Rasmus Salla 276441
Janne Vikelä H293494

TESTING PLAN

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

Information technology and Communication Sciences
09/2021

CONTENTS

1. INTRODUCTION 3
2. TEST ENVIRONMENT 4
    2.1 Utilized tools 4
    2.2 Test setup diagram 5
3. TEST CASES 7
    3.1 Testing practices 7
    3.2 Selected parts of the library to be tested 7
    3.3 Excluded parts 8
    3.4 Planned documentation 8
    3.5 The bug classification 8
    3.6 Test cases 9
REFERENCES 10
APPENDIX A: TEST CASE PRIORIZATION 11

DEFINITIONS, ACRONYMS AND ABBREVATIONS

JavaScript	Programming language
React	Open-source front-end JavaScript library
MoSCoW method	Prioritization technique
GitHub	Distributed version control system
Travis CI	Continuous-integration service for GitHub and Bitbucket
npm	Node package manager
Mocha	JavaScript testing framework
mochawesome	HTML/CSS test reporting tool for Mocha
Coveralls	Test coverage tool for GitHub and Bitbucket
GCOV	Coverage data file format
LCOV	Graphical front-end format for GCOV data
mocha-lcov-reporter	Generates LCOV data from Mocha's test reports
Chai	Assertion library for testing with JavaScript

1. INTRODUCTION

The purpose of this document is to act as a testing plan for a JavaScript library. The library contains general purpose algorithms for algorithmic, numerical, string and type comparison tasks, and it is to be used with React application for E-commerce purposes.

This testing plan seeks to introduce the testing environment that is used, what tools were chosen for the testing, what kind of tests are performed etc. Formatting a testing plan for this library is done to document the testing process, as our library serves an important purpose in the functionality of the React application.

Additionally, the testing process has a strict time constraint so test prioritization is a must. Constructing a testing plan helps to formalize this prioritization process. The test case is prioritization is done using the MoSCoW method.

In chapter 2 the testing environment is introduced, which will contain the chosen tools, libraries and packages used for the testing along with an overall view of test setup. In chapter 3 the testing process shall be introduced, which contains the types of tests that are done, what parts of the library are tested, what are excluded, test result documentation process, and finally the qualification process of the eventual actual test results.

2. TEST ENVIRONMENT

2.1 Utilized tools

The code for the library and the tests is hosted in GitHub. This was a requirement set by the course's assignment specification, and we had nothing against this. GitHub offers a strong integration with many continuous integration tools and testing frameworks while also being a best-in-class version control system. The repository is hosted under the name *COMP.SE.200-2021-2022-1* under the author *gitranes* (link), which is a fork of the course's repository (link).

The GitHub hosted repository will be utilizing a continuous integration pipeline hosted by Travis CI. This was also a requirement set by the course. The reason why Travis CI was chosen is that it offers an unbeatable integration with GitHub to make executing vast number of tests easy with every change to the library or tests.

Considering that the library and tests will be made in JavaScript, naturally our project will also include npm (Node package manager) as a package manager. npm will be utilized in almost every step of the testing process from building, execution to test result generation. All of the tools of our testing environment integrate with npm somehow.

For our testing framework we decided on Mocha. The reasoning for this is that it is popular choice and the only framework we had any real experience from. Mocha integrates well with GitHub and Travis CI, which would also make it a rather natural choice.

To complement Mocha's reporting capabilities, we will be integrating mochawesome as a test reporting tool to our Travis CI. This tool helps to create more readable test results from Mocha's test results as HTML/CSS, the results of which can be utilized in our test report.

The testing coverage will be reported using Coveralls using the mocha-lcov-reporter. Coveralls is free-to-use for an open-source repository, and it offers great testing coverage tooling for many different languages. Even though code coverage is by no means an absolute software metric, it can be used to roughly see how much library code our tests test. The mocha-lcov-reporter package is used to convert the Mocha coverage results to a format that Coveralls understands.

Finally, for the actual testing we will be utilizing Chai, because again, this is the only testing library we were somewhat accustomed to. Like many of our other tools Chai integrates directly with Mocha, and thus also with our existing tools. Chai offers a vast

selection of test assertion tools for plain asserts to exceptions and so on, which will be heavily utilized in our tests.

2.2 Test setup diagram

In the previous chapter the contents of the testing environment were introduced along with a set of tools for test execution and reporting. This chapter seeks to give a general view of how these tools work together to eventually test the library and report the tests' results.

For the purpose we chose to provide the general view as an UML sequence diagram instead of an activity diagram. This is because in the sequence diagram the actors, which in this case are the tools of our environment, are more apparent. This makes interpreting the diagram much more natural. Furthermore, the CI pipeline with all its steps is a sequence by nature.

