TESTING NODE.JS APPS

A detailed guide for testing your Node.js applications the right way.

From the Engineers of RisingStack

TABLE OF CONTENTS

CHAPTER I. : GETTING NODE.JS TESTING AND TDD RIGHT

Making changes to a large codebase and making sure it works is a huge deal in software development. We've already talked about a few great features of Node.js testing before, and it is very important to emphasize how crucial it is to have your code tested before you release it to your users.

It can be tedious to have proper test coverage when you have to focus on pushing out all the new features, but think about your future self, would you like to work on code that's not tested properly? If not, read this guide on getting testing and TDD (test-driven development) right.

GETTING TDD RIGHT

When new people join a the project, you'll have to make sure that whenever they make a breaking change to the codebase, your tests will indicate it by failing. I have to admit that it is hard to determine what a breaking change is, but there is one thing that I've found really handy: TDD.

Remember, you dont write tests for yourself!
You write them for those who make changes later.

Test driven development is a methodology for writing the tests first for a given module and for the actual implementation afterward. If you write your tests before your application code, that saves you from the cognitive load of keeping all the implementation details in mind, for the time you have to write your tests. At least for me, these are the two best things in it. I always found it hard to remember all the nitty-gritty details about the code that I had to test later. With TDD I can focus more on the current step that I'm taking.

It consists of 3 steps:

* writing failing tests
* writing code that satisfies our tests
* and refactor.

It's that simple and I'd like to encourage you to give it a try. I'll guide you through the steps I usually take when I write a new module, and I'll also introduce you to advanced testing principles and tools that we use at RisingStack.

Step 1: Creating a New Module

This module will be responsible for creating and fetching users from our database, postgresql. For that, we're going to use knex.

First, let's create a new module:

```
npm init -y
```

And install the tools required for testing

```
npm install mocha chai --save-dev
```

Don't forget to add the following lines to the package json

```
"scripts": {
  "test": "mocha lib/**/**.spec.js"
},
```

Step 2: Creating the first test file

Let's create the first test file for our module:

```
'use strict'

const User = require('./User')
const expect = require('chai').expect

describe('User module', () => {
  describe('"up"', () => {
    it('should export a function', () => {
      expect(User.up).to.be.a('function')
    })
  })
})
```

I always like to create a function called "up" to that encapsulates the creation of the table. All I currently care about is to be able to call this function. So I expect it to be a function, let's run the tests now:

```
      AssertionError: expected undefined to be a function
at Context.it (lib/User.spec.js:9:29)
```

This is our first failing test, let's fix it.

```
'use strict'

function up () {
}

module.exports = {
  up
}
```

This is enough to satisfy the current requirements. We have so few code, that there is nothing to refactor just yet, let's write the next test. I want the `up` function to run asynchronously; I prefer Promises to callbacks, so I'm going to use that in my example.

Step 3: Creating a Node.js test case

What I want is the `up` function to return a Promise, let's create a test case:

```
it('should return a Promise', () => {
  const usersUpResult = User.up()
  expect(usersUpResult.then).to.be.a('Function')
  expect(usersUpResult.catch).to.be.a('Function')
})
```

It will fail again, to fix it we can just simply return a Promise from it.

```
function up () {
  return new Promise(function (resolve) {
    resolve()
  })
}
```

You see my point now. Always take a small step towards your goal with writing your tests and then write code that satisfies it. It is not only good for documenting your code, but when it's API changes for some reason in the future, the test will be clear about what is wrong. If someone changes the up function, use callbacks instead of promises - so our test will fail.

Advanced Testing

The next step is to actually create tables. For that, we will need knex installed.

```
npm install pg knex --save
```

For the next step I'm going to create a database called `nodejs_at_scale` with the following command in the terminal:

```
createdb nodejs_at_scale
```

And create a `database.js` file to have the connection to my database in a single place.

```
'use strict'

const createKnex = require('knex')

const knex = createKnex({
  client: 'pg',
  connection: 'postgres://@localhost:5432/nodejs_at_scale'
})

module.exports = knex
```

```
it('should create a table named "users"', () => {
  return User.up()
    .then(() => db.schema.hasTable('users'))
    .then((hasUsersTable) => expect(hasUsersTable).to.be.true)
})
```

```
'use strict'

const db = require('./database')

const tableName = 'users'

function up () {
  return db.schema.createTableIfNotExists(tableName, (table)
=> {
    table.increments()
    table.string('name')
    table.timestamps()
  })
}

module.exports = {
  up
}
```

The actual implementation

We could go more in-depth with expecting all of the fields on the table, but I'll leave that up to your imagination.

