Test-driven development: NoRestForTheWiccad: a RESTFul web API for witches and wizards

Introduction

The aim of this lab is for you to carry out a few cycles of the test-driven development workflow in the Javascript language. Along the way, you will learn a few web development tricks as well.

We have developed a simple RESTFul web API for you to use in this lab. It is a data storage and recall system with a magical theme.

Set up nodejs

You can work on this lab in the Coursera labs environment, or you can set up a local development environment on your own machine.

If you decide to set up the system on your local machine, we recommend that you use node.js v12 LTS.

Setting up to run locally on Windows and Mac

Please obtain and install node.js from the main site here: https://nodejs.org/en/

Setting up to run locally on GNU/Linux

If you are on Linux, we recommend that you use the binaries from node source. follow the instructions here to do that: https://github.com/nodesource/distributions

Verify you can do the work

You are now going to test your node install. Windows users: we recommend that you use Powershell to run these commands. Mac users: we recommend that you use the Mac Terminal to run the commands. GNU/ Linux users: use the terminal application you prefer.

In your terminal of choice, check the output of this command:

```
node --version
```

Hopefully you will see something like this:

```
v12.18.3
```

Working in the Coursera lab environment

The Coursera lab environment has already been configured for you with all the essentials to complete this activity. The lab is integrated with Visual Studio Code, an extremely popular code editor optimised for building and debugging modern web and cloud applications. The environment also gives you access to:

```
Node v14.15.5
NPM  v16.14.11
```

Start the lab environment application

It is simple to launch a lab exercise. You only need to click on the button "Start" below the activity title to enter a lab environment. Let's explore this lab activity. Go ahead and click on the "Start" button!

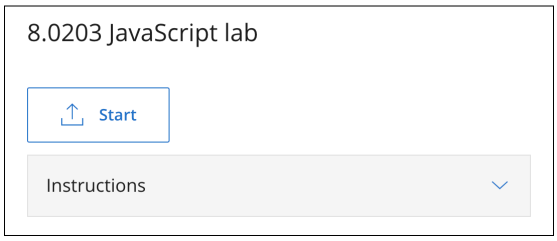

Figure 1: Coursera "Start" button

The Lab environment application

The lab environment application may take some time to load and you will be prompted with the following animation:

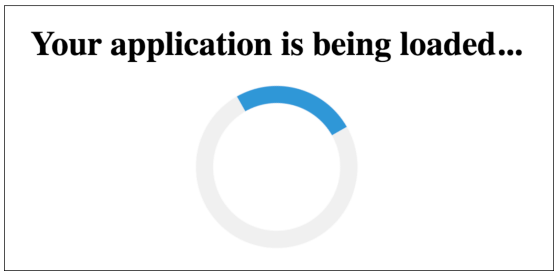

Figure 2: Coursera loading screen

Just wait a few seconds for your lab activity to start. It should not take longer than thirty seconds. If you experience any issues please load the activity again.

Once the application loads, you will see an instance of Visual Studio Code IDE as shown below:

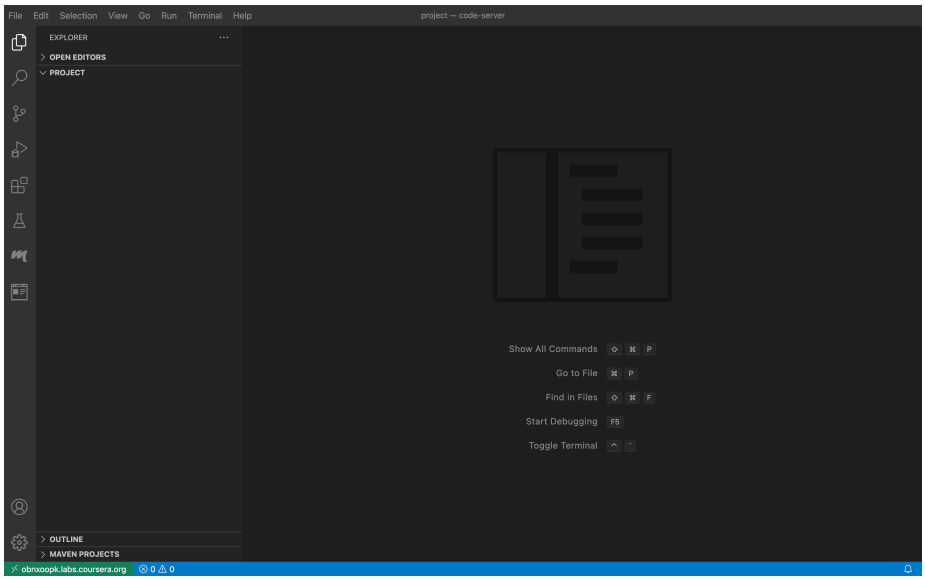

Figure 3: Visual Studio Code editor

The "Welcome" page illustrates most of the features to get you started with the environment. For now you can close the "Welcome" tab by clicking on the X icon next to the tab name as we will explore the main sections together. The "Welcome" page does not always appear so do not worry as it will not make any difference in the way the IDE works.

The Visual Studio User Interface

The Visual Studio User Interface is very easy to learn and it comes with a great selection of features. We will not explore in details all of the functionalities as it is not in the purpose of this course. We will instead look at the main sections required for you to get started with the labs.

VS Code comes with a simple and intuitive layout that maximises the space provided for the editor while leaving ample room to browse and access the full context of your folder or project.

The VS Code layout sections that you will require for this exercises are the Side Bar, the Editor Groups and the Panels.

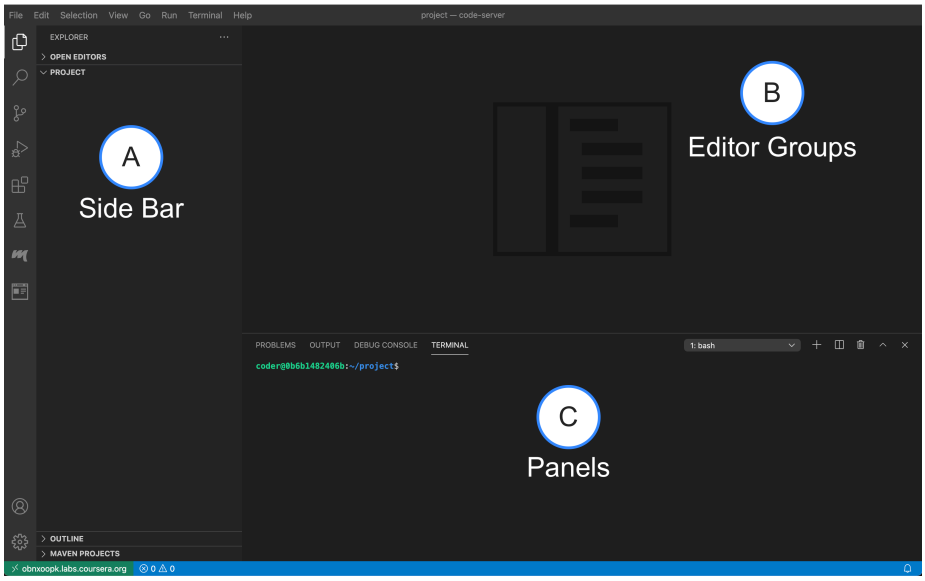

Figure 4: Visual Studio Code layout sections

- **Side Bar (A)** - Contains different views like the Explorer to assist you while working on your project. Mostly used to navigate your folders.
- **Editor Groups (B)** - The main area to edit your files. You can open as many editors as you like side by side vertically and horizontally.
- **Panels (C)** - You can display different panels below the editor region for output or debug information, errors and warnings, or an integrated terminal.This might not be visible when you open the IDE. The next section shows you how to open it.

Do not worry if your IDE configuration looks slightly different from the above picture. You can always drag the tabs around to adjust your layout configuration.

