preventing bugs	-
eq.js	due to Javascript it's highly valuable to check	-

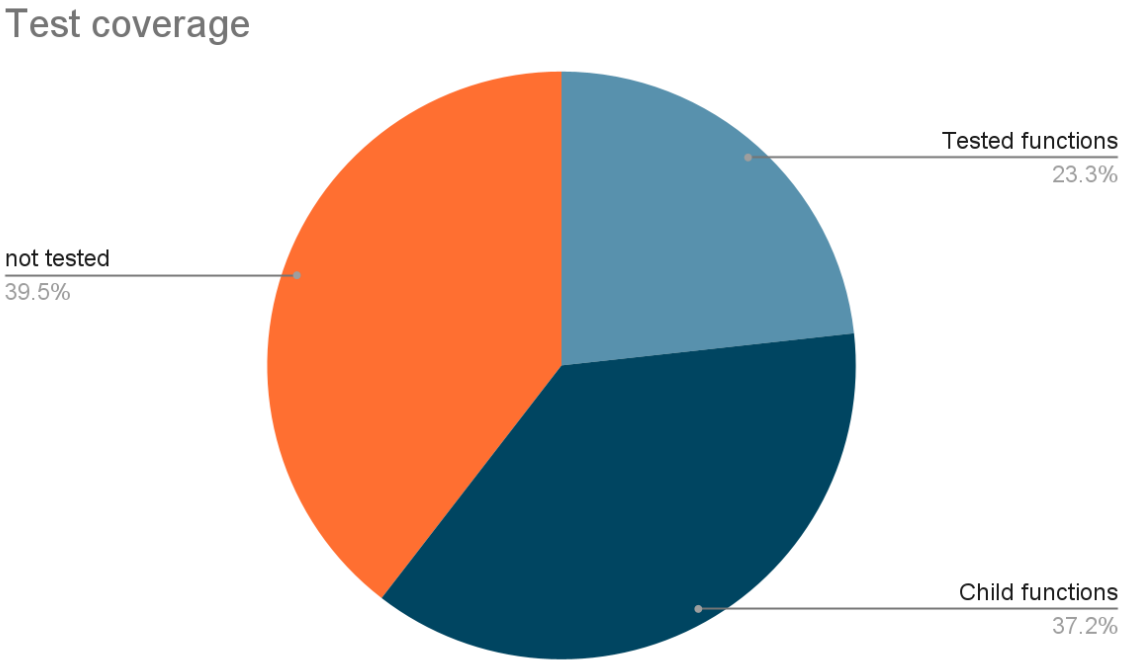
	how the “equal” interoperates within the software	
isDate.js	uses two internal and one interfacing function	getTag isObjectLike nodeTypes
drop.js	uses two interfacing functions	slice toInteger
toNumber.js	uses two interfacing functions	isObject isSymbol
isEmpty.js	uses two internal functions and four interfacing functions.	getTag isArgument isArrayLike isBuffer isPrototype isTypedArray

3.2 Prioritization

As we have only limited time and resources to test this library, we have chosen 10 source files from the utils to be tested. As the problem domain of this test plan's target is an E-commerce store, we have decided to prioritize source files, which function can be said to be integral part of a working E-commerce webstore. We have concluded that mathematical functions are of high priority, and have taken that into account in prioritization.

3.2 Coverage

The test coverage of the functions of the interface is illustrated in the pie chart below:



Tested functions means the functions that are directly tested, whereas the child functions are functions that are called within the tested functions or other child functions of the tested functions.

3.3 Reporting

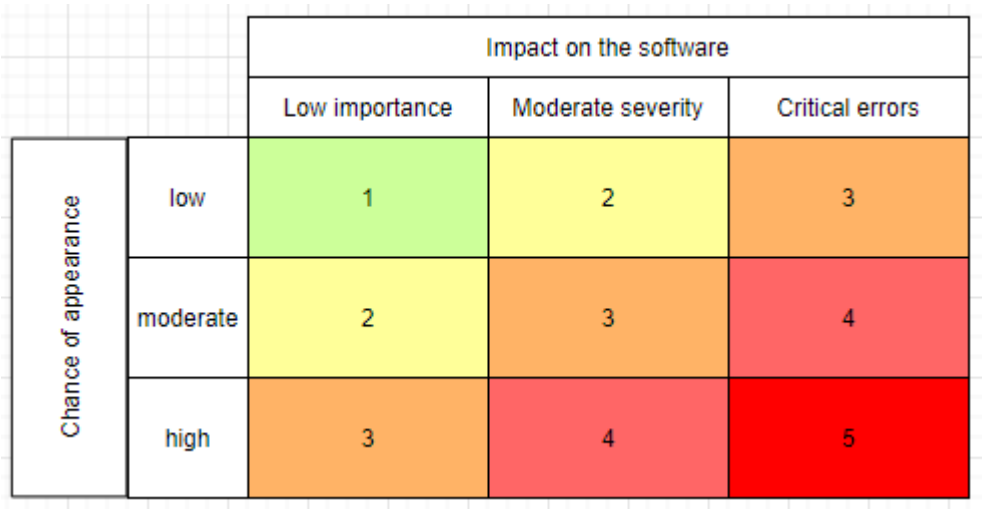
The final report contains the test results of the tests run, a description of what the issue is, and a proposition for a solution to fix the issue. The final report / document should contain enough information which allows the developer to have a valid idea of what the problem is and the first step on how to approach fixing this, thus allowing a smoother transition to the next version of the software and continuity to testing of it. Coverage reports will pre presented in this project's Coveralls page. Coverage reports are generated with Jest.

