Smart NFC Card Application

Testing Policy

Vast Expanse (Group 7)

COS 301 - 2019

Wian du Plooy	*u17237263*
Duncan Vodden	*u17037400*
Tjaart Booyens	*u17021775*
Savvas Panagiotou	*u17215286*
Jared O'Reilly	*u17051429*

Table of Contents

Introduction

This document will outline the testing policies followed by the team throughout the lifetime of the project. It shows the types and methods used for testing the code as well as give examples of some tests.

Objectives

Through the course of developing the application, testing needs to be done and enforced by everyone on the team. Testing the application will ensure all features work as expected and if something went wrong, you can find the problem and fix it.

Tests need to pick up on errors that were not found in the developing of the functions. It needs to be extensive and it needs to test all aspects of a function to find areas where it might not function properly.

Types of Testing

The purpose of testing is to ensure that the software does what it intends to do. We need to make sure that all functions of the service behave as expected when certain parameters are given as input to ensure that expected output is given to the end-user. This section provides definitions of the different types of tests we have implemented for our software.

Unit Testing is when individual components (units) of the system gets tested to ensure that the basic logic of the unit works. The tests use stubs/mock data as parameters and/or outputs. Unit tests can be run at different granularities ranging from testing the program/class to testing individual functions.

Integration Testing is when different components (units) gets tested together to see if they integrate as expected. Integration Testing takes place after all related unit tests are done, to ensure that the individual units aren't causing the problems.

Regressions Testing is used when major code changes were made, to ensure that all other functions still work, and no new bugs are created. It ensures the integrity of the system after new functions, etc. got added or updated.

Continuous Testing is when tests are run automatically as part of the development process. As code are put/merged together, the tests are run to get immediate feedback on the success of the merge.

Methods

We use a conjunction of Agile testing and White-Box testing. This is because our main design principle is agile, and because all the developers need to do testing as well.

White box testing verifies internal workings of the system.

This ensures that we can change functions any time during the project lifetime, if you update the unit tests relating to that function. Regression tests should also validate whether the change to a function affected the working of other functions.

Also, since all testers are developers as well, they know what are valid parameters and what are invalid. Thus, you can implement tests to ensure expected output is received with every case. We have predetermined inputs with expected results which are checked against the outputs.

Coverage Criteria

Function coverage - Unit tests need to cover all the functions to ensure that each function works as expected.

Statement coverage - Tests need to cover all the source code to ensure that no code goes untested.

Condition coverage - Tests should cover all possible scenarios of conditions. For example, the ‘if’ and ‘else’ part should be tested.

Required Resources

- Required Software for the tests to run:
    - Jasmine
    - NodeJS
- Time
    - Tests should not run longer than 5 minutes on localhost
    - Due to Travis CI running tests on pull requests across the internet, it should not take more than 10 minutes to complete the tests, depending on the server load

Description of the Test Process

After a specific function is coded, the person responsible for the function is in charge of writing several unit tests for that function and ensuring that the test passes.

All tests that will be written will comply with a strict and uniform structure that will give enough detail of what the test is actually testing. All tests should be short, concise and to the point, while still providing enough information in the report so that people running the tests are sure which tests are run and on what part of the software it is running.

Testing will be done through the Jasmine testing framework since jasmine has a wide variety of available tests and is well known with the JavaScript language. Tests will also be run automatically by Travis CI on pull requests to ensure that the software is in an acceptable state before merging into a different branch. This will ensure that bugs are identified before merging it into a branch that should be in a deployable state at all times.

The logs of Travis CI tests can be found here Travis CI.

When doing tests, it should be clear that the test is either a Unit or an Integration test.

Structure of Tests

All files that will be used to run tests on the software will be named "*.spec.js" and will be stored under the "spec" folder in the root directory. These files will be recognized by Jasmine when the tests are being run.

Testing will be structured the way it is done on Jasmine.

The following is used in the Jasmine testing framework:

- Describe (Suite) - Describes your tests.
- It (Spec) - Tests that are run inside of a Suite.
- Expect - Defines what you expect from the server given your input.
- Matcher - Jasmine provides various matches to enable a rich testing suite, the matcher is used to compare the actual response from the server to the value in expected.
  Different matchers are explained by Jasmine here.

Test Reporter

Jasmine provides a console reporter to show the results of all the tests in a structured way once it is run. It also gives a summary of the tests that run and how many have passed and failed. If a test failed it also shows you a stack trace on where the test failed as well as what the actual value from the server response is.

An image of the console reporter can be seen below.

```
2055    69. POST http://localhost:3000/admin/login
2056  -       should return with statusCode 200
2057        ✓ should return with statusCode 200 (<1ms)
2058  -       should set content type = application/json
2059        ✓ should set content type = application/json (<1ms)
2060  -       should return a json object
2061          {
2062              "success": true,
2063              "message": "Incorrect username and/or password.",
2064              "data": {}
2065          }
2066        ✓ should return a json object
2067          {
2068              "success": true,
2069              "message": "Incorrect username and/or password.",
2070              "data": {}
2071          } (<1ms)
2072  Server shut down
2073
2074
2075  >> Done!
2076
2077
2078  Summary:
2079
2080  Passed
2081  Suites:  69 of 69
2082  Specs:   202 of 202
2083  Expects: 192 (0 failures)
2084  Finished in 0.513 seconds
2085
2086  The command "npm test" exited with 0.
2087
▶ 2088  store build cache
2103
2104
2105  Done. Your build exited with 0.
```

Example of a test

An example of a test is given below:

```
describe("A suite is just a function", function() {
  var a;
  it("and so is a spec", function() {
    a = true;
    expect(a).toBe(true);
  });
});
```

Tests should be in a uniform structure that is explained below:

- All tests belonging to the same area of the software should be encapsulated by a root description, specifying what is to be tested. It should specify which class is going to be tested and also if it is a Unit or Integration tests.
  I.e. Server Unit Testing
- In this description, there will be multiple describes testing different functions of the software. These functions all have a different endpoint, so the endpoint needs to be specified. It should also be specified whether the request was made via “GET” or “POST”.
  I.e. POST localhost:3000/admin/login
- In this inner description, there will be multiple “its”, which will test various aspects of the response received from the server. It should make sure that the instance of the server should be running to ensure that all other tests should pass.
  I.e. server.run should be called
- Inside of every it, the expected result should be typed out to make sure that anyone looking at the report after testing knows what the expected output was and if it was received or not (this will be known when a test passes).