The resulting sequence diagram will be abstract in nature, as the specifics of the signals between the different tools and the implementation of the tools is not important. The diagram's purpose is to provide only a general view of how the tools operate together.

The sequence diagram is visible in figure 1. From the figure you can see that the tools described in the previous chapter are at the top, and that the sequence describes a change in the version control followed by test execution and eventually test result gathering.

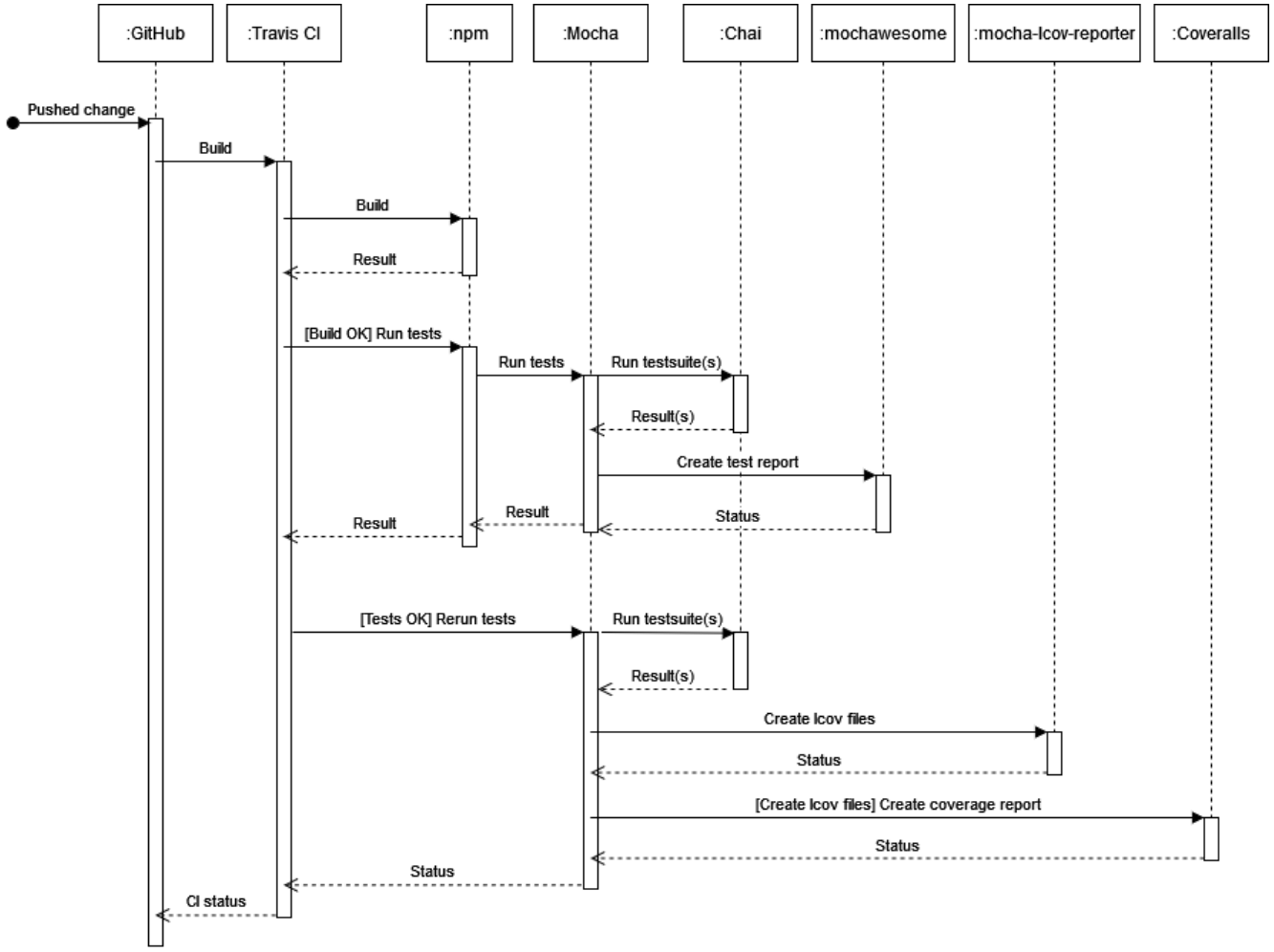

Figure 1. *Sequence diagram of our test setup*

From figure 1 we can tell that Travis CI is the main driver in our test environment, as it is responsible for starting the execution of all our tools. The CI pipeline process begins with building the library and tests. If the build step is successful, the pipeline continues towards executing the tests for the first time.