If bugs / issues are found their impact is classified based on the following categories:

- Errors or Bugs of Low Importance
    - Errors that have only low impact. For example cases which should not occur in the normal usage of the software or because of typical checks that are normally done by the interface. In addition cases which can be found with values at the limits of data type value range.

- Issues of Moderate Severity
    - Error or bugs that could occur in the normal usage of the software or error that could have negative business influence for any party using this library. For example error, where arithmetic operation returns incorrect or inaccurate result
- Critical Errors
    - Errors or bugs that could have severe consequences or even break the software that is using this library. For example cases, where JavaScript engine throws an unhandled exception

Also for the bugs / issues found an estimate on their probability is assumed based on prior experience and knowledge. With the two classifications a risk management matrix is formed to classify the bugs based on their criticality. An example matrix is found in the picture below.



The matrix classifies the bugs into five categories, which prioritises the importance of the bugs to be fixed.

4. SCHEDULES

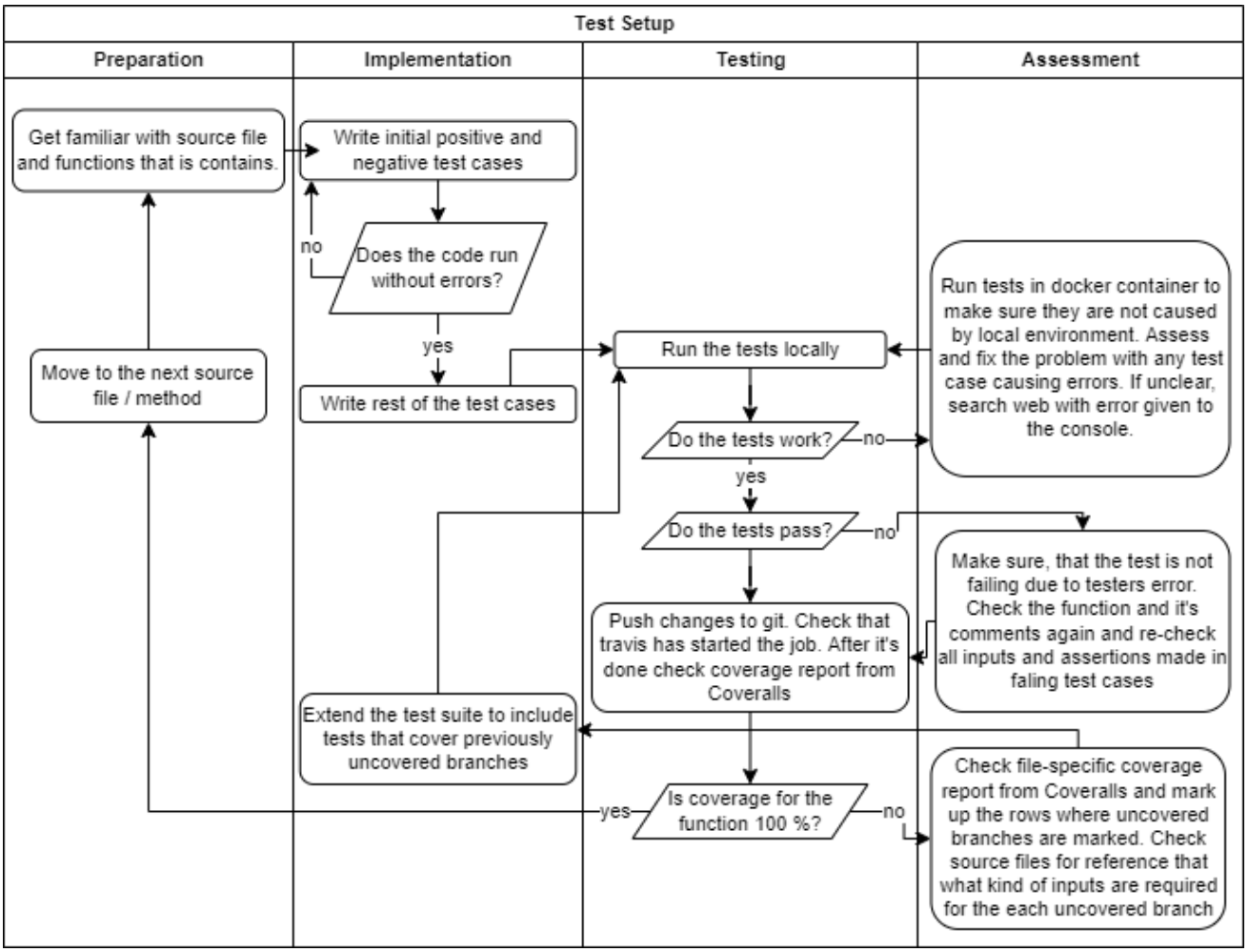
Milestones for the testing are the following:

