**Table of Contents**

[Table of Contents](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc0)

[Creating Automated Unit Tests Using JEST](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc1)

[Step 0 -  Set up the environment](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc2)

[Step 1 - Write a Simple Test](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc3)

[Step 2 - Write a Simple Test that Deliberately Fails](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc4)

[Step 3 - Write a Simple Test for our dataAPI end point](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc5)

[Step 4 - Test Driven Development](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc6)

[Step 4\_1 - Write the Test](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc7)

[Step 4\_2 - Run the Test](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc8)

[Step 4\_3 - Create the Code](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc9)

[Step 4\_4 - Run the Test Again](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc10)

[Step 4\_5 - Add More Tests](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc11)

[Step 5 - Running Tests Using GitHub Actions](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc13)

[Step 6 - Disable All GitHub Actions](http://cege0052.cege.ucl.ac.uk/cege0043tutorials/tutorials/week6/practical6-step2.html#toc14)

**Creating Automated Unit Tests Using JEST**

JEST is a popular JavaScript testing framework  Unit tests are tests that work with individual functions or very small groups of functions - i.e. the bottom level of testing code.

Will make use of  test-driven development - in other words, we will:

* write a test
* run the test (which will fail)
* write the required code
* re-run the test until the code passes the test (next week)

In this example, we will write and deploy the test code and code to check that we are able to call an endpoint in dataAPI.js

NB: Commit frequently throughtout this practical, but DO NOT PUSH your work until the final step of the practical.  Pushing your code triggers JEST tests via GitHub Actions and we don't want to run those until we've finalized all the tests on our server.

NB: JEST tests automatically start the API server so make sure that you **stop the server** (i.e. are not already running the server via NODE or PM2) before you run any tests

**Step 0 -  Set up the environment**

We'll be working in the API environment for this practical.

1. Pull/merge all the previous API code to the main branch

2. Create a branch called *jest*

3. Clone the code to your server

4. Add JEST as the test enviornment in package.json scripts, as follows

npm install jest --save

**Step 1 - Write a Simple Test**

1. On your server, make a directory called *js* under the root directory of the API repository  - we will use the js directory to store our test files

cd /home/<<your CS username>>/code/<<your github API repository name>>

mkdir js

2. Create a new file called *simpleJest.test.js*

3.  Add the following text to the file - this is a simple, hard coded test that we can use to demonstrate that JEST is working as expected

"use strict";

test("string matches",() => {

// test for match the string - Success

let string1 = "First test for cege0043"

expect(string1).toMatch(/cege0043/);

// test for not match the string - this test will also

// pass as not.toMatch is correct in this case

expect(string1).not.toMatch(/abc/)

})

4. Upload the file into the *jest* directory

NB:  all JEST test files should be named *<<filename>>.test.js* and will be placed in the *jest* directory

5. Run the test by typing the following commands

cd /home/<<your CS username>>/code/<<your API repository name/

./node\_modules/jest/bin/jest.js --verbose --detectOpenHandles ./js/simpleJest.test.js

5. You should get results as follows

A screen shot of a computer code

Description automatically generated

Note that we have two important statistics here

- the total number of TEST SUITES that have been run  - this is the number of *test files* that have been used and whether all the tests in each file passed or not

- the total number of TESTS that have been run and whether they passed

(these numbers will vary depending on whether we run each test file separately as we did here, or whether we run all the test files at once)

**Step 2 - Write a Simple Test that Deliberately Fails**

1. Add another test to simpleJest.test.js as follows.  This test is a test that will  fail

test("string does not match",() => {

// test for match the string - Success

let string2 = "Second test for cege0043"

expect(string2).toMatch(/second/);

})

2.  Run the test suite using the command above

3. You will see a result similar to - but not identical to - the image below.

A computer screen shot of a program

Description automatically generated

4. Take your time to understand the test results - in partiuclar you can see that JEST indicates exactly which line of the testing script failed.

 - Note that in this case we still have one test suite (i.e. we're running one test file) - which is flagged as FAILED as at least one of the tests in that file failed

- We also have two tests in total, one which passed, the other which failed

5.  Adapt the above testing code to fix the problem so that the test passes.

**Step 3 - Write a Simple Test for our dataAPI end point**

**Note**: Before running this test you should have the Data API running.   You can use the NODE command for this, and then open a separate command line window to run the test

So far, we've only run fake tests.  JEST (and unit testing in general) is most useful as it can automatically test your code every time someone makes a change, to make sure that their change hasn't broken existing code.  To acheive this, we need to have a way to run JEST tests on our code.

Fortunately, JEST has the ability to start a web server, run some tests, and then stop the web server.  This is achieved via the *[supertest](https://www.npmjs.com/package/supertest)* package.

