

Testing Semester Project Report

Before the execution of this project we set ourselves learning goals that we wanted to achieve. We selected these learning goals in terms of what we thought we needed to improve upon and what we thought was most interesting. Here are the learning goals we set ourselves.

Learning goals:

- Get familiar with TDD by doing it.
- Use BDD to create efficient tests.
- Acquire proficiency in continuous integration using Travis CI.
- Learn how to test with Gherkin/Cucumber (test our Node.Js backend).
- Get experience with test automation using Cucumber and Selenium hand in hand.
- Learn how to test the databases (Neo4j and MongoDB) with JavaScript

TDD

One of the main focuses that we had in a very beginning of the Project was to get familiar with the TDD by doing it. We accomplished that learning goal partially. What we mean by that is in a beginning we have set up most of our tests using Cucumber and Mocha before creating the actual functionality of our application. However during the process, due to the time constraints we didn't follow this principles with everything what we created, only the main functionality got properly tested. We started off a little too slow in terms of the actual creation of functionality of the application and spent too much time learning about Cucumber and how to properly test databases in Node.Js. This eventually led us to have to rush in completing the necessary functionality of our application. All in all we believe that we still need to practice TDD and that in order to be more comfortable with it we should initially get comfortable with the test suites we decide to use.

BDD

Our second learning goal was to use BDD to create efficient tests. This is very similar to TDD but instead of solely focusing on testing the application BDD brings an edge of focusing on the behavior our application has to achieve, rather than testing the code itself. We put this as a separate learning goal as we believe that TDD can be accomplished without BDD thus making them two different things. This can be a gray zone as many believe it is very much the same thing.

Continuous Integration

Another learning goal that we had was about acquiring proficiency in continuous integration. In order to accomplish that goal we had to decide what tool we are going to use. The choice was fairly easy, we picked Travis CI because of many reasons. First of all we use Travis at work so we are already quite familiar with it. Second thing is that Travis is completely free if integrating an open source github project like ours.

We had Travis in school during this course so it was also very easy and quick for us to set everything up. Travis was ready to use right after a github repository was linked to it and after the `travis.yml` file was added to the root directory of our project.

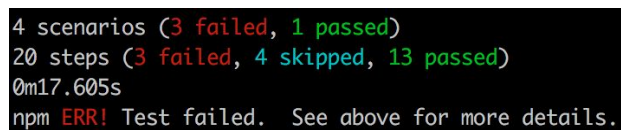
Cucumber Testing

We decided to use Cucumber as a testing framework due to our lack of knowledge about it and its relevance in the business world. What makes Cucumber interesting is that it starts off by having test cases written in a way that anyone could read and understand it and then goes deeper into the code. We found the way that Cucumber incorporates a non technical approach with the technical side of testing.

The way we used our Cucumber tests was to test the front-end. Although, testing the front-end also indirectly tests the backend as we expect data to be returned to the front-end from the back-end. The way we created our cucumber tests was by initially creating features that our application would contain. This is harder than it seems as you have to write them out clearly and only include the necessary steps. Even with a lot of attempts we still have not perfected our feature descriptions. From these feature descriptions we are able to create steps which are the actual test cases. These steps (test cases) contain the testing logic and the assertions to whether the tests are actually passing.

To test the front-end with Cucumber we used a testing module that would execute all of the front-end actions, from accessing the web application to clicking buttons and checking for returned values. To do this we used a module called 'Zombie' it is a headless JavaScript browser testing module. We used this instead of selenium as it is less heavy for JavaScript applications and consists just what we need for our test cases.

Although we attempted to carry out successful testing with Cucumber our final results are not perfect. This is due to a mix of poor testing and lack of time to complete all the features we initially wanted to complete.



```
4 scenarios (3 failed, 1 passed)
20 steps (3 failed, 4 skipped, 13 passed)
0m17.605s
npm ERR! Test failed. See above for more details.
```

Image: Cucumber test results

13 tests are passing, 3 failing and 4 are skipped. The reason for tests being skipped is that once a unit test in the test scenario fails the following unit tests within the scenario are skipped. So actually these tests could be passing but we never reach them. The tests that are failing are the tests that check whether our tables in the front-end are populated. The reason they fail is currently unknown but with a little extra time we will figure it out and fix it.

All in all one out of our 4 features is fully functional and that makes sense as we were not able to complete the actual functionality of 3 of the unit tests. But we have 7 unit tests that are not passing meaning that 4 of them are failing due to our test conditions being executed wrongly.

