Test plan

Comp.SE.200 Software Testing

TOMMI ILMONEN 253543, VILLE SAARINEN 252987

Table of Contents

Changes

Version	Date	Changes
V1.0	18.10.2020	-
V1.1	07.12.2020	- Take the whole application to consideration in the introduction. - Updated documentation of the test results.

Definitions and acronyms

CD	Continuous Deployment
CI	Continuous Integration
CLI	Command Line Interface
npm	Node Package Manager, registry for JavaScript third party packages

1. Introduction

This document is the test plan for the provided utility library [1]. The purpose of this test plan is to show what, how and why parts of the library are tested. One other aim of this document is to help the next phase of this assignment, so that testing is carried out as planned.

It is impossible to make tests and test planning so that the functionality of the testable library is guaranteed to be error free, since testing cannot find all the possible mistakes. Therefore, the purpose of the test plan, and its tests in the next phase, is to provide more information about the errors, wanted functionality and quality of the library to be tested.

This library will be part of a larger shopping application created with React.js. The aim of testing in this assignment is to test the quality of the given library for the parts needed in the application. This can be done by planning testing so that it covers many high-quality test cases that are able to find possible errors or unwanted functionality in the given library.

In the planning phase of testing, we must determine what do we test and how. This is simply because everything cannot be tested. The amount of test cases can be reduced dramatically with equivalent test cases. This means that a function does not need to be tested with all possible correct input values, only one or few is enough since they are equivalent test cases. Planning the test cases so that they cover as many situations as possible is a good way to keep the amount of test cases sensible. Also, this way tests are carried out in a way that focused more on quality rather than the magnitude of tests.

The testable library is written in JavaScript and consists of simple helper functions and can be considered as a utility library. Since the library contents and functionalities are simple functions, unit testing is mainly used. The baseline for the unit tests is that both negative and positive testing is used. In positive testing the aim is to test that the function does what it is supposed to with correct values as well as what happens in boundary conditions. The negative testing ensures that wrong or unwanted inputs are handled correctly.

2. Used tools

Tools to be used are mainly installed from npm registry, since it automates the installing process of the project when set up to a new location. This requires Node.js and npm CLI to be installed on the system [2]. Npm packages are configured in package.json file in project's root directory. When the project is cloned, those packages can be loaded and installed with 'npm install' -command.

The library to be tested is written in JavaScript. Therefor we chose to use Jest [3], or to be more precise Jest-extended [4], as our testing framework. Jest provides easy way to construct unit testing and it has also ability to give test coverage reports.

Jest can also be easily migrated to Travis CI, Coveralls and GitHub which are used in the next phase. Testing framework is not used in this phase but is involved in planning because the aim of this document is to give guidelines for the second phase.

We will be using Travis CI as a testing and integration platform for our project. It allows automatic testing and code integration on pre-defined GitHub events, such as push and pull request. It also provides extensive user interface to examine the test results. [5]

As a coverage reporting tool, we will be using Coveralls. It can be easily integrated to GitHub and more importantly to our Travis CI pipeline. It provides a website which can be used to view automatically created test coverage reports. It also provides an overall coverage button which can be embedded to our project's GitHub readme-file. [6]

3. Test plan

This section provides more details on the actual tests to be performed. It also describes how the test results are categorized and how they can be evaluated.

3.1. Tests to be performed and test cases

The given JavaScript library consist solely of helper or utility functions. Therefore, only unit testing can be done since there is no reasonable way to construct integration tests simply because there is nothing to be integrated with each other.

In unit testing we focus on the most important methods of the module. In this case, since test objects are helper functions, the focus is on the:

1. Return values and basic functionality of the functions. Functions should do what they are documented to do and should work on all correct parameters, including boundary conditions.
2. Functions should not alter given parameters if parameters should not be altered.
3. What happens when functions are used with incorrect parameters. Suitable exception or describing error should be thrown/shown or the module or function should recover by itself if possible.

