Testing Semester Project

In combination with Database

# Mocha, Chai, Knex.js, Travis CI and Istanbul

Node.js with Express Framework and MySQL Database

**Developed by Group Duo**
**Yoana Dandarova and Manish Shrestha**

**Date: 26.05.2017**

## 1. Introduction

This project is done in combination with the Database class using relational and graph database systems. It uses Node.js with Express framework as the backend.

For the testing aspect of the project, we are working only with MySQL database part. Now, let us introduce the testing frameworks implemented in the project.

- **Mocha**: Mocha is a JavaScript test framework running on node.js, featuring browser support, asynchronous testing, test coverage reports, and use of any assertion library.
- **Chai**: Chai is a assertion library. Role of a assertion library is to match result of a test to a description.
- **Knex**: Knex.js is a SQL query builder for Postgres, MSSQL, MySQL, MariaDB, SQLite3, and Oracle
- **Travis CI**: Travis CI is a hosted, distributed, continuous integration service used to build and test software projects hosted at GitHub.
- **Istanbul**: Istanbul is a code coverage analysis script for JavaScript which you run when executing your unit test.

## 2. Implementation

### 2.1 Mocha and Chai and Knex

*(We used different sources for mocha,chai and knex:*
*http://mherman.org/blog/2016/04/28/test-driven-development-with-node/#.WSfs8mh96M9*
*http://knexjs.org/)*

Inside the project  test.js is the file where our tests are specified. We created a separate file called
queries.js which functions talk to the mysql database. Both the functions that talk to the db and the routes are tested in the same file: test.js but there are created separate tests for db functions and routes. All routes are specified in app.js file.

We used Knex as SQL query builder for Node.js. Our mysql db is accessible from everywhere even from localhost , that's why if you download our project and run it you will have valid connection to our db even if you did not set up one yourself. We have created migration file for our db it is : `migrate_books.js`. We have set up and seed file to seed our db it is : `server/db/seeds/books.js`. The only problem we had in the whole project was to create beforeEach and AfterEach hook, we tried but we encountered problems. That's why if you pull our project please run first this command from the command prompt:

```
knex seed:run --env test
```

and  then run the tests : `npm test.`

**2.2 Travis CI:**

As we already know, Travis CI is a continuous integration service. So, when we write test for mocha, we can automate testing on commit to your repo on github. When something is committed on github in the repo, github sends signal to travis-ci.org to run the test.

For setting it up, we create a file on the root folder of the project called "`.travis.yml`" with its contents shown in the image below:

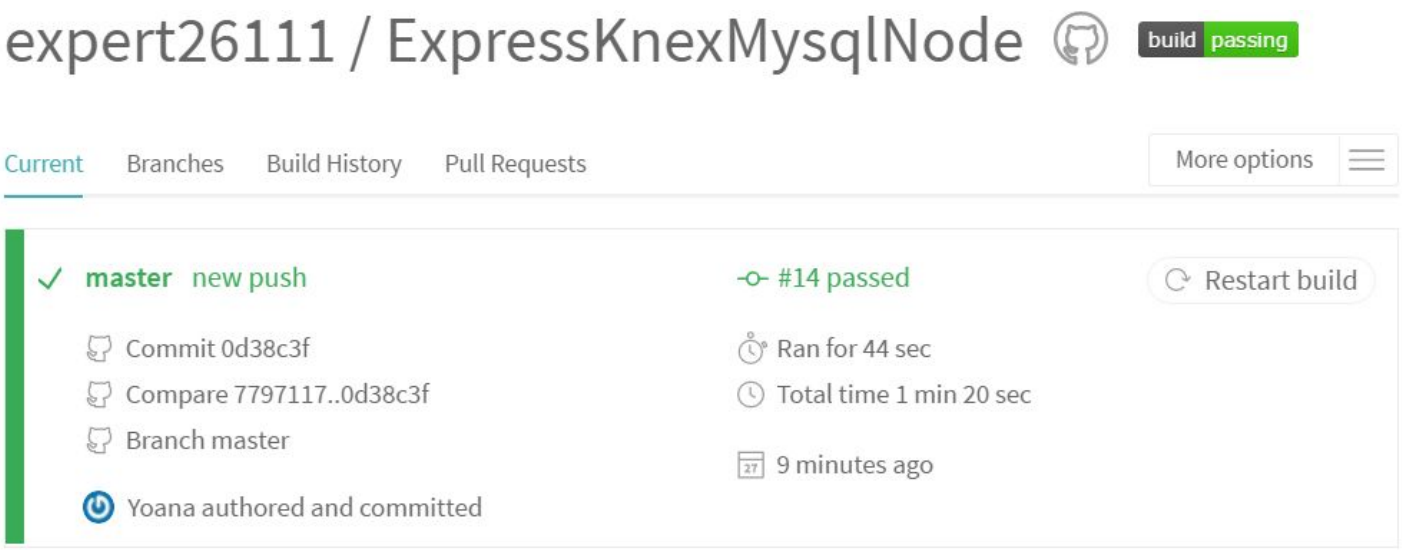
```
language: node_js
services:
    - mysql
node_js:
  - "iojs"
  - "7"
before_script: chmod 0777 ./node_modules/.bin/mocha
```

Here, we define the programming language and its version we're using.