Note about the integrated terminal

The integrated terminal panel is normally hidden by default in Visual Studio Code and you need to manually enable it. Click the "View" tab on the top navigation bar and then click the "Terminal" tab to enable the panel window like shown below:

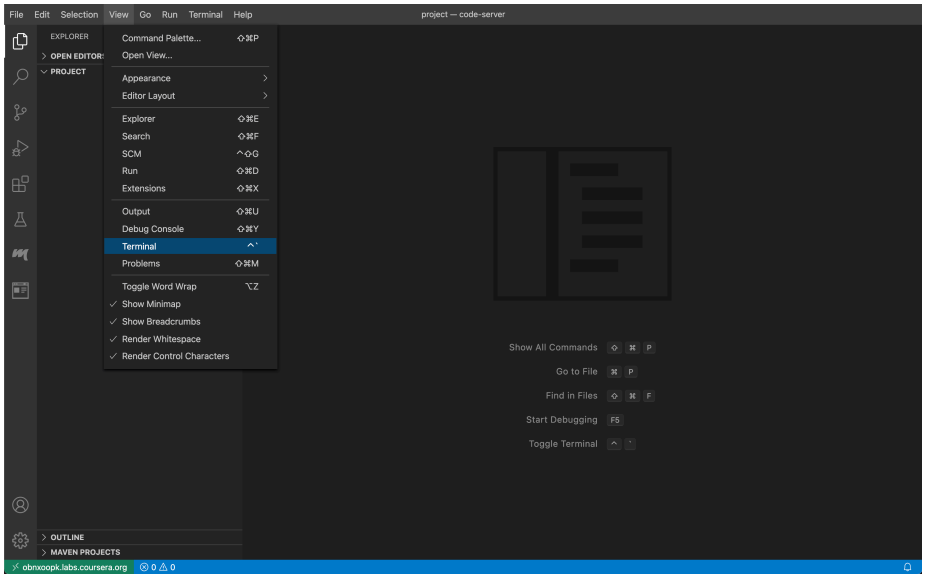

Figure 5: Visual Studio Code Terminal

Final touches

You are now set to work in the Coursera lab environment. Just few other things before you begin this exercise:

- The starter code is already included inside the environment: there is a folder called "norestforthewiccad" in the Side Bar section.
- You can create, edit, and delete files and folders in the Side Bar section.
- The Terminal exercise path is /home/coder/project/norestforthewiccad/, type **cd /home/coder/project/norestforthewiccad/** in the Terminal if you get lost.

Get the API to run

Now that you have the basic environment set up, it is time to try and run the web application you will be testing and improving.

Download the zip file for norestforthewiccad and unzip it.(only for local development)

You should have a folder called norestforthewiccad. In your terminal, cd to the folder where you unzipped norestforthewiccad. Run the following commands:

```
npm install
node index.js
```

Now point your web browser at http://localhost:3000, as instructed by the console output. You should see some JSON output from the server.

Accessing http://localhost:3000 from Coursera

In order to access the http://localhost:3000 from the Coursera development environment simply do the following:

- Click on the "Browser Preview" plugin as shown on the image below:

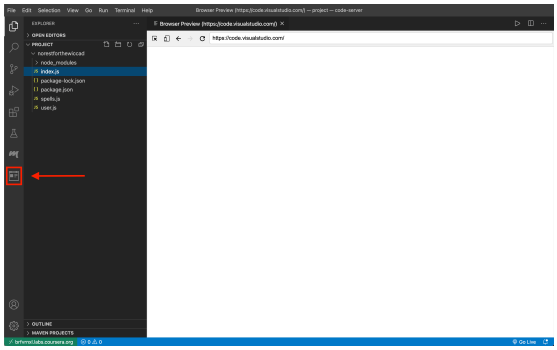

Figure 6: Visual Studio Code Browser Preview

- Now point the "Browser Preview" plugin at http://localhost:3000

Once you point your web browser at **https://localhost:3000** you should see some JSON output from the server.