- Testing plan frame 2.10.2021
- Finished testing plan by 17.10.2021
- Unit test written 15.11.2021
- Automated regression environment set up 20.11.

- Testing pipeline built with the unit / integration tests 30.11.
- Report finalized and returned 5.12.

5 TESTING STRATEGY

We will implement unit tests for this library using Jest JavaScript testing library. In addition we will set up a GitHub repository, configure a continuous integration pipeline using Travis CI and use Coveralls test coverage reporting to assess test coverage. As the target framework is written in javascript that is dynamically typed we shall take that into account when writing each unit test. Flowchart of testing methodologies can be found in the diagram below. After testing is done, coverage reports will be included as part of the testing report.



5.1 Tools

Tools chosen for the testing purposes are the Javascript testing frameworks / libraries Jest [1]. We will use Jest's built- in coverage reports to assess how well our source files are covered and how For continuous integration the Travis CI[2] is given as a requisite for the assignment. For test coverage reporting the Coveralls[3] is used, reason being the same as with Travis CI. These were chosen based on the previous experience of the testers thus reducing the possibility of errors and learning curve required for the usage of said tools. Also, the libraries are widely adopted in the industry 'hence gathering more experience with them was considered to be a major advantage. For running the tests in a virtual environment we have chosen to use Docker, as it provides us all the tools necessary, is easy to use through pre-written scripts and contains little to no overhead in terms of configurations and files. Main advantage of using a virtual running environment is that each time the tests are run the environment is the same and the host machine won't affect their outcome. Tests will be implemented so that they can be run with any host operating system through Docker.

5.2 Assumptions

As we are working with JavaScript it is important to define our assumptions. We assume that if no further information is given in documentation or comments of tested functions, it should work with base logic of JavaScript, even though it sometimes contradicts basic logic. For example in JavaScript value NaN is not equal with value NaN and string '1' is equal with number 1 if type free equality is used (==). If other information is given, it is taken into consideration and those functions are tested with given constraints. For example in function eq (eq.js) additional information is given, that the function is performing a 'Same value zero' comparison between given elements. In addition, type definitions of each function block comments are checked. For example we assume that add-function (add.js) should always return a number-type value or NaN as it is defined in it's comment.

5.3 Unit Testing

Unit testing will be performed using Jest-framework. In unit testing we will test each source file and method independently with various different input values. Input

values will be determined based on each function's parameters data type. These basic test values are determined below based on the assumed data type of the parameter. In addition we will be testing each method's data type restrictions by inserting presumably incorrect types of data.

- Numeric parameters
    - -1/0, -1, 0, 1, 1/0
- String / text parameters
    - “” (empty string), new String, “Mary had a lamb” and {text: “test text”}
- Object
    - Object(), [], null, undefined
- In addition we will be using error prone values, like null, undefined and NaN in as function call parameters

5.4 System and Integration Testing

Integration of different functionalities is achieved by testing the functions calling other functions. This allows the testing of the integration of lower level functions and their work principles within the higher-level function. Methodology to achieve the integration tests is to write the appropriate unit tests for the higher-level functions with the tools chosen.

5.5 Continuous integration

Continuous Integration is achieved by using Travis CI, a continuous integration platform. For code changes pushed to the chosen version control system, Travis CI automatically clones the code into a new virtual environment and does a series of tasks to build and test the code, thus determining whether or not the build can be considered passed or broken. After each build a coverage report is generated and visualized in Coveralls.

5.6 Pass Criteria for the Library

Our pass criteria for the library is that no critical errors or issues of moderate severity are found. If issues are found we will assess their impact on the target library and try to suggest fixes for them.

