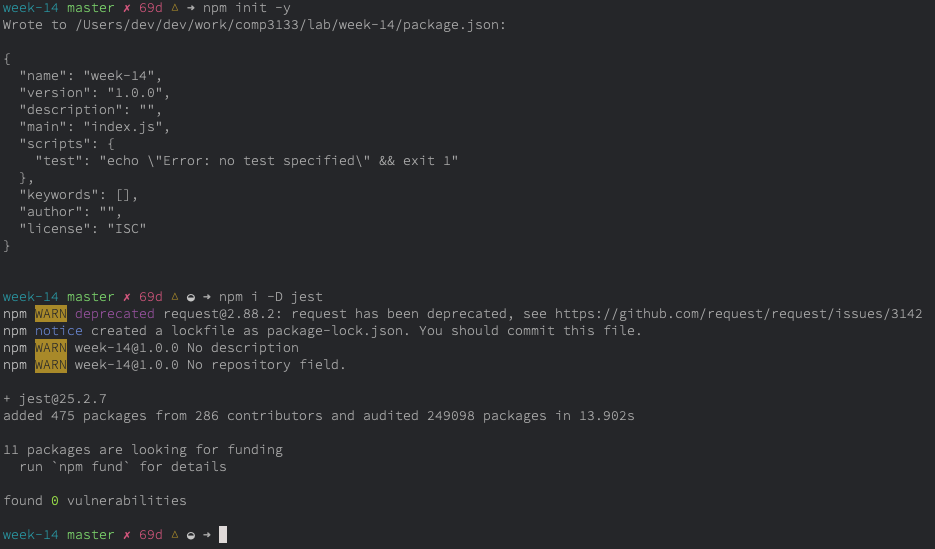
Jest

# A delightful JavaScript Testing Framework with a focus on simplicity

To Get started with Jest, it first must be installed as a dependency. You can use Yarn or NPM to install jest. The Jest documentation uses Yarn but NPM will also work.

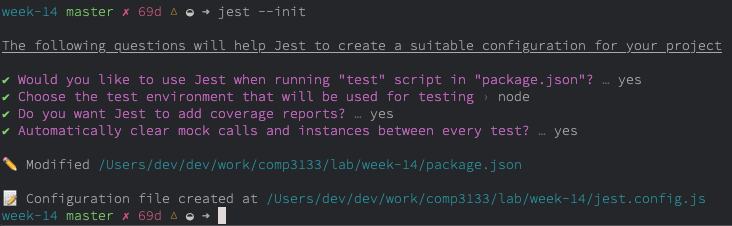
Yarn: yarn add –dev jest

NPM: npm install –save-dev- jest

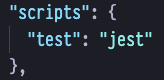


After Jest is installed, we’ll use it’s CLI tool to generate a basic configuration file for NPM and jest based on a few questions. Jest will create the file with a short description for each option.

Here is how mine looks after answering their questions. Your choices may differ but feel free to follow my lead.



Once completed you can change your package.json’s script tag to include the keyword “jest” so jest can be used to complete your tests.



Let’s get started by writing a hypothetical function to ensure that Jest has been included properly in our project.

Create a file called sum.js, the content of the file can be found below:

**function** **sum**(a, b) {

**return** a + b;

}

module.exports = sum;

Then, create a new file named sum.test.js. The .test. denotes the file as a unit test that will contain our actual test to be run by Jest.

Check the following page for what the test looks like.

sum.test.js

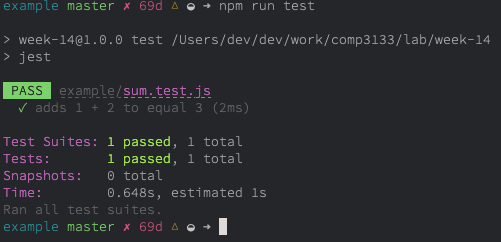
**const** sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {

expect(sum(1, 2)).toBe(3);

});

Finally, we are ready to run yarn test or npm run test. Let’s see the result.



As we can see above, the npm command for testing has been executed and Jest has provided us with a passing result. Jest also provides some metrics on how many tests were run, total number of tests passed and would show any errors if there was any in our code.

