TASK

Testing and Refactoring Node.js

Visit our website

Introduction

WELCOME TO THE TESTING AND REFACTORING NODE.JS TASK!

In this task, you will learn to test and refactor your code using Mocha and Chai.

A note from the

HyperionDev Team

The reason that we need a structured way of approaching testing and refactoring our code is well described in the book "Refactoring JavaScript: Turning bad code into good code" by Evan Burchard. Consider the scenario he describes below. It is something all programmers can probably sympathise with and want to avoid.

"JAVASCRIPT JENGA

Programmers are tasked with making changes to the frontend. Each time they do, they make as small a change as possible, adding a bit of code and maybe some duplication.

There is no test suite to run, so in lieu of confidence, the tools they have available to ensure the quality of the codebase are 1) performing as many manual checks of critical functionality as is allowed by time and patience, and 2) hope. The codebase gets larger and a bit more unstable with every change.

Occasionally, a critical bug sneaks in, and the whole stack of blocks topples over, right in front of a user of the site. The only saving grace about this system is that version control allows the unstable (but mostly okay...probably?) previous version of the blocks to be set up without too much trouble. This is technical debt in action. This is JavaScript Jenga." - *By Evan Burchard (2017) page 95.*

NODE.JS TESTING FRAMEWORKS

Two very useful and popular frameworks used for testing Node.js projects are **Mocha** and **Chai**. Mocha is defined by its creators as "a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting while mapping uncaught exceptions to the correct test cases". Chai is often used with Mocha. Mocha provides the environment to run your tests. It organises and executes tests. Chai is an assertion library that runs in the testing environment and is used to test whether things are right or not. This will be easier to understand once you actually work with these tools. So let's get started.

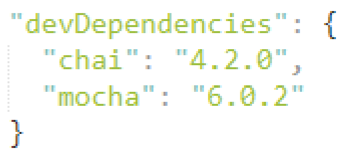
We are going to write some tests for the boilerplate code created when you create an app using the **Express generator**. This generator tool, express-generator, quickly creates an application skeleton.

- Step 1: Create a project using the express generator. To do this:
    - Navigate to where you want to create the application using your command line interface.
    - Type `npm install express-generator -g` to install the express generator.
    - Create the app by typing `express myapp` where 'myapp' is the name of the app you are creating.
    - Install the dependencies by typing `npm install`.
    - You can now test the resulting app by typing `npm start`.
    - Explore the code generated by this tool.

- Step 2: Install Mocha and Chai
  To do this, type the following in the command line interface (in the project directory you have created):
  `npm install --save-dev mocha chai`

  The switches `--save-dev` will make changes to the package.json file so that these tools are only used in the development environment.

  After using the `npm install` statement above in the command line interface, your package.json file has the following section added:



```
"devDependencies": {
  "chai": "4.2.0",
  "mocha": "6.0.2"
}
```

- Step 3: Update your scripts in package.json file
  Your scripts element of the package.json file should be updated to show how to run tests. We specify that Mocha will be used as shown below. Mocha will by default treat all JavaScript files in the 'test' directory of the app (see the next step) as tests. Therefore, adding this script to your package.json file allows you to run any tests you create in the test directory by typing "npm test" in the command line interface.

```
"scripts": {
  "start": "node ./bin/www",
  "test": "mocha"
},
```

- Step 4: Create a 'test' directory in your project folder
  Mocha will automatically treat all JavaScript files in this directory as tests. Create a file called 'users.test.js' in this directory that will test that the default functionality created for URL http://localhost:3000/users works as expected.

  The code that should be in the 'users.test.js' file is shown below. We are going to analyse this code to see how to write a unit test using Mocha and Chai in the steps that follow.

```
let expect  = require('chai').expect;
let request = require('request');

describe('Status and content', function() {
    describe ('Users page', function() {
        it('status', function(done){
            request('http://localhost:3000/users',
                    function(error, response, body) {
                expect(response.statusCode).to.equal(200);
                done();
            });
        });

        it('content', function(done) {
            request('http://localhost:3000/users',
                    function(error, response, body) {
                expect(body).to.equal('respond with a resource');
                done();
            });
        });
    });
  });
```

  Note: This code example uses the request package which allows you to test HTTP requests by making HTTP calls. To use this package first install it by typing `npm install request --save` in the command line interface.

- Step 5: Start coding the test by using the describe() function
  Mocha uses 2 important functions to create and organise tests: `describe()` and `it()`. The `describe()` function groups tests together and describes what you are testing. In the example above, we will test that the page /users of our app works as it should. Specifically, we are testing the status and