6. IMPLEMENTING AND RUNNING THE TESTS

We have implemented tests for each function defined in section 3.1, built the pipeline defined in section 4.5 and written scripts to run these tests in Docker container. All files for the target library and all tests are included in this test project's GitLab repository [5]. Test coverage reports are found in the projects Coveralls page [6]. Additional information for running the tests locally or in Docker can be found from the reporisory's README.md.

Out of 10 test suites (one for each function) only 4 passed. All in all we implemented 101 tests out of which 25 failed. Next we go through each function that we chose to test.

6.1 Test results by function

add.js

- coverage 100%
- 20 test cases, 18 passing
- Issues found:
    - If the Add-function is given a string type of input, it will concatenate it with the other input resulting in a string type of output. This is an issue, as the functions comment defines, that the functions parameters and output should be of type number. Because of that, we did assume that giving the function parameter of string type it would return NaN (Not a Number). This is considered an issue of moderate severity as the function works well with any number type of inputs.
        - Example: add("1", 1) will return "11" instead of 2
- Possible fix:
    - Make the function check that all parameters are of type number and return NaN otherwise OR if string is given as a parameter, try to convert it to number and then calculate addition

compact.js

- coverage 100 %
- 4 test cases, 1 passing
- Issues found:
    - Removes the first element of a given collection.
        - Example: compact[1, true] will return [true]
- Possible fix:
    - row 15 change 'resIndex = -1' to 'resIndex = 0'

countBy.js

- coverage 100 %

- 3 test cases, 1 passing
- Issues found:
    - returned counts are one less than the real number of properties. Could not determine what caused this issue. This is considered to be critical error, as the function is not working at all as intended or as described in the comments

difference.js

- coverage 100%
- 13 test cases, 13 passing
- no issues found (all tests passed)

divide.js

- coverage 100%
- 19 test cases, 4 passing
- Issues found:
    - Given dividend is never used, as the division is done by dividing divisor with itself. This is considered a critical error with high change of appearance.
        - Example divide(3, 9) will return 1 instead of 3
- Possible fix:
    - This can be easily fixed by changing the function call to be ‘dividend / divisor’

drop.js

- coverage 88.89 %
- 8 test cases, 8 passing
- no issues found (all tests passed)

eq.js

- coverage 100 %
- 10 test cases, 9 passing
- issues found:
    - boolean true and integer 1 returns true instead of false, this is considered a high risk bug due to if integer values of 1 and 0 are taken as boolean instead within the system.
    - risk factor: 3
        - change of appearance: high
        - impact on the software: low
- possible fix:
    - use strict comparison (===) in line 32 of the source code of eq.js instead of equality operator (==).

isDate.js

- coverage 70%
- 4 test cases, 4 passed
- no issues found (all tests passed)

isEmpty.js

- coverage 78.79 %

- 13 test cases, 13 passing
- no issues found (all tests passed)

toNumber.js

- coverage 85 %
- 9 test cases, 7 passing
- issues found:
    - issue with detecting binary and octal values. Given a binary string of “0010” expected outcome is the decimal value of 2, instead gives out 10. Given an octal string of 30041 expected outcome is the decimal value of 12321, instead gives out the 30041.
    - risk factor: 4
        - change of appearance: moderate
        - impact on the software: critical
- possible fix:
    - go through the test logic and the regex templates to detect the binary and octal formats.

6.2 Final verdict

As reported above the target library has multiple critical errors and as such should not be used in its current state in a production environment. We have suggested possible fixes to multiple issues found. Out of 43 functions inside folder src we chose 10 to test as instructed. From these tests we have gathered coverage reports that contain 25 files. This means that our chosen function uses other functions from the scope of this testing. From these 25 files we have achieved statement coverage of 85 %, branch coverage of 65 %, function coverage of 75 % and line coverage of 85 %. We could not achieve 100 % coverage for all 10 of our chosen functions and could not find exact solutions to found issues as we do not have enough expertise with JavaScript. However we feel confident that our tests catched most of the errors in our chosen 10 files. We are not confident at all on the general quality of this library. As we found out that 60 % of our tested functions did not pass the tests, we would assume that the rest of the 33 functions probably contain a high number of errors as well. To conclude we would suggest extending this testing to include all of the files in this target library and fix all found errors before considering usage of this library in customer applications and usage.

References

[1] Jest test framework: https://jestjs.io/

[2] Travis CI: https://travis-ci.org/

[3] Coveralls: https://coveralls.io/

[4] Docker: https://www.docker.com/

[5] This project's Gitlab repository:
https://github.com/Latemus/COMP.SE.200-2021-2022-1/tree/main

[6] This project's Gitlab repository:
https://coveralls.io/github/Latemus/COMP.SE.200-2021-2022-1