Congratulations on completing your first test with Jest! This test used two keywords reserved by the test suite. expect and toBe. These were both used in conjunction for us to take two values and ensure we could expect 1 + 2 to be 3. Jest offers a number of “matchers” that can be used to craft test cases. Jest has great documentation and I urge you to check out their section on [Using Matchers](https://jestjs.io/docs/en/using-matchers) for more information on what’s available to create quality tests to ensure the stability of your code.

Now that we have Jest configured, lets move forth and create a couple exercises to get the hand of writing unit tests. Unit tests are absolutely vital to include in your source to ensure you are not writing or working with a brittle code base and provide metrics on how well your code is running.

Exercise One

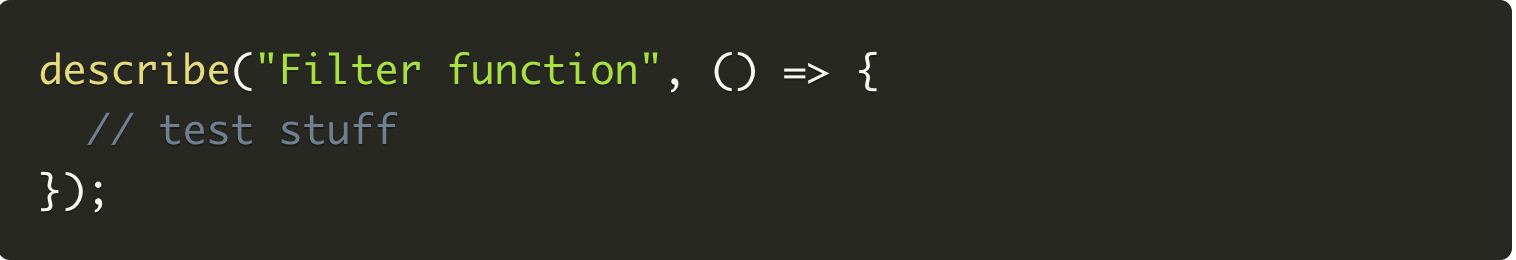
First things first, how do we know know what to test? When it comes to testing even a simple block of code could paralyze beginners. The most common question I’ve heard from JR devs is “How do I know what to test?”. If you’re creating a web application a good starting point would be testing every page of the app as unit tests can often times act as the only documentation around. The two most common scenarios you’ll run into is inheriting legacy code which has no unit tests running. It might be a good idea to suggest implementing test cases for the work that does not include tests. The second scenario is the need to implement new functionality out of thin air. Of course you’ll want to ensure that your code is backed by unit tests, right? So let’s do just that.

The formula for unit testing usually looks something like this:

1. Import the function to test
2. Give an **input to the function**
3. **Define what to expect** as the output
4. **Check if the function produces the expected result.**

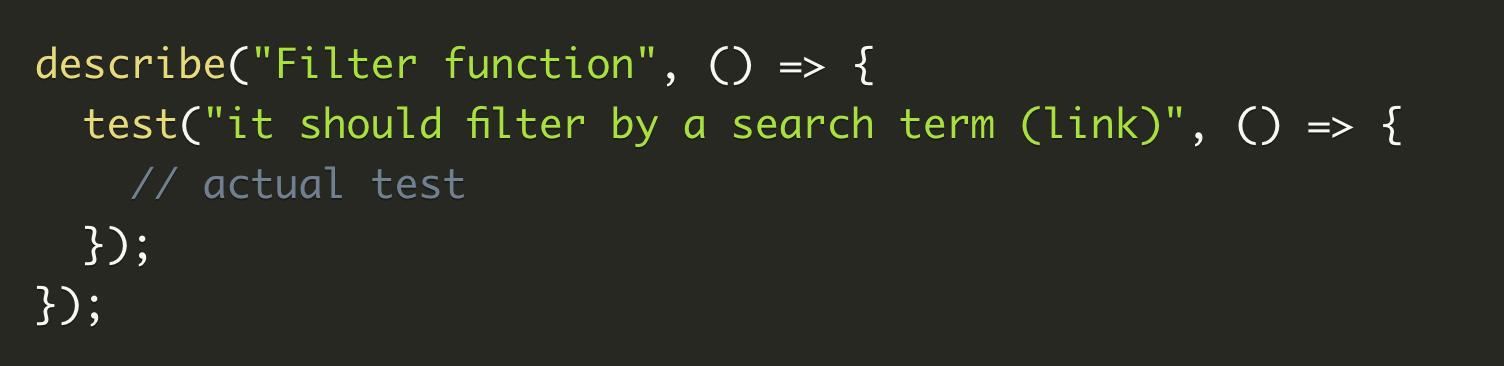
That’s all we’re going to be doing here. Testing won’t sound so abstract or bizarre if you think of it in these terms: **input – expected output – assert the result**. Let’s do our first one together.