This means that unit testing is carried out so that with correct input the returned values should be what expected. Also, return values should be correct in boundaries, that is with limit values, if those exist in the unit to be tested. Big focus is also on error situation testing, that is what happens when functions are used with incorrect parameters.

The test cases are planned with equivalence classes. With equivalence classes the same case covers large amount of different test cases. For example, we should not test addition of two numbers with all possible correct inputs, since all the test cases are equivalent, one or a few cases would suffice. This approach keeps the amount of test cases sensible and provides more tests with higher quality.

We will be also using manual glass box testing to test and ensure quality, coverage, and sufficiency of the documentation. Documentation is important part of any program and it needs to be good enough to support the usage of this library. This will not be part of the automated test cases, but a manual investigation done by tester.

3.2. Definition of passed test

Below is defined criteria for different level of test to be considered passed. These are minimum terms for a test to be passed. Tests can be also more detailed and comprehensive.

Singular test case is considered to be passed when these conditions are true

1. Test case passes all assertions

Function is considered to be passed when these conditions are fulfilled

1. All test cases for the function pass
2. All different (non-equivalent) test cases are covered
3. Functions test coverage is at least 95%
4. There are no documentation related issues

Tests are considered to be passed when

1. all functions pass their tests
2. overall test coverage is at least 95%
3. there are no documentation related issues on the library itself

3.3. Parts of the library to be tested

This library is so small that we can write unit tests for all the functions that cover the approaches listed in the above ***tests to be performed section***. Though, this does not mean that everything will be tested, since it is impossible, but will give a reasonable result of the quality of the library.

The source folder contains also '.internal'-folder which is used by this utility library under testing. This folder will be excluded from the testing as well from coverage and issue reports.

3.4. Categorizing of found errors and issues

Issues will be categorized first by the nature of the test: positive testing, negative testing or documentation. In positive testing the test generates an issue if it does not work correctly with correct input. These issues are classified with critical or error severity depending on the quality of input value. On border condition input values, the issue will be labeled as error. With normal values inside the given input constraints issues will be labeled as critical since the function does not work at all with most of the allowed values. On border cases the function does not work only with those few border cases, so the function is still somehow usable, therefore the issue is not as critical, but still severe.

In negative testing we feed the functions unallowed values by purpose. We expect the function to handle occurring errors within the function or to throw documented exception. With these tests the possible issues are that the occurred error is not handled at all ('Not handled') or the error is handled in a way that is not documented i.e. not in expected manner

('Incorrectly handled'). Both of these issue types are on error severity, since they do not prevent the main usage of the function.

In documentation testing there are two sub-categories, missing documentation and incorrect documentation. Missing documentation has severity of warning since it does not directly harm the usage of the library. Incorrect documentation will be labeled with error severity because it leads the user of the library to possibly use it incorrectly.

3.5. Documentation of test results

Test result reports will be automatically documented by Travis CI and Coveralls in their corresponding websites. From Travis CI each test case can be separately examined by the user. More detailed document will be created by the testing personnel on the basis of these automatic reports. This document will include all issues that arise from failed tests. These issues will be categorized with the classes from chapter 3.4. This document will also include issues from glass box testing, which mainly emphasizes the documentation of the tested library. All of these issues will have information about the error and the reasons behind it.

4. References

[1] COMP.SE.200-2020-assignment, library under testing, GitHub repository, website. Available (referenced 17.10.2020): https://github.com/tgcslearningtech/COMP.SE.200-2020-assignment/tree/master/src

[2] Downloading and installing Node.js and npm, npm documentation, website. Available (referenced 17.10.2020): https://docs.npmjs.com/downloading-and-installing-node-js-and-npm

[3] Jest, official home page, website. Available (referenced 17.10.2020): https://jestjs.io/

[4] Jest-extended, npm registry, website. Available (referenced 17.10.2020): https://www.npmjs.com/package/jest-extended

[5] Travis CI, official home page, website. Available (referenced 17.10.2020): https://travis-ci.org/

[6] Coveralls, official home page, website. Available (referenced 17.10.2020): https://coveralls.io/