Add mocha to the project

Now we need to add some packages to the set up - the mocha unit testing framework and the chai assertion framework. You can either open another Terminal tab or you can exit the running web server by pressing Control+C on your keyboard. This command adds the mocha, chai and chai-http modules to the project:

```
npm add mocha chai chai-http
```

Now add this to packages.json's scripts section to allow us to easily run the tests:

“scripts”: { “test”: “node_modules/.bin/mocha” }

Now run the tests:

```
npm test
```

You will see some output a bit like this:

```
> norestforthewiccad@1.0.0 test ./norestforthewiccad
> mocha -w

Error: No test files found: "test"
npm ERR! Test failed.  See above for more details.
```

You are now ready to create the tests!

Optional extra - hot code reloading

You can enable hot code reloading using nodemon. Hot code reloading means your app will automatically restart every time you make an edit to the code. In the top level folder, where the node_modules folder is, type this:

```
npm install nodemon
./node_modules/nodemon/bin/nodemon.js  index.js
```

Now your app will reload every time you edit a file. If you want to get really fancy, you can set this command up in your packages.json file in the scripts section:

```
"scripts": {
 ....
    "app": "./node_modules/nodemon/bin/nodemon.js  index.js"
},
```

Now you can run the app as follows:

```
npm run app
```

If you edit any of the app files, it should automatically reload.

Backup your work in the Coursera development environment

It is always a good idea to backup your work and indeed a best practice in the cycles of the test-driven development workflow. In order to backup your work inside the Coursera development environment do the following:

- Run this command inside the terminal: /home/coder/coursera/backup.sh
- This will create a backup.zip file inside your root directory /home/coder/project

Note: This feature only works with the old lab experience. If you wish to download a copy of your work you will have to first switch your lab to the old experience. You can do so by clicking on the "switch back to the old lab experience" link inside the "Help" menu button located on the top right corner as shown on the image below:

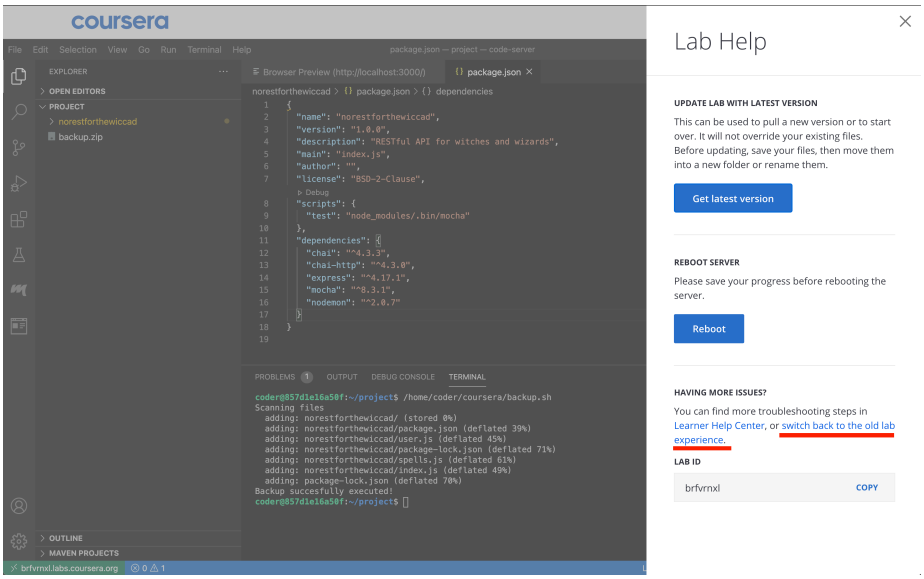

Figure 7: Coursera Help Tab

Do not worry as the latest lab experience will be restored in future accesses. Once the lab reboots, simply drag and drop the generated "backup.zip" file from the lab environment to your desktop to save a local copy of your work.

Once you have finished with the backup process, please remember to reboot the lab in order to restore the latest lab experience.

Writing some unit tests

Now we should have our development environment ready and we have the web application running. We are ready to write our first unit test.

Create a simple test

Make a folder in the norestforthewiccad folder called 'test'.