Create a file called filterbyTem.test.js and create a test block.

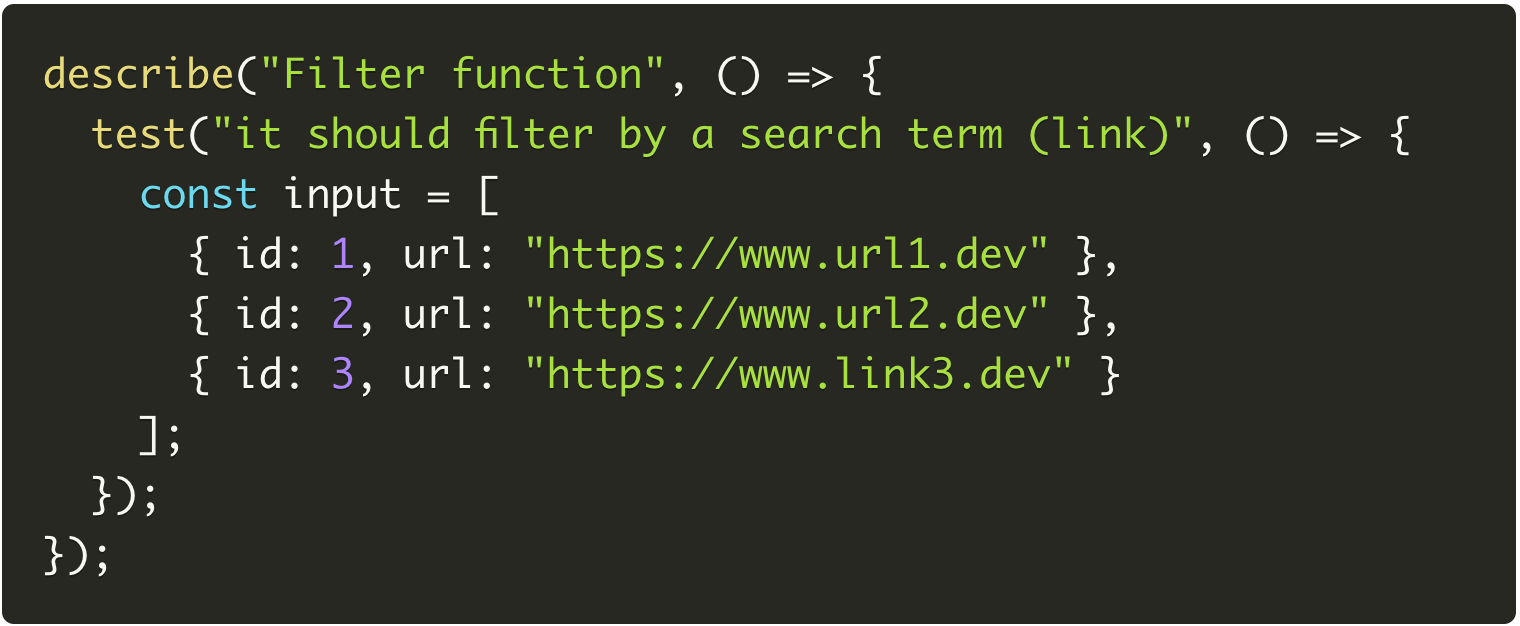


Above, I’d like to introduce you to our friend describe. Jest is very declarative. Unit tests should feel no different than reading English. Jest provides a method known as **describe** which is used to contain one or more related tests. Every time you start writing a new suite of tests, wrap it in a **describe** block. As you can see it takes two arguments: a string for describing the test suite and a callback for wrapping the actual test.

Next, we’re going to meet another Jest function called **test** which is the actual block. We previously used it in our sum test.

