Test Course - Final Project

in Collaboration with the Database Course.

Group Members

Group Awesome

- Athinodoros Sgouromallis, cph-as293
- Ben Jones, cph-bj120
- Frederik Alexander Vygg Larsen, cph-fl64
- Rihards Pacevics, cph-rp84

Learning Goals

For this project we will be working interdisciplinary with the Database Course, and focusing on testing our Gutenberg Data Mining project.
We are aiming at using Test Driven Development for a full-stack javascript project.

- Unit Testing
- Mocking
- (Continuous) Integration Testing
- Test Driven Development
- Performance Testing
- Client Side Testing
- Code Coverage

We will be using the Agile Testing Quadrants to verify that our tests cover the most important areas.

The Project

We have made a full stack javascript project using nodejs and express, that we have also used for the database project. This required us to make an application that could find information about cities contained in the books on the Gutenberg Free Library. We have used two main database paradigms, MongoDB(Document Oriented) and the popular graph database Neo4J. Using this project as our basis for the testing project made sense in that we could hopefully deliver a good quality application for the database course, and get experience at trying to test all aspects of a web application. We might not have a fully tested application, but we will have managed to achieve the above learning goals through our work on this project.

Unit Testing

For our unit testing we have used the Mochajs framework with the ‘should’ and ‘expect’ assertion libraries. These provide a really flexible structure for writing easy to execute tests in a very easy to read format. Like JUnit they have the beforeEach and after hooks so that we can ensure we have a clean database to run each test on.

Mocking

During the project we haven’t used mocking as much as we thought we might. We tried to mock out the database connection using the mockgoose module, that creates an in-memory instance of MongoDB. It has a side effect, this being that every time the TravisCI build starts, it tries to download the entire source code for MongoDB. This makes TravisCI very slow to build, and seems like a waste of time when we know that TravisCI has an extremely reliable method for setting up database instances.

Also since every unit test should test the unit and not the methods that might generate the data that will be used as arguments to the methods of a unit mock objects (with js-mock) are used to mock the behaviour of other components so that the unit can be tested in a fully controlled environment.That way we can be sure that if the test fails the problem exists solely in the subject function.

Continuous Integration Testing

We have been using Travis CI as our continuous integration platform. It has excellent options for node js projects and offers many configurations that make it an easy platform to use. We have configured it to run instances of both our MongoDB and Neo4J, so that it runs our tests every time we push to github, and shows us a code coverage report with the exact differences between each commit, and the effect they have on coverage. If the build fails, we cannot push to the master branch, so we should always know that our production code is safe. Travis is a great tool, but when there are lots of people working on a project, and as the

project gets bigger, the build time increases. As we use the free version, only one build can take place at a time. For bigger projects we would definitely benefit from paying for a version that can support multiple builds.

On top of Travis a module that adds git hooks was used to prevent committing code to github that did not pass the tests. The module is called husky and we made the choice to add a pre-commit hook that will run the tests. If the test exists with 0 then it allows the code to be committed. This prevented pointless builds on travis that could possibly slow down the process.

Test Driven Development (TDD)

TDD was used in parts of the project and in some cases it proved to be very effective by removing freedom, and requesting discipline while coding. On the other hand since the programming language we choose (javaScript) was not one the team had gained experience throughout the semester it was not possible to predict the actual codes behavior so that we can write the correct tests beforehand so a lot of refactoring took place to make the tests work.

Performance Testing

For performance testing JMeter was used to monitor and collect data on our APIs response time. Since the project is made to find out what database is the best fit for a specific scenario (books and cities mentioned), JMeter was the perfect tool to generate the corresponding reports. JMeter will invoke each route 5 times with different arguments for both the mongo API and the neo4j API. The reports show that neo4j is considerably faster than the mongo one (average time : mongo 450ms, neo4j 84ms).

Although JMeter can be used to assert that the response is the expected one, Unittesting was chosen as a better fit for the task.

Client Side Testing

Selenium Webdriver was implemented for client side testing. The webdriver is installed through the node package manager, and has many out-of-the-box tools to use for automated tests. After setup, the webdriver directly starts a specified browser instance, and the webdriver also controls it.

Having automated tests for client side functionality is a great way to document that your code is working rather than having to click and take screenshots, etc., the selenium automation library does it all in a “synchronous”, easy to understand approach.
Through client side testing, bugs are automatically reduced, and the application design becomes much easier; knowing if things work as intended.

Logging the code coverage, which become available post-testing, there is the possibility to see any potential regress in the application, or discover any broken features, when adding new code to your application logic.

Using the angular framework, there were some challenges locating elements with the standard selenium webdriver. Solutions like implementing Protractor.js was considered, but that required an instance of the Selenium Server, and the idea was dismissed. A shame really because protractor.js is built exactly for the purpose of supporting automated tests in an angular environment.

It was discovered that running the tests came with a lot of difficulty, seeing that the tests threw exceptions, stopped the build, and failed. The error was due to a missing instance of the web application server. Once fixed it all ran smoothly. What wasn't considered after this was the cooperativeness of the Travis CI which obviously failed due to the same mistake. The immediate solution was to draw out the execution of the client side tests in the normal "npm test", and got a build command of its own instead. That way Travis was not attempting to run these tests.

Code Coverage

For code coverage we have been using the istanbul module. This module is very configurable and allows us to see detailed reports of our code coverage. We initially set a target that 80% of all functions in the application should be fully tested, as this would cover most of the important functionality of the code. Our coverage figure has fluctuated hugely during the project, showing that we have maybe not been very disciplined in our efforts to use test driven development. In our ambition to get some code up and running, as well as our running behind time, we have found that sticking to TDD is very hard!
Below is a screenshot of our coverage as it stands:

87.5% Statements 119/136 **62.5%** Branches 15/24 **62.5%** Functions 15/24 **87.31%** Lines 117/134

File		Statements		Branches		Functions		Lines	
server/		80.49%	33/41	0%	0/4	25%	1/4	79.49%	31/39
server/connector/		80%	8/10	100%	0/0	33.33%	1/3	80%	8/10
server/models/		100%	11/11	100%	0/0	100%	0/0	100%	11/11
server/modules/		100%	12/12	50%	2/4	100%	1/1	100%	12/12
server/routes/		88.71%	55/62	81.25%	13/16	75%	12/16	88.71%	55/62

Test Quadrants

We have spent most of our efforts on the technology facing quadrants from the agile testing model. The application itself is mostly focused on the api and the responses that we get from it. Designing and using the front end of the application became a low priority as we fell seriously behind with the database side. We have covered Q1 and Q4 most of all, with unit tests, component tests and performance testing.

Did we achieve our Learning Goals ?

While we haven't fully tested every aspect of the application, we have experimented with all the different testing techniques that we outlined in our goals, and have hopefully improved our abilities in finding what needs testing most of all, and how to test those things. We haven't always been organised in our working routine, and this is where we could make the most improvements in a future project. An example of this is our test data. We were quick to make data that we could use on the MongoDB part of the application, but did not make the same data for the neo4j side. So we haven't been able to see if we get the same results from the two database technologies.

Overall the results have been positive in that we now have the knowledge and methods to go out there and be able to properly test a full-stack application.