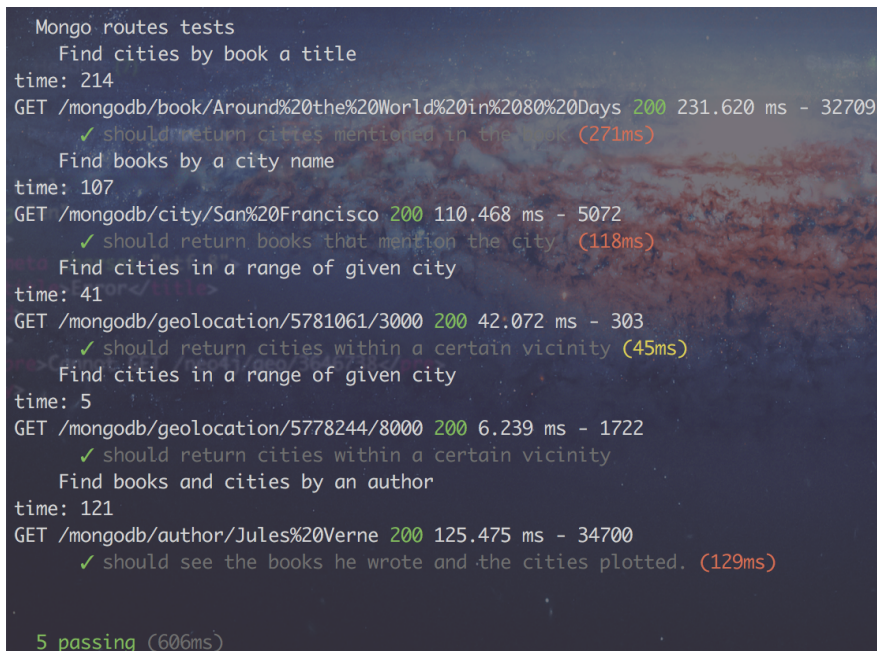
API Unit testing

In order to test the behaviour of our routes we created a couple of unit test using Mocha together with Chai. Since we got the whole functionality of the MongoDB calls to work, all its tests are passing as you can see on the figure below. However we didn't manage to make the last Neo4j query to work properly, that's why the test is failing.

Tests basically test if we get the positive status from the server, only if the data was returned from the databases. We also check the structure of the data and amount of objects that we expect to receive. In order to get the expected data from the database, for tests we set up local databases for both Neo4j and MongoDB.

Learn how to test the databases (Neo4j and MongoDB) with JavaScript

When testing the API we did not use any mocking tools to mock the database. The reason we did this is because mocking a database in a JavaScript based system is not reliable. This is due to the fact the JS does not consist of any concrete objects. This being said we could have created an embedded database, but we decided to go against that as an embedded database does not have the same behavior as a real Mongo or Neo database. Thus we created test databases that only contained test data but would look exactly as it would in a production database. This made it so that when we were to test in the final database we wouldn't be surprised with errors and compatibility problems.

A screenshot of a terminal window showing Mocha test results for MongoDB routes. The tests are organized into groups: 'Find cities by book a title', 'Find books by a city name', 'Find cities in a range of given city', and 'Find books and cities by an author'. Each group has a 'time' value and a 'GET' command. The results show that all tests passed, with a total of 5 passing tests in 606ms. The background of the terminal window features a dark, abstract, reddish-brown pattern.

```
Mongo routes tests
  Find cities by book a title
time: 214
GET /mongodb/book/Around%20the%20World%20in%2080%20Days 200 231.620 ms - 32709
  ✓ should return cities mentioned in the book (271ms)
  Find books by a city name
time: 107
GET /mongodb/city/San%20Francisco 200 110.468 ms - 5072
  ✓ should return books that mention the city (118ms)
  Find cities in a range of given city
time: 41
GET /mongodb/geolocation/5781061/3000 200 42.072 ms - 303
  ✓ should return cities within a certain vicinity (45ms)
  Find cities in a range of given city
time: 5
GET /mongodb/geolocation/5778244/8000 200 6.239 ms - 1722
  ✓ should return cities within a certain vicinity
  Find books and cities by an author
time: 121
GET /mongodb/author/Jules%20Verne 200 125.475 ms - 34700
  ✓ should see the books he wrote and the cities plotted. (129ms)

5 passing (606ms)
```

Image: Mocha (Chai) MongoDB routes tests

```

Neo4j routes tests
Find cities by book a title
Successfully connected to Neo4j database
GET /neo4j/book/Around%20the%20World%20in%2080%20Days 200 146.522 ms - 23248
✓ should return cities mentioned in the book (257ms)
Find books by a city name
GET /neo4j/city/San%20Francisco 200 43.474 ms - 11
✓ should return books that mention the city (60ms)
Find books and cities by an author
GET /neo4j/author/Jules%20Verne 200 20.745 ms - 11
✓ should see the books he wrote and the cities plotted
Find cities in a range of given city
GET /neo4j/geolocation/5781061 500 20.495 ms - 12
1) should return cities within a certain vicinity

action (req, res, next) {
  3 passing (427ms)
  1 failing
    name
    user) {
      1) Neo4j routes tests Find cities in a range of given city should return cities within a certain vicinity :
        Uncaught AssertionError: expected 500 to equal 200
        + expected - actual
        500
        -500
        000: It'd be better to use a cookie to "remember" this info,
        000: flash session
        000: return res.redirect(URL.format({
        000: pathname: '/users',

```

Image: Mocha (Chai) Neo4j routes tests