In the first test execution the npm uses mocha with the mochawesome set as a test reporter to run all your Chai test files (suites). Whether or not the Chai tests successfully executed, the test report will be generated. We are planning to include a link to this test report in GitHub along with the CI result.

Then if the first test execution is successful, the pipeline executes the same set of tests again using only Mocha (instead of npm). Though, this time the reporter is set to the mocha-lcov-reporter, which is able to generate lcov files from the results required by Coveralls. If the lcov generation is successful, the results are forwarded to Coveralls, which generates the coverage reports. Much like with the mochawesome test report, we are also planning to include a link to the coverage report in GitHub.

3. TEST CASES

3.1 Testing practices

The testing in this exercise is limited to unit and integration testing. No further testing (System, acceptance) is required as the actual system where the library is used exists yet. Moreover, the current library consists of relatively simple functions, the testing of which mostly requires assertion of the correct output. In addition to unit testing, some integration testing is to be conducted as we want to make sure that the methods work together in tandem.

In practice, the integration testing is done with the functions that use other functions in their action. Also, some integration testing may be done by feeding the output of some function as input to other.

3.2 Selected parts of the library to be tested

The functions to be tested are selected by two criteria. The first is how complex they are. For example, functions with iterations or functions using other functions are picked. The simplest functions are excluded simply, as they can be reviewed statically. Also, the most relevant ones based on the story are picked.

The methods to be tested are selected based on the MoSCoW principle.

The attached **excel file** contains the MoSCoW classification of the functions to be tested. Only then ten Must test functions are listed below in Table 1:

Function	**Rationale**
camelCase.js	Quite complex function, which is used elsewhere.
chunk.js	Complex and integrates other functions, incl. Slice
countBy.js	Uses other functions and somewhat complex.
isBuffer.js	Convoluted and tricky to comprehend
isEmpty.js	Extremely tricky implementation
isTypedArray.js	Uses complex regex
memoize.js	Complex implementation
toInteger.js	Tests almost all of the other to methods.
toString.js	Complexity
words.js	Complex regexes

Table 1: *List of functions to be tested.*

3.3 Excluded parts

The attached excel file contains all functions classified by the MoSCoW principle. The other than Must functions are excluded.

The exclusions of some functions are done for three reasons:

- Simple functions: Some of the functions are so simple that their testing is unnecessary and too time consuming, as static analysis is enough to proof the functions
- Functions that can be tested by other functions. In this case, a function that is used by other function is tested with the higher level function. Thus, the lower level function is excluded
- Irrelevance for user: Some of the functions bear features that will most likely be unnecessary in the online store use context

3.4 Planned documentation

The functions tested will be listed and the test type by each function will be mentioned. For example, a function is tested for equivalence testing and limit values. Then, the tests are listed on a high level. The high level tests may be “Correct input”, “False input”, “Correct limit behaviour”, “False limit behaviour” etc. Then, if the output is correct, the test is labeled success.

If output is fail or function malfunctions, the nature of the failure is documented. For example, wrong output is listed and the input leading to it also mentioned for further correction. Also, if the malfunction is not limited to output, the other error classes are documented based on their error message.

3.5 The bug classification

The bug classification is divided into two sub-classes:

- *Wrong output:* The function runs properly but returns incorrect value.
- *Error:* The function encounters an error when testing without returning value.

With the wrong outputs, the errors are then further explained in the documentation by listing the input and output for further tracing.

With other errors, the errors are classified by the error hierarchy for further tracing.

3.6 Test cases

Mostly, the testing is focused on equivalence testing. The inputs are tested with clearly correct inputs and clearly false inputs to see if the outputs behave as expected. Moreover, with functions including value limits, indexing or loops, the testing also is focused on limit value testing to see if the conditions hold and indexing runs properly.

REFERENCES

Group repository in GitHub	https://github.com/gitranes/COMP.SE.200-2021-2022-1
Course repository in GitHub	https://github.com/otula/COMP.SE.200-2021-2022-1
Travis CI homepage	https://www.travis-ci.com/
npm homepage	https://www.npmjs.com/
Coveralls homepage	https://coveralls.io/
Coveralls package	https://www.npmjs.com/package/coveralls
mocha-lcov-reporter package	https://www.npmjs.com/package/mocha-lcov-reporter
Chai homepage	https://www.chaijs.com/
Chai package	https://www.npmjs.com/package/chai

APPENDIX A: TEST CASE PRIORIZATION

See excel file **test_cases.xslx** provided in the zip.