Since our node.js app is working with connection my MySQL database, we need to start it on travis using "`services: - mysql`". And for mocha test to be executed by travis with proper permissions, the final line had to be added which is "`before_script: chmod 0777 ./node_modules/.bin/mocha`". And that was it for setting up Travis to test the project every time it was pushed to GitHub.

If the project's tests failed (or passed), it would notify all the contributors by email and on the webpage itself, you can see the which test failed with details on the "Job log" tab.

This helps the developers to go back and rectify the error and make the tests pass and merge with the build without breaking the whole project.




This is the latest build and Travis report from the project.

**2.3 Istanbul (Code Coverage):**

Code Coverage is a measurement of how many lines of your code are executed while the automated tests are running.

While we are using Mocha to do the actual test in our javascript project, Istanbul is a great additional library to check code coverage. Istanbul provides both CLI and Http interface to output the results of code coverage.

Definition of code coverage metrics provided by Istanbul are as follows:

- Statements: How many of the statements in you code are executed.
- Branches: Conditional statements create branches of code which may not be executed (e.g. if/else). This metric tells you how many of your branches have been executed.
- Functions: The proportion of the functions you have defined which have been called.
- Lines: The proportion of lines of code which have been executed.

In the project itself, we installed Istanbul package from npm and we added a couple of lines of scripts in the “package.json file” as shown below.

```
"scripts": {
  "test": "mocha test.js",
  "start": "app.js",
  "coverage": "istanbul cover node_modules/mocha/bin/_mocha test.js",
  "showcoverage": "start coverage/lcov-report/index.html"
```

After this, if we run “npm run coverage” in the command line, we get the output in the command line itself. “npm run showcoverage” opens the HTML output in a browser for us to inspect.

Below are the outputs from Istanbul.

## all files ExpressAppBooks/

**82.04%** Statements 137/167 **40%** Branches 4/10 **84.62%** Functions 44/52 **81.6%** Lines 133/163

| File | | Statements | | Branches | | Functions | | Lines | |
|---|---|---|---|---|---|---|---|---|---|
| app.js | | 81.01% | 64/79 | 40% | 4/10 | 66.67% | 16/24 | 80% | 60/75 |
| knexfile.js | | 100% | 1/1 | 100% | 0/0 | 100% | 0/0 | 100% | 1/1 |
| test.js | | 82.76% | 72/87 | 100% | 0/0 | 100% | 28/28 | 82.76% | 72/87 |

The **above** is the HTML output from Istanbul where we can go to individual files and check which lines/statements have been covered and which are not. It can also be interacted with here: Code Coverage for Testing.
The **below** is out from Istanbul in the command line interface (CLI), which shows the overall code coverage. Unlike the HTML output, we cannot inspect each files and lines from the CLI output.

```
================================ Coverage summary ================================
Statements   : 83.33% ( 150/180 )
Branches     : 41.67% ( 5/12 )
Functions    : 85.71% ( 48/56 )
Lines        : 82.95% ( 146/176 )
==================================================================================
```

### 3. Investigation on problems relating to Database Testing

- **Testing scope too large**: A tester needs to identify the test items in database testing otherwise he may not have a clear understanding of what he would test and what he would not test. Therefore, if you are not clear on the requirement, you may waste a lot of time testing uncritical objects in the database. Also, since a database is quite large, figuring out the  things that have to be tested and others that have to be left out can be a big challenge.
- **Scaled-down test database**: Normally testers are provided with a copy of the development database to test. That database only have little data, which is sufficient to

run the application. So there is a need to test the development, staging and as well as production database systems.

- **Changes in database structure**: This is one of the most common challenges in DB testing. There are times when the database structure gets changed while you design and execute tests. Hence, it is necessary that everybody involved are aware of the changes made to the database during testing. If and when the changes are made, you should analyze the impact of the changes and modify the test.
- **Complex test plans**: The database structure is normally complex and it has huge data, so there is a possibility that you are executing incomplete or same tests repeatedly. So there is a need to create a test plan and proceed accordingly and checking the progress regularly.
- **Good understanding of SQL**: To test a database, you should have a good knowledge of SQL queries and the required database management tools.
- **Multiple tests at the same time**: Another challenge in DB testing is that you run multiple tests at the same time. You should run one test at a time at least for the performance tests. You do not want your database performing multiple tasks and under-reporting performance.

### 4. Reflection

In reflection, we tested our routes that talked to the db and we used `chai.should` as assertion library successfully. Our routes and test cases are working correctly. Our investigation for testing against db concluded that we need to specify different environmental variables depending on which type of db we are using: production, development, test. We created migration file and seed file that worked properly. We learned that we need hooks to be executed before and after tests in “describe” or “it”.

So we conclude that our investigation was properly addressed. We did set up Travis-ci and it did run properly for us. During our development travis helped us, as we were sure that the test cases of each of us were passing correctly and were are not pulling something wrong.

We successfully used Istanbul code coverage tool and successfully created code coverage report. What the code coverage showed to us was that our “branches” are weak.

If you hit non existing route the server returns 404. But we have not created tests for this. And after seeing the report and all red areas we have better picture why. The code coverage report visualised for us pretty well what exactly has been tested and what not

and after getting familiar with terms as statements, branches, functions, lines we feel we understand better the point of the tool. So we conclude that our goals are achieved.

From our personal perspective, we are satisfied with our result, we learned a lot and are happy that our project as whole is working and resembles small working system. We are also proud that we managed in unfamiliar environment like setting up the whole project and troubleshooting all the problems that came along.