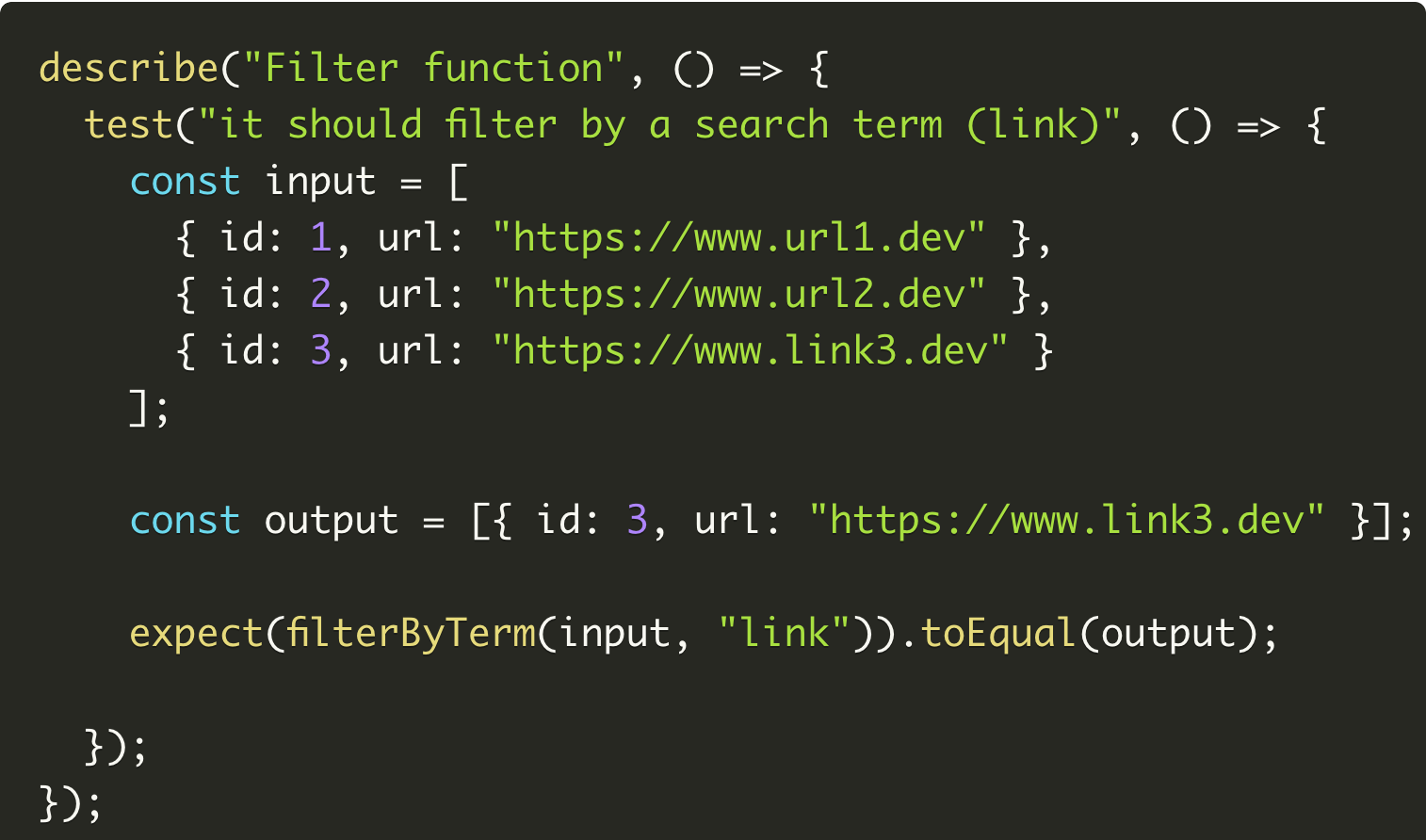


We are now ready to write our first test case. Remember, testing is a matter of **inputs, functions, and expected outputs**. First let’s define a simple input, an array of objects:



Next we’re going to define the expected result. If we look at the test case (the string passed as the first parameter to the test function), we are to filter out the array so our output only contains an object with a url that contains the word “link” in it.

In a Jest test you should wrap the function call inside **expect** which coupled with a **matcher** (a Jest function for checking the output) makes the actual tests. Follow through to the next page for an example of what the complete test would look like.



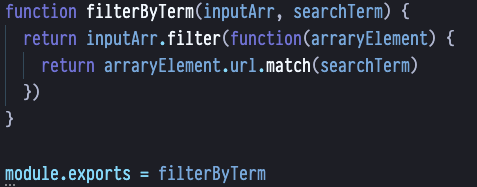
At this point we can run our test suite. Using Yarn or NPM, you know the command, just make sure you’ve altered the package.json to ensure that Jest is handling your tests in the script portion of the package json.