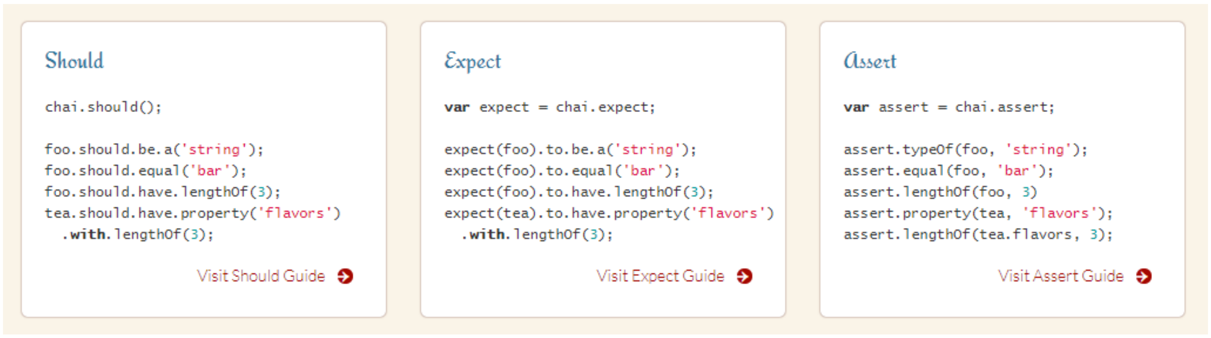
content of that page. Each `it()` function should test only one specific behaviour.

- Step 6: Use the `it()` function to describe the actual test
  The `it()` function should provide an easy to read description of the test as shown in the code example. The `it()` function also contains the code needed to execute the test.

- Step 7: Use Chai to write the actual code that will be used to execute the test.
  The `describe()` and `it()` functions are provided by Mocha. To perform the actual test, we need to use assertions to describe what we expect the output to be. We could use Node.js' built-in `assert()` module to do this, as shown in the example below.

```
let assert = require('assert');
assert(5 > 7);
```

The `assert()` module compares two values. If the two values aren't equal, an error is thrown and the code execution is terminated. Chai, as we mentioned earlier, provides an assertion library that is much richer. With Chai we can use **3 assertion styles**: *assert*, *should* or *expect*.


Should

```
chai.should();

foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.lengthOf(3);
tea.should.have.property('flavors')
  .with.lengthOf(3);
```

Visit Should Guide

Expect

```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(tea).to.have.property('flavors')
  .with.lengthOf(3);
```

Visit Expect Guide

Assert

```
var assert = chai.assert;

assert.typeOf(foo, 'string');
assert.equal(foo, 'bar');
assert.lengthOf(foo, 3)
assert.property(tea, 'flavors');
assert.lengthOf(tea.flavors, 3);
```

Visit Assert Guide

In the code example that we are analysing in this section, you will notice that we use the expect style. These statements are easy to read and it should be clear what we expect from our code.
`expect(body).to.equal('respond with a resource');`

The code below is from the official Chai documentation. It shows some other `expect` statements that could be used.

```
 let expect = require('chai').expect
  , foo = 'bar'
```

```
, beverages = { tea: [ 'chai', 'matcha', 'oolong' ] };

    expect(foo).to.be.a('string');
    expect(foo).to.equal('bar');
    expect(foo).to.have.lengthOf(3);
    expect(beverages).to.have.property('tea').with.lengthOf(3);
```

- Step 8: Run the test
  Run the test from the command line by:
  - Starting your app as usual by typing `npm start` in the command line.
  - In a second command line interface navigate to your project directory and type `npm test` in the command line.

See the sample code related to this task that gives an example of how Mocha and Chai can be used to test a simple module.

Some other testing frameworks include **Jasmine** and **Jest**. You learned to use Jest to test React apps in previous tasks.

REFACTORING CODE

Another important step to improving the quality of code is refactoring. Refactoring doesn't change the behaviour of the code and it doesn't remove errors. Rather, it has to do with restructuring code to make it more readable, maintainable and testable. Refactoring takes code that works and improves it.

JavaScript is an especially interesting programming language to refactor because there are so many ways and approaches to using JavaScript, and some work better than others.

Before we start refactoring code, take note of these two important points:
- Use version control. When you are changing code you always run the risk of 'breaking' it, so always make sure that you use version control so that if a change doesn't have the desired effect, you can go back to a previous working version of your code. If you need a refresher on using Git for version control, refer back to the tasks that cover this topic.
- Write tests before you refactor. Since we don't want to change the behaviour of code during the refactoring process, it is important to have tests (automated unit tests written using Mocha, Chai and Jest are

recommended) in place that can be used to see that your code still operates as expected after changes. Testing and refactoring go together.

Refactoring includes doing some of the following:

1. Make sure that your code is broken down into units (functions, classes, modules etc). We have already considered the benefits of modular design in previous tasks. You would have noticed that effectively testing code is made much simpler if your code is broken up into units. TDD ensures that modular design is used when coding. When refactoring it is important to change the structure of the code so that chunks of code are extracted to appropriate units such as functions or modules.

2. Be as brief as possible. If you can do something with fewer lines of code without detracting too much from the readability of your code, do so.

3. Make sure inputs to functions and output from functions are clear. Make sure that your functions return real, meaningful output. Make input into your functions as clear as possible. When you can use explicit input instead of `this` or non-local variables, do so.

4. Use meaningful names. Throughout your code, make sure that all variables, functions, modules, classes etc are given meaningful, descriptive names.