Now we are at the refactor stage, and you can already feel that this might not be the cleanest code we wrote so far. It can get a bit funky with huge promise chains so let's make it a little bit easier to deal with. We are great fans of generators and the co module here at RisingStack, we rely on it heavily at a day-to-day basis. Let's throw in some syntactic sugar.

```
npm install co-mocha --save-dev
```

Let's shake up that boring test script with our new module.

```
{
  "test": "mocha --require co-mocha lib/**/**.spec.js"
}
```

Now everything is in place let's refactor:

```
it('should create a table named "users"', function * () {
  yield User.up()
  const hasUsersTable = yield db.schema.hasTable('users')

  expect(hasUsersTable).to.be.true
})
```

Co-mocha allows us to write our `it` blocks as generator functions and use the yield keyword to suspend at Promises, more on this topic in our Node.js Async Best Practices article.

There is even one more thing that can make it less cluttered. There is a module called chai-as-promised.

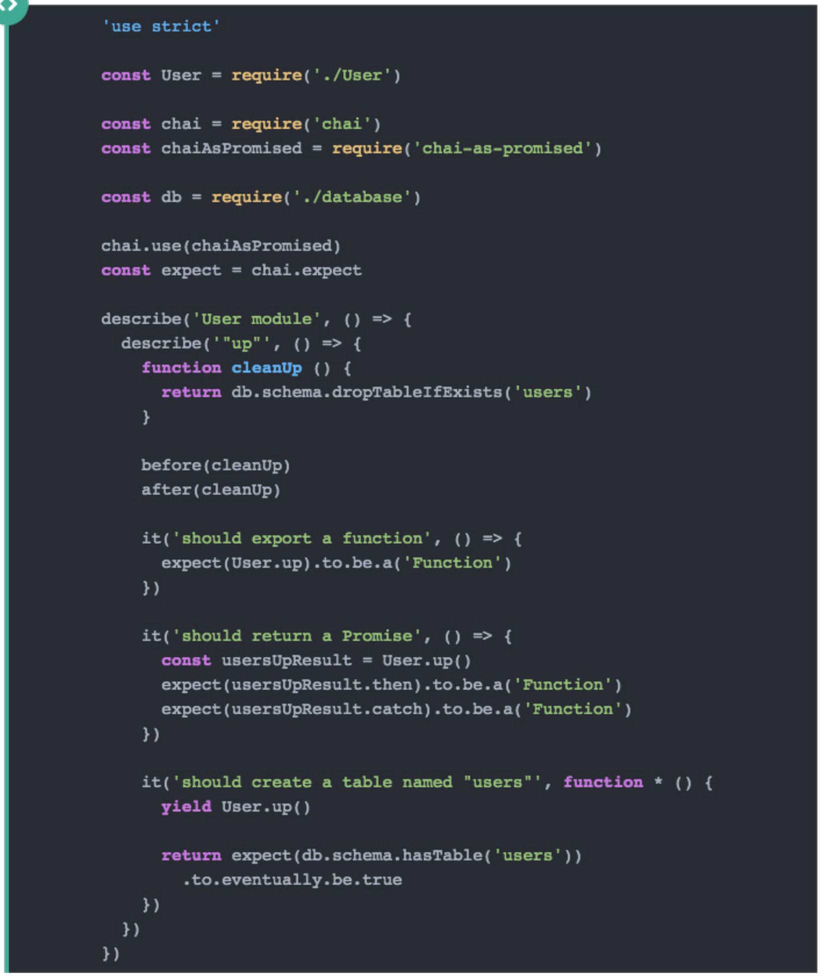
```
npm install chai-as-promised --save-dev
```

It extends the regular chai components with expectation about promises, as `db.schema.hasTable('users')` returns a promise we can refactor it to the following:

```
'use strict'

const User = require('./User')

const chai = require('chai')
const chaiAsPromised = require('chai-as-promised')

const db = require('./database')

chai.use(chaiAsPromised)
const expect = chai.expect

describe('User module', () => {
  describe('"up"', () => {
    // ...
    it('should create a table named "users"', function * () {
      yield User.up()

      return expect(db.schema.hasTable('users'))
        .to.eventually.be.true
    })
  })
})
```

If you look at the example above you'll see that we can use the `yield` keyword to extract the resolved value out of the promise, or you can return it (at the end of the function), that way mocha will do that for you. These are some nice patterns you can use in your codebase to have cleaner tests. Remember our goal is to express our intentions, pick whichever you feel closer to yours.

Let's clean up before and after our tests in a `before` and `after` block.



```
'use strict'

const User = require('./User')

const chai = require('chai')
const chaiAsPromised = require('chai-as-promised')

const db = require('./database')

chai.use(chaiAsPromised)
const expect = chai.expect

describe('User module', () => {
  describe('"up"', () => {
    function cleanUp () {
      return db.schema.dropTableIfExists('users')
    }

    before(cleanUp)
    after(cleanUp)

    it('should export a function', () => {
      expect(User.up).to.be.a('Function')
    })

    it('should return a Promise', () => {
      const usersUpResult = User.up()
      expect(usersUpResult.then).to.be.a('Function')
      expect(usersUpResult.catch).to.be.a('Function')
    })

    it('should create a table named "users"', function * () {
      yield User.up()

      return expect(db.schema.hasTable('users'))
        .to.eventually.be.true
    })
  })
})
```

This should be enough for the "up" function, let's continue with creating a fetch function for our User model.

After expecting the exported and the returned types, we can move on to the actual implementation. When I'm dealing with testing modules with a database, I usually create an extra describe block for those functions that need test data inserted. Within that extra describe block I can create a `beforeEach` block to insert data before each test. It is also important to create a `before` block for creating the table before testing.

```
describe('fetch', () => {
    it('should export a function', () => {
      it('should export a function', () => {
        expect(User.fetch).to.be.a('Function')
      })
      it('should return a Promise', () => {
        const usersFetchResult = User.fetch()
        expect(usersFetchResult.then).to.be.a('Function')
        expect(usersFetchResult.catch).to.be.a('Function')
      })

      describe('with inserted rows', () => {
        const testName = 'Peter'

        before(() => User.up())
        beforeEach(() =>
          Promise.all([
            db.insert({
              name: testName
            }).into('users'),
            db.insert({
              name: 'John'
            }).into('users')
          ])
        )

        it('should return the users by their name', () =>
          expect(
            User.fetch(testName)
              .then(_.map(
                _.omit(['id', 'created_at', 'updated_at'])))
          ).to.eventually.be.eql([{
            name: 'Peter'
          }])
        )
      })
    })
  })
```

Notice that I've used lodash to omit those fields that are dynamically added by the database and would be hard (or even impossible) to inspect on otherwise. We can also use Promises to extract the first value to inspect its keys with the following code:

```
it('should return users with timestamps and id', () =>
  expect(
    User.fetch(testName)
      .then((users) => users[0])
  ).to.eventually.have.keys('created_at', 'updated_at', 'id',
'name')
)
```

Testing Internal Functions

Let's move forward with testing some internals of our functions. When you're writing proper tests only the functionality of the current function should be tested. To achieve this, you have to ignore the external function calls. To solve this, there are some utility functions provided by a module called `sinon`.

The Sinon module allows us to do 3 things:

* **Stubbing**: means that the function that you stub, won't be called, instead you can provide an implementation. If you don't provide one, then it will be called as `function () {}` empty function).

* **Spying**: a function spy will be called with its original implementation, but you can make assertions about it.

* **Mocking**: is basically the same as stubbing but for objects not only functions

To demonstrate the use of spies, let's introduce a logger module into our codebase: winston. Guess what the code is doing by its the test over here:

```
it('should call winston if name is all lowercase', function *
() {
  sinon.stub(logger, 'info')
  yield User.fetch(testName.toLocaleLowerCase())

  expect(logger.info).to.have.been.calledWith('lowercase
parameter supplied')
  logger.info.restore()
})
```

And at last let's make this one pass too:

```
function fetch (name) {
  if (name === name.toLocaleLowerCase()) {
    logger.info('lowercase parameter supplied')
  }

  return db.select('*')
    .from('users')
    .where({ name })
}
```

This is great, our tests pass but let's check the output:

```
with inserted rows
info: lowercase parameter supplied
    ✓ should return users with timestamps and id
info: lowercase parameter supplied
    ✓ should return the users by their name
info: lowercase parameter supplied
    ✓ should call winston if name is all lowercase
```