1. On your server, install the following package:

npm install supertest --save

1. Create a new file called dataAPI.test.js

2. Add the following code into the file - this references the required packages for the test

'use strict';

const supertest = require('supertest');

const request = supertest('http://localhost:4480');

3. Now write the code to run the test - make sure the string matches exactly what you've put in the dataAPI.js file

test('returns a welcome message', async () => {

const response = await request.get('/');

expect(response.status).toBe(200);

expect(response.text).toEqual("hello world from the Data API on port: 4480");

});

4. Run the test as follows

./node\_modules/jest/bin/jest.js --verbose --detectOpenHandles ./js/dataAPI.test.js

**Step 4 - Test Driven Development**

Note: Before running this test you should have the Data API running.   You can use the NODE command for this (as you will also be changing the code in dataAPI.js), and then open a separate command line window to run the test

For this part of the practical, we will use a test driven development approch to  meet the following functional requirement:

*Take the text submitted by the user and reverse it*

We'll be developing an end point on dataAPI.js to  meet this functional requirement

**Step 4\_1 - Write the Test**

1. Add a new test called *reverses the text* to the dataAPI.test.js file.  This test should:

* Call an API end point /*reverseText?texttoreverse=xxxxx xxxxx*
* Check that the response status of the return is 200
* Check that the text has actually been reversed correctly

Where *xxxxx xxxxx* is any text that you want to use for testing.

2. Save the text, upload your code to the server, commit but don't push

**Step 4\_2 - Run the Test**

3.  Run the test as above  - the test should fail

**Step 4\_3 - Create the Code**

4. We now need to add the code into dataAPI.js.  Note that this code sample is **deliberately incorrect** so that you can use a test driven development approach to update the code until the above test passes.  To fix the bugs, you may need to think about:

* how endpoints work
* [how arrays work in JavaScript](https://www.w3schools.com/js/js_arrays.asp)
* how we are sending status values to the browser with the result from any request to the API

/\*\*

\* Route serving basic test message

\* @name /reversetext

\* @function

\* @memberof module:dataAPI

\* @inner

\* @param {string} - the string to be reversed

\*/

dataAPI.get('/reversetext',function (req,res) {

//note, this is the buggy version for the students to fix

let originalText = req.query.texttoreverse;

console.log(originalText);

// convert it to an array - we can use the built in Array.from for this

let textArray = Array.from(originalText);

console.log(textArray.length);

let reversedText = "";

// loop over the array backwards

// here i > 0 is the condition that, while true, will keep the loop

// going - as soon as i > 0 returns false, the loop stops

for (let i = textArray.length; i > 0; i--){

console.log(i);

reversedText = reversedText + textArray[i];

}

// finally send the result back to the browser

res.send(reversedText);

});

5.  Upload the code, and commit but do not push the code

**Step 4\_4 - Run the Test Again**

6. Run the test again - did your code pass?

7. Fix any bugs, re-running the test each time until the code passes the test.

NB:  Challenge your skills here - try and debug the code by looking at the code and working out what is going wrong.    The test runner will show the console.log messages that are in the code - use these to help you out!

(You could just test your code from a browser of course, but remember the point here is that these tests run every time SOMEONE makes a change to any part of the code - you might not even be on the project when someone else changes the code)

**Step 4\_5 - Add More Tests**

In testing it isn't sufficient to just run one small test and then stop - you need to try and think of all possible combinations.

8. Extend the *reverses the text*  test to include the following situations:

* The text contains spaces in the middle
* The text contains a space at the beginning
* The text contains a space at the end
* The text contains a % sign
* The text contains a $ sign
* The text contains an &
* The text contains a \*

You may find [URL Encoding](https://www.w3schools.com/tags/ref_urlencode.asp) relevant for these tests

9. Don't forget to commit as soon as you get each test working

**Step 5 - Running Tests Using GitHub Actions**

Now that the tests work locally, you can use GitHub Actions to test automatically.   We already have a simple testing file call js.yml, in the .github/workflows directory.  You first need to edit this file so that the API is started before the tests are run.

1. Find the js.yml file, which is in the following directory (note that there is a . before gitub)

 /home/<<your cs username >>/code/<<your github api repository name>>/.github/workflows

2. In a text editor, edit the file so that the final text looks as follows:

name: Javascript Unit Tests

on:

push:

branches:

pull\_request:

branch: [main]

workflow\_dispatch:

jobs:

Jest:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v1

- name: Run Tests

run: |

npm install

node dataAPI &

npm run test

#

3.  Save the file on your server, and commit

4. Push the code to GitHub.

5. Go to the GitHub website and check that the action has run without errors.  If there are any errors drill down into the action to see the cause

**Step 6 - Disable All GitHub Actions**

NB:  Once you have completed this practical you should manually SWITCH OFF ALL the GitHub Actions on your server (there are three in total).

This is because:

* The JSDoc actions, they create new files or updated files in GitHub which can cause conflicts with your local code (see JSDoc tutorial)
* The JEST tests that we will be writing in future weeks require access to the cege0052 database.  This is behind the UCL firewall (which is why you need to use a VPN).  GitHub's automatic code runners don't have access to the VPN and don't have a UCL username and password, so all your automated tests will fail (in a real situation, the database would be accessible externally)

1. To switch off all the actions, you need to click on each action and then click on the three dots to the right and select *disable workflow*

*A screenshot of a computer

Description automatically generated*

2. Next time you push your code, make sure that none of the actions are running