5. Remove unnecessary code. As we code we often create a lot of code that we can, and should, remove from the final version of our app.
   Delete:
   - Variables that are declared but never used.
   - Functions that are never called.
   - Code that you commented out because you thought you might need but you don't.
   - Statements used for debugging. Your code is likely to include things like Console.log() statements that you used to debug your system. For the final version of your app, get rid of these types of statements.

There are many more steps you can take to refactor your code to improve quality. An exhaustive description of refactoring techniques is beyond the scope of this bootcamp. See the recommended supplementary reading (**here**) for more information.

BEST PRACTICE

There are many ways of getting a program to work - sort of, some of the time, fingers crossed. Often a key factor that separates 'cowboy coders' from professional software developers is the application of best practice when coding.

A good resource when it comes to JavaScript best practice is found **here**. Some of the key recommendations detailed in this project are summarised below.

Coding style guidelines

When you write code, you should stick to a coding style. A coding style has to do with all aspects of how you write your code including things like where you place the curly braces and what naming conventions you use. Several bodies have written style guides for JavaScript. It is a good idea to follow a style guide from a recognised source. For example, Google has a style guide that you can access **here**. The best practice guide we referred to above provides some general guidelines including:

- Declare variables using the keyword const instead of var. More about this in the Task about ES6.
- Require modules at the beginning of files, not in functions.
- Use tools like ESLint and prettier to ensure that you are using good style practices in your code.

Project structure guidelines

We have mentioned the importance of having a clear and organised project structure for your full stack web application a few times already. Here is a summary of some of the key guidelines for a good project structure:

- Have separate **app.js** and **server.js** files in your Express app. The **app.js** file should be used to declare your API (see the task, Code Reuse) and the **server.js** file should contain the server network declaration. See examples of this **here**.
- Store configuration information in config files. configuration information, like URL or API key for any APIs we are using, should be stored in a config file. Create a directory called 'config' that will hold all the configuration information for your app. In this directory, create either a **.json** or a **.js** file to store configuration information.

Some more general guidelines

- Start all new projects with `npm init`.

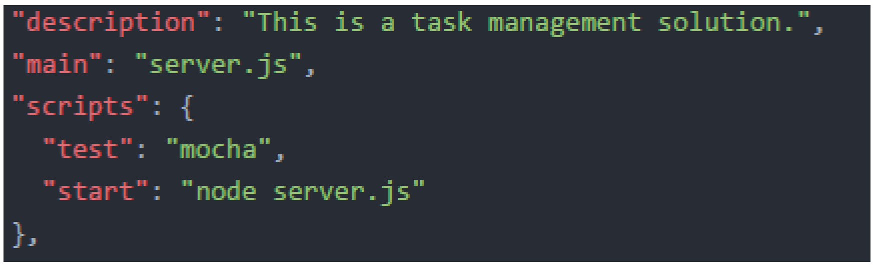
- Make sure all dependencies and correct versions of dependencies are stored in package.json by typing the following two commands into the command line interface:
  ```
  npm config set save=true
  npm config set save-exact=true
  ```

- Pay attention to security. An obvious concern for all web developers is ensuring that the applications they write are secure. The many issues related to ensuring the security of code is beyond the scope of this bootcamp. However, it is recommended that you use a tool like **Helmet** to provide at least some security for your app. Helmet is very easy to use. See the **instructions here**.

- Use async. Use promises and async-wait instead of callbacks wherever possible.

- Make sure your app automatically restarts. You can use tools like Nodemon to ensure this. Consider other **process managers recommended by Express**.

- Include scripts in your package.json file. Indicate which scripts to use when starting your app and when testing your app. See the example below:



```
"description": "This is a task management solution.",
"main": "server.js",
"scripts": {
  "test": "mocha",
  "start": "node server.js"
},
```

Compulsory Task

Follow these steps:

- Create a Node.js app that includes a module that calculates the insurance cost on a fleet of vehicles insured with pay-as-you-drive insurance. The app should have an endpoint that takes as input the kilometers driven for each vehicle. For each item, the app should calculate the insurance cost based on the kilometers driven, before finally summing all the values and returning it as the result.

  Use the table below to calculate insurance costs.

Kilometers driven	**Insurance cost**
0-20	R200
21-50	R200 + R1/km above 20km
51-100	R220 + R0,80/km above 50km
101+	R260 + R0,50/km above 100km

- Do not duplicate algorithms in your tests. Pick a set of inputs, calculate the correct answer by hand and check whether your module produces it. If you do the calculations in your test code you're not really testing anything - it would be like checking that a calculator works using the same calculator.

- Refactor your previous task. Submit a report that includes:
  - A summary of how you had to change your code to conform to the naming conventions specified in the Google Style guide **found here**.
  - Screenshots that show how you have changed your code to ensure that you are now using Helmet to improve the security of your code.
  - A description of how you have changed your package.json file and your config files to follow best practice guidelines. Also, include screenshots of portions of your code that show the changes made to your package.json and config files.
  - A description and screenshots of any other refactoring that you have done. Which variables/functions/classes, etc have you renamed and why? What

code have you deleted and why? Which functions have you changed and why?

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think the content of this task, or this course as a whole, can be improved or think we’ve done a good job?

Click here to share your thoughts anonymously.