The logger was called, we even verified it through our tests, but it is also visible in the test output. It is generally not a good thing to have your tests output cluttered with text like that. Let's clean that up, to do that we have to replace the spy with a stub, remember I've mentioned that stubs will not call the function that you apply them to.

```
it('should call winston if name is all lowercase', function *
() {
  sinon.stub(logger, 'info')
  yield User.fetch(testName.toLocaleLowerCase())

  expect(logger.info).to.have.been.calledWith('lowercase
parameter supplied')
  logger.info.restore()
})
```

This paradigm can also be applied if you don't want your functions to call the database, you can stub out all of the functions one by one on the db object like this:

```
it('should build the query properly', function * () {
  const fakeDb = {
    from: sinon.spy(function () {
      return this
    }),
    where: sinon.spy(function () {
      return Promise.resolve()
    })
  }

  sinon.stub(db, 'select', () => fakeDb)
  sinon.stub(logger, 'info')

  yield User.fetch(testName.toLocaleLowerCase())

  expect(db.select).to.have.been.calledOnce
  expect(fakeDb.from).to.have.been.calledOnce
  expect(fakeDb.where).to.have.been.calledOnce

  db.select.restore()
  logger.info.restore()
})
```

As you can see, it is already a bit tedious work to restore all of the stubs by hand at the end of every test case. For this problem, sinon has a nice solution called sandboxing. Sinon sandboxes allow you to define a sandbox at the beginning of the test and when you're done, you can restore all of the stubs and spies that you have on the sandbox. Check out how easy it is:

```
it('should build the query properly', function * () {
  const sandbox = sinon.sandbox.create()

  const fakeDb = {
    from: sandbox.spy(function () {
      return this
    }),
    where: sandbox.spy(function () {
      return Promise.resolve()
    })
  }

  sandbox.stub(db, 'select', () => fakeDb)
  sandbox.stub(logger, 'info')

  yield User.fetch(testName.toLocaleLowerCase())

  expect(db.select).to.have.been.calledOnce
  expect(fakeDb.from).to.have.been.calledOnce
  expect(fakeDb.where).to.have.been.calledOnce

  sandbox.restore()
})
```

To take it an additional step further you can move the sandbox creation in a `beforeEach` block:

```
beforeEach(function () {
  this.sandbox = sinon.sandbox.create()
})
afterEach(function () {
  this.sandbox.restore()
})
```

There is one last refactor to take on these tests, instead of stubbing each property on the fake object, we can use a mock instead. It makes our intentions a little bit clearer, and our code more compact. To mimic this chaining function call behavior in tests we can use the `returnsThis` method.

```
it('should build the query properly', function * () {
  const mock = sinon.mock(db)
  mock.expects('select').once().returnsThis()
  mock.expects('from').once().returnsThis()
  mock.expects('where').once().returns(Promise.resolve())

  yield User.fetch(testName.toLocaleLowerCase())

  mock.verify()
})
```

Preparing for Failures

These tests are great if everything goes according to plan, but sadly we also have to prepare for failures, the database can sometimes fail, so knex will throw an error. It is really hard to mimic this behavior properly, so I'm going to stub one of the functions and expect it to throw.

```
it('should log and rethrow database errors', function * () {
  this.sandbox.stub(logger, 'error')
  const mock = sinon.mock(db)
  mock.expects('select').once().returnsThis()
  mock.expects('from').once().returnsThis()
  mock.expects('where').once().returns(Promise.reject(new
Error('database has failed')))

  let err
  try {
    yield User.fetch(testName.toLocaleLowerCase())
  } catch (ex) {
    err = ex
  }
  mock.verify()

  expect(logger.error).to.have.been.calledOnce
  expect(logger.error).to.have.been.calledWith('database has
failed')
  expect(err.message).to.be.eql('database has failed')
})
```

With this pattern, you can test errors that appear in your applications, when possible try to avoid try-catch blocks as they are considered an anti-pattern. With a more functional approach it can be rewritten as the following:

```
it('should log and rethrow database errors', function * () {
  this.sandbox.stub(logger, 'error')
  const mock = sinon.mock(db)
  mock.expects('select').once().returnsThis()
  mock.expects('from').once().returnsThis()
  mock.expects('where').once().returns(Promise.reject(new
Error('database has failed')))

  return expect(User.fetch(testName.toLocaleLowerCase()))
    .to.be.rejectedWith('database has failed')
})
```

CONCLUSION

While this guide concludes most of what we do here at RisingStack on testing, there is a lot more to learn for us and for you from these projects' excellent documentation, links to them can be found below:

Test runner: Mocha. Assertions: Chai. Stubs/Mocks: Sinon
Utilities: Chai-As-Promised & Sinon-Chai

If you have made it this far, congratulations, you are now a 5-dan test-master in theory. Your last assignment is to go and fill your codebase with the knowledge you have learned and create greatly-documented test cases for your code in TDD style!