Please create a file called filterByTerm.js in the same folder carrying filterByTerm.test.js This is the file that is to hold the logic for the test. It’s best practice to keep your tests in one file (filterByterm.test.js) and the logic to be tested in a separate file (filterByTerm.js). The idea here is to import the exported function in filterByTerm.js and import it into the test file so that it’s available for testing. Leaving my markdown editor alone for a moment, I will create the two files so you have an accurate portrayal of what I’m alluding to.

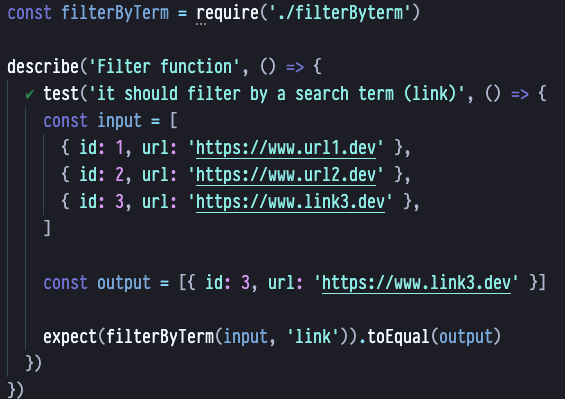
Let’s check what’s hiding in my editor and see what exactly is going on.

Please move forward to the next page.

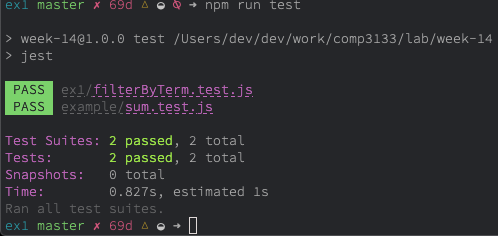
Logic to be tested:



The test:



And finally, the output from Jest showing the result of both our example and our first test.



You’ll see beside both the PASS labels that jest has tested both our initial test that helped us get off the ground (sum.test.js) and also our first exercise (filterByTerm.test.js).

After seeing just how Jest works, now it’s your turn to create some test cases as a I have above and create some new functions that need testing. There is a concept known as TDD (Test Driven Development) where logic and functionality is based on tests written **before** the development is completed. Once the test passes, you know that you have met the feature requirements for the new implementation. Feel free to try things out this way but we won’t get too deep into the ideology of TDD. If you’re curious to learn more about TDD, a great introduction can be found [here](https://www.guru99.com/test-driven-development.html).

When you’re completing the challenges, remember this one thing about tests:

**input – expected output – assert the result**

While your testing functionality, you will often times need to provide the input like we did with the array, craft your expect statement and use the necessary matchers to obtain the result. Feel free to use the example above to help you get through the exercises below.

Exercise Two

1. Describe a function that removes vowels from a sentence
2. Write a test case that defines what the function should achieve
3. Use the string ‘A dog is a friendly pet if it’s trained properly’
4. Write the functionality needed to remove all vowels from the above string
5. Test the functionality to ensure that your implementation removed all vowels

Exercise Three

1. Describe a function that is able to filter objects out that contain a false value
2. Write a test case that clearly defines the objective of the function
3. Implement an array of objects. The object should contain two keys, one key is called authenticated and the other key is called user\_id. Create your own user id’s. There should be 5 objects. Three of them must have authenticated set to true and two of them must have a value of false
4. Create the object and the functionality needed to complete the aforementioned task in step 3
5. Use Jest to ensure that the function is working as expected and you are left with only authenticated users

Exercise Four

1. Create an array with 10 elements. 7 Elements should be odd integer numbers, 3 even.
2. For every odd integer, multiply it by the previous even element.
3. The input array to be used is [ 2, 33, 4, 41, 6, 9, 77, 11, 13, 27 ]. Write a function that returns a new array containing the elements which were multiplied previously
4. The expected output should be [ 66, 164, 54, 462, 66, 78, 162 ]
5. Run Jest to ensure that the output expected matches the above array.

Don’t forget about checking Jest’s [matching operators](https://jestjs.io/docs/en/using-matchers)!