Unit testing node apps

Unit testing node and express

Our goals

- Learn about mocha testing framework
- Learn about testing patterns
- Test our rest server

What is unit testing

- Unit testing is code we write that runs our application code in a certain way and expect certain results
- Test will pass if the result match expectation otherwise it will fail
- Quality software we deliver means
  - Frequent updates
  - Low amount of bugs for our users
  - Our users experience low amount of crashes
  - Easy to scale and add more features
- One of the things quality software depends on is high percentage of test code coverage
- Programmer will write tests on his code if he feels comfortable with the testing technologies

CI / CD

- DO NOT WRITE TESTS IF YOU ARE NOT USING A TOOL TO RUN THEM AUTOMATICALLY
- Automation server is a server that will run tasks when building and deploying our app
- One of those tasks can be run our tests
- For example we can configure that server that every time we push our code to our SVN all the tests will run automatically and I will be alerted if a test failed
- We can also configure that server to deploy the code if the tests are passing - CI / CD
- Popular automation servers - Jenkins, TravisCI

Testing pure functions

- A pure function is a function that the result of that function depends only on the input arguments
- For example we have a function called whatLength which will get a string argument and return the length of the string
- Proper tests for this function might be:
  - Run the function with ‘hello’ make sure 5 is returned
  - Run the function with 10 make sure TypeError is thrown
  - We need to examine all the cases even the failed ones
- Those kind of functions are easy to test but in some cases our function depends on external things like database, networking, file system etc.

Testing non-pure functions

- When running our test even if we are testing non-pure the result should be the same in every run of the test
- To achieve this we have to make sure that the externals that the function relies on will have fixed results
- This will make sure the the side effects that can effect our function remain static and it turns our non pure function to pure with fixed externals
- Faking those externals are referred as mocking

Mocha - Testing Framework

- Mocha brings us tools to easily write our tests
- We can group tests together
- prepare the ground for running our tests
- Do cleanup when our tests complete
- Integrate mocha with all the popular assertion libraries
- Integrate mocha with various reporting tools
- Integrate mocha easily with your CI tool

Mocha - First Test - EX

- Let’s write our first test using mocha
- Install mocha using npm
  - npm install mocha —save-dev
- create the first test where you test that ‘hello world’ string length is 11 chars
- we will use the assert built in library
- we will use it to write our test
- Try and place a break point to debug the tests

Mocha - Async Test - EX

- We often will want to test function and classes that return result in the future in an async way
- In the testing function we can place an argument usually called done which is a function we have to call when our async result is back
- When grabbing that argument mocha will not go to the next test until we specifically call done
- In our previous test try to place the assertion in a setTimeout and call the done after the assertion completes

Mocha - Async Test - Promises - EX

- Instead of using the done argument, the test function can also return a Promise
- This allows us to use the async await for the test functions
- create a file with the text hello world
- import fs and using utils.promisify turn the readFile to return a promise
- Read the file using async await and assert the content is hello world

Mocha - describe - EX

- Using describe we can group tests together.
- We can also nest subgroups within groups
- Wrap the previous test with a subscribe

Mocha - hooks - EX

- Mocha hooks will run at certain times before or after running the tests
- They are used to prepare the ground for the test or for cleanup after the test
- The hooks are
  - before
  - after
  - beforeEach
  - afterEach
- The hooks can be async (works like async test functions)
- before running the test, create the file with hello world, then in the test read the file and verify the content

Mocha - mocking - EX

- When the function we are testing relies on external source like database or network we can fake that source and make it return a fixed result
- For this example let's create a service that will query our todo rest server
  - https://nztodo.herokuapp.com/api/task/?format=json
- The service will return the json data
- We have a function the uses this service to grab the data and count the number of todo tasks
- Let's create a test for that function and for this we will have to mock our service

sinon

- sinon allows us to fake functions, server requests
- Spy on functions
- mock functions and services
- It will allow us to mock and then easily restore the original
- use sinon.mock to mock the todo service and return a fake promise

Unit testing express app - EX

- To test our express server application we will have to launch our server
- In the tests send requests
- When the tests ends close our server
- Create an express app the sends an hello world, write a test to verify that when sending a request we are getting hello world

Unit testing express REST server

- When our server interacts with a database it is best to run the tests with a database as well
- We still want the data in the database to be the same in every test run
  - before each test clean the tables we will interact with
  - before each test populate the data with seed data for the test
- Create a REST server with an api to query the users
- Write a test for that api

Summary

- The more comfortable you will feel with writing tests the more you will do it
- The more you will do it, the better quality your software will be
- Make sure to make the testing functions pure by mocking
- Make sure to clean the database and plant seed data