NODE.JS END-TO-END TESTING WITH NIGHTWATCH.JS

In this article, we are going to take a look at how you can do **end-to-end testing with Node.js, using** Nightwatch.js, a Node.js powered end-to-end testing framework.

In the previous chapter of Node.js at Scale, we discussed Node.js Testing and Getting TDD Right. If you did not read that article, or if you are unfamiliar with unit testing and TDD (test-driven development), I recommend checking that out before continuing with this article.

WHAT IS NODE.JS END-TO-END TESTING?

Before jumping into example codes and learning to implement end-to-end testing for a Node.js project, it's worth exploring what E-2-E tests really are.

First of all, end-to-end testing is part of the black-box testing toolbox. This means that as a test writer, you are examining functionality without any knowledge of internal implementation. So without seeing any source code.

Secondly, end-to-end testing can also be used as user acceptance testing, or UAT. UAT is the process of verifying that the solution actually works for the user. This process is not focusing on finding small typos, but issues that can crash the system, or make it dysfunctional for the user.

ENTER NIGHTWATCH.JS

Nightwatch.js enables you to write end-to-end tests in Node.js quickly and effortlessly that run against a Selenium/WebDriver server.

Nightwatch is shipped with the following features:

* a built-in test runner,
* can control the selenium server,
* support for hosted selenium providers, like BrowserStack or SauceLabs,
* CSS and Xpath selectors.

Installing Nightwatch

To run Nightwatch locally, we have to do a little bit of extra work - **we will need a standalone Selenium server locally, as well as a webdriver**, so we can use Chrome/Firefox to test our applications locally.

With these three tools, we are going to implement the flow this diagram shows below.

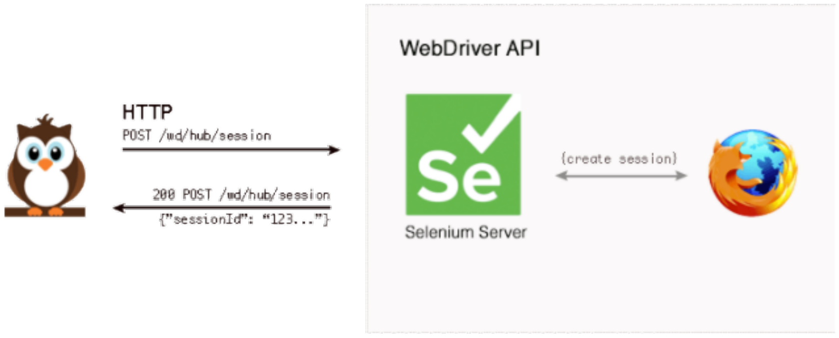


Photo credit: nightwatchjs.org

STEP 1: Add Nightwatch

You can add Nightwatch to your project simply by running `npm install nightwatch --save-dev`.

This places the Nightwatch executable in your `./node_modules/.bin` folder, so you don't have to install it globally.

STEP 2: Download Selenium

Selenium is a suite of tools to automate web browsers across many platforms.

Prerequisite: make sure you have JDK installed, with at least version 7. If you don't have it, you can grab it from here.

The Selenium server is a Java application which is used by Nightwatch to connect to various browsers. You can download the binary from here.

Once you have downloaded the JAR file, create a `bin` folder inside your project, and place it there. We will set up Nightwatch to use it, so you don't have to manually start the Selenium server.

