Testing

Unit testing your node application

Lesson Plan

- why/what/how to test
- Mocha
- Testing models
- Testing Views
- Stubbing
- Spying
- Testing REST API

Lesson Summary

https://www.nerdeez.com/articles/node/express-testing

Why “not” to test

- I need to write test code which will take too long
- Tests maintenance
- I need to run the tests all the time

Why test

- once you are familiar with the tools it takes almost no time to write a test
- It’s faster to write a test then test in the browser every time
- Guarantees a more quality software product
- you don’t have to run all the tests we can combine them with CI tools to automatically run when you want to deploy your code
- Faster deployments

What to test in express

- models
- controllers - routes
- templates - view
    - common changes here so we need to be careful what to test
    - don’t test presence of css classes
    - don’t test appearance
    - test stuff like i have a form on that page with a text input with attribute name=”username”

Mocha

- mocha is a testing framework
- can be used for node testing as well as client side testing
- How it works
  - divide you tests to files with pattern: ***.test.js**
  - in each file divide tests to groups and subgroups
  - before each test run setup consistent data for each test
  - after each test if needed perform cleanup
- We install mocha with npm
  - npm install mocha --save-dev
- we run our tests
  - mocha
- to debug the test run with node debugger the file:
  - `node_modules/mocha/bin/_mocha`

Mocha - describe

- resided in a test file - *.test.js
- define a group of test
- can be nested in another describe for subgroups
- no limit to the amount of nesting
- a describe group is defined by a name and a function where the group tests will be located
- recommended not to use lambda functions with mocha

mocha - it

- single test
- each test has a description and a running function
- the function will have one or many assertions
- the test function can optionally have a done argument which is a function to end async tests
- you can also place async functions as the test function and use async/await
- you can use as assertion library the built in node assert library

node built in - assert

- assertion tests that will throw error if we are not equal to what we expect
- Will throw AssertionError
- two modes: recommended to use strict
- deepEqual - will iterate on all enumerable properties and traverse down their enumerable properties and so on, and will check equality
- equal - test if values are equal

befores, afters

- placed inside a describe block
- befores - are used to prepare the data in test or the model tested
- before - will run once before the first test in group
- beforeEach - will run before each test in the group
- afters - used for cleanups
- after - will run once after the last test of the block
- afterEach - will run after each test in the block

first test

- install mocha with npm
- create a test file
- create a group of tests with describe
- in that group place a single test with it
- in that test make a passing assertion
- create a beforeEach for the test
- run the test in debug mode
- run an async test

Testing service

- lets create a service with a function that checks if a mail is valid or not and return a true or false
- lets build a test for that service

Testing routes and views

- lets build an express app with 2 routes
  - /login
  - /welcome
- the login will contain a login form with a single input to enter mail, if mail is valid (using the previous service) we will direct to the welcome route
- if mail is invalid we will redirect to the login with a query param for error
- we will build a test for the get login and make sure we get a login form and an error label when there is error query param
- we will use cheerio to parse the html to jquery format
- we will make sure on good post we are directed to the welcome and on invalid post we are directed to the login

Stubbing our service

- we already tested the email service
- when we are testing the views we would usually want consistent data especially if our services are querying the database
- its common to mock the service and create response according to what we want to test in the view
- we can use sinon to do it
- sinon is a library to create spies, stubs and mocks
- you use sinon.fake to create a function
- you use sinon.replace to replace a function with a fake
- you call sinon.restore after each test
- lets create a stub for our email function

Testing our rest API

- Lets test the api for user we created on the postgres sequelize lesson
- we need to set environment variable DATABASE_URL to a test db
- every test we will clear all the user in the database and create new users for every test
- we will test the get users and create new users

Testing Summary

- the data for a test should not change between tests run
- the data should not depend on 3rd party changable data like a different server or database that can change between tests
- when testing the rest api you will need to make sure the db data stays the same in each test
- when not testing the services you will often need to fake them using sinon

EX

- add tests for 2 methods in your rest api
- add test for the login view