In that folder, place a file called tests.js, with the following content:

```
var assert = require('assert');
describe('TestSetTestingMultipleModules', function () {
  describe('TestSubSetTestingASingleModule', function () {
    it('should compute basic maths such as 2 + 2 passer', function () {
      assert.equal(2+2, 4);
    });
  });
});
```

Rerun the test command:

```
npm test
```

Now you should see some output which ends like this:

```
  TestSetTestingMultipleModules
    TestSubSetTestingASingleModule
       should compute basic maths such as 2 + 2 passer


  1 passing (3ms)
```

Excellent - you are ready to write some real tests!

Test the http://localhost:3000/ '/' route with mocha, chai and chai-http

Replace the code in test/tests.js with the following code:

```
var chai = require('chai');
var chaiHttp = require('chai-http');
var assert = require('assert');

chai.use(chaiHttp);

describe('Test top level / route', function() {
  it('it should have a 200 status code', function (done) {
    chai.request('http://localhost:3000') // the top level web address
        .get('/') // the route to add to the top level address
        .end((err, res) => { // what to do once the request returns
          assert.equal(res.status, 200); // check we have the 200 OK HTTP code
```

```
          done(); // finish up
        });
  });
});
```

Make sure you have your app running, then run the test with:

```
npm test
```

You should see output like this:

```
> norestforthewiccad@1.0.0 test /norestforthewiccad
> mocha


Test top level / route
 it should have a 200 status code


1 passing (29ms)
```

Write a failing test

Remember that the first step of the test-driven lifecycle is to write a failing test? Well it is time to do that. Put this into your tests.js file, below the other 'it' function call:

```
it('it should send the right message', (done) => {
chai.request('http://localhost:3000')
    .get('/')
    .end((err, res) => {
      let data = JSON.parse(res.text);
      assert.equal(data.message, 'Welcome to the norestforthewiccad API'
);
      done();
    });
});
```

If you are like me, you find it frustrating when tutorials keep giving you code fragments. So here is the complete content for test/tests.js:

```
var chai = require('chai');
var chaiHttp = require('chai-http');
var assert = require('assert');

chai.use(chaiHttp);

describe('Test top level / route', function() {
  it('it should have a 200 status code', function (done) {
```

```
    chai.request('http://localhost:3000') // the top level web address
        .get('/') // the route to add to the top level address
        .end((err, res) => { // what to do once the request returns
          assert.equal(res.status, 200); // check we have the 200 OK HTTP code
          done(); // finish up
        });
  });
  it('it should send the right message', (done) => {
    chai.request('http://localhost:3000')
       .get('/')
       .end((err, res) => {
          let data = JSON.parse(res.text);
          assert.equal(data.message, 'Welcome to the norestforthewiccad API');
         done();
       });
  });
});
```

Can you change the code in index.js so that it passes the test?

Work on the /spells route

Does the /spells route exist?

Ok now we have a passing test, time to write another failing test. Here is your test:

```
it('it should have a spells route', (done) => {
chai.request('http://localhost:3000')
    .get('/spells')
    .end((err, res) => {
      assert.equal(res.status, 200);
      done();
    });
});
```

Can you fix that one? You have 2 minutes. As a clue, look in index.js to see how the spells are routed. A 404 code means the address does not exist.

Does /spells provide any data?

Now over to you - /spells should output some data a bit like this:

```
[
  {
     id: 1001,
     name: "Rabbit foot positivity",
     ingredients: [
```

```
        {name:"Foot of rabbit"},
        {name:"Juice of beetle"}],
     result: "Good luck"
  },
  {
     id:1002,
     name: "Fox exeunta",
     ingredients: [
        {name:"Foul of lion"},
        {name:"Spirit of hobo"}],
     result: "Fox removed",
  },
....
]
```

Can you think of a test that would test one aspect of that output?

Can you adapt the code so it passes your test?

The /spells/:id path

Next up, the spells/:id path. If you request http://localhost:3000/spells/1002, you should receive a spell with that id.

Can you think of a simple test for that? Can you fix it?

Extended work

If you want to carry on with this work, can you fix the other routes? POST /spells (add a new spell) and PUT /spells/:id (update a spell)?