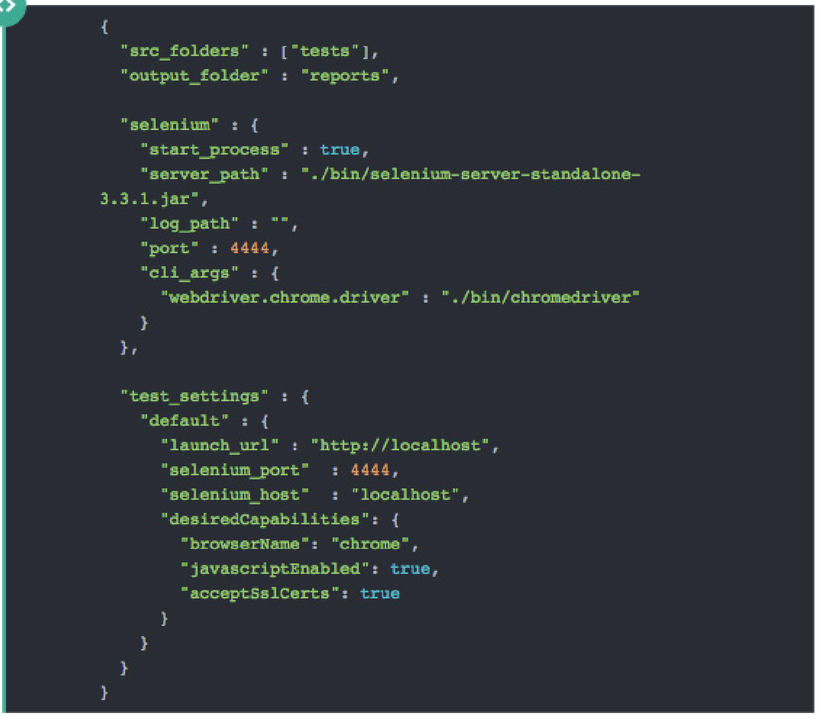
STEP 3: Download Chromedriver

ChromeDriver is a standalone server which implements the W3C WebDriver wire protocol for Chromium.

To grab the executable, head over to the downloads section, and place it to the same `bin` folder.

STEP 4: Configuring Nightwatch.js

The basic Nightwatch configuration happens through a `json` configuration file. Let's create a `nightwatch.json` file, and fill it with:



```
{
  "src_folders" : ["tests"],
  "output_folder" : "reports",

  "selenium" : {
    "start_process" : true,
    "server_path" : "./bin/selenium-server-standalone-
3.3.1.jar",
    "log_path" : "",
    "port" : 4444,
    "cli_args" : {
      "webdriver.chrome.driver" : "./bin/chromedriver"
    }
  },

  "test_settings" : {
    "default" : {
      "launch_url" : "http://localhost",
      "selenium_port"  : 4444,
      "selenium_host"  : "localhost",
      "desiredCapabilities": {
        "browserName": "chrome",
        "javascriptEnabled": true,
        "acceptSslCerts": true
      }
    }
  }
}
```

With this configuration file, we told Nightwatch where can it find the binary of the Selenium server and the Chromedriver, as well as the location of the tests we want to run.

Quick Recap

So far, we have installed Nightwatch, downloaded the standalone Selenium server, as well as the Chromedriver. With these steps, you have all the necessary tools to create end-to-end tests using Node.js and Selenium.

Writing your first Nightwatch Test

Let's add a new file in the `tests` folder, called `homepage.js`.

We are going to take the example from the Nightwatch getting started guide. Our test script will go to Google, search for Rembrandt, and check the Wikipedia page:

```
module.exports = {
  'Demo test Google' : function (client) {
    client
      .url('http://www.google.com')
      .waitForElementVisible('body', 1000)
      .assert.title('Google')
      .assert.visible('input[type=text]')
      .setValue('input[type=text]', 'rembrandt van rijn')
      .waitForElementVisible('button[name=btnG]', 1000)
      .click('button[name=btnG]')
      .pause(1000)
      .assert.containsText('ol#rso li:first-child',
        'Rembrandt - Wikipedia')
      .end()
  }
}
```

The only thing left to do is to run Nightwatch itself! For that, I recommend adding a new script into our `package.json`'s scripts section:

```
"scripts": {
  "test-e2e": "nightwatch"
}
```

The very last thing you have to do is to run the tests using this command:

```
npm run test-e2e
```

If everything goes well, your test will open up Chrome, Google & Wikipedia.

Nightwatch.js in Your Project

Now as you understood what end-to-end testing is, and how you can set up Nightwatch, it is time to start adding it to your project.

For that, you have to consider some aspects - but please note, that there are no silver bullets here. **Depending on your business needs, you may answer the following questions differently:**

Where should I run? On staging? On production? When don I build my containers? What are the test scenarios I want to test? When and who should write end-to-end tests?