**Unit Testing with Jest**

[[Jeff P](https://waxlyrical.medium.com/?source=post_page-----ab03e7e923b6--------------------------------)](https://waxlyrical.medium.com/?source=post_page-----ab03e7e923b6--------------------------------)

[Jeff P](https://waxlyrical.medium.com/?source=post_page-----ab03e7e923b6--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ffcdc33fe8f64&operation=register&redirect=https%3A%2F%2Fwaxlyrical.medium.com%2Funit-testing-with-jest-ab03e7e923b6&user=Jeff+P&userId=fcdc33fe8f64&source=post_page-fcdc33fe8f64----ab03e7e923b6---------------------post_header-----------)

11 min read

·

Oct 30

2

I kind of never really saw the bigger picture with unit testing until I watched an excellent YouTube video by Kyle over at Web Dev Simplified (by the way, follow this guy, his videos are excellent!)

NOW I get it…. ok, so for small projects of say 1000 lines of code or less, it’s probably not necessary, as it’s fairly straightforward to skim through the code and identify the issues, especially if you’ve been adding console logs along the way that help confirm things are working as expected. However if you’re involved is medium to high complexity projects over thousands and thousands of lines of code, and/or your working as part of a team, then unit testing becomes really important.

Here are a couple of variables, and a few functions….it’s a very silly example, but at least it hopefully explains the importance of unit testing:

(Important note: no errors at all are reported with this code in VS Code!)

const word3 = 'brush';  
const word4 = 'pick';  
  
function checkLength(word3) {  
 if (word4.length > 4) {  
 return true;  
 } else {  
 return false;  
 }  
}  
  
function checkIfWordContainsI() {  
 if (word3.includes('i')) {  
 return true;  
 } else {  
 return false;  
 }  
}  
  
function concatTwoWords(word1, word2) {  
 return word1 + word3;  
}

if I called checkLength() without an argument, what response would I get? true or false?

checkLength()  
// false

what about if I pass in word 3 as the argument when I call it?

checkLength(word3)  
// false

It doesn’t matter either way, because the function is checking the if statement on word 4, which happens to be “pick”, so will always return false.

To some that might seem obvious, but if you’re new to JavaScript, something like that might not be too easy to spot, especially when you don’t have VS code highlighting any errors, so you’re not actually sure where you’re meant to be looking.

What about the second function? What if I call function checkIfWordContainsI() and pass in the word “island” ? Would I get true or false based on the code above?

checkIfWordContainsI("island")

Once again, if you’re an experienced Javascript programmer, it’s probably obvious that this would return “false”, but if you’re new to programming, it might not be so obvious that it doesn’t matter what you try to pass in to this function, the fact is that it’s still checking if word3 (i.e. “brush”) contains a letter i, which of course it doesn’t.

Let’s just quickly look at the last function:

function concatTwoWords(word1, word2) {  
 return word1 + word3;  
}

Ok so it’s expecting two arguments, and will place them together…. so what will the console return if I call this function with the following arguments?

concatTwoWords("tooth", "paste")

If your answer was “toothbrush”, then congratulations, you are good at spotting things that might not be obvious to some!

The issue here is that someone could’ve accidentally typed in this function wrong…. what they meant to type in was

function concatTwoWords(word1, word2) {  
 return word1 + word2;  
}

when in fact they typed in:

function concatTwoWords(word1, word2) {  
 return word1 + word3;  
}

Now because word3 has already been declared as a valid string, JavaScript will correctly assume you want to concatenate word1 (which is the first argument you passed in) with word3 (which is the declared variable of “brush”.

So JavaScript did EXACTLY what your code asked it to! No errors are reported, because technically there’s nothing wrong here….it just so happens that you might be seeing something that you weren't expecting, and once again, because there's no code error, it would be difficult to find in a project with thousands and thousands of lines of code, especially if the function was far more complex, and contained multiple if/else statements!

This is what unit testing solves!

**Using Jest**

Unit testing essentially allows you to write out tests for all your functions, so that you can confirm that the results will be what you expect them to be…every time. If any test fails, then you know exactly which function to go back and check carefully. Going back to the concatTwoWords function example, you may be expecting the function to ALWAYS return a concatenation of two words that are passed into the function, and if they don’t, then somethings not right. Perhaps you’re working on a project with another coder, who accidentally changed word2 to word3, or perhaps it was a slip of the finger when typing….the point is that you are notified of where an issue is, and can go and fix it.

So let’s see how we could write a very basic unit test for the concatTwoWords function…

First we need to ensure that we have a package.json file in our project folder. If we don’t already have one, we can run:

npm init -y

The -y flag in the npm init -y command is used to initialize a new package.json file without prompting you for any input. It stands for "yes," and it automatically accepts all the default values when generating the package.json file.

Next we can install Jest using NPM into our current directory:

npm install --save-dev jest

I would also strongly suggest installing jsdom as well, and I’ll explain why it could be useful shortly….

npm i jest-environment-jsdom --save-dev

ok now we have our packages installed, we can set our package.json file to use jest by including the following:

"scripts": {  
 "test": "jest"  
 },

with this in place, we should be able to run “**npm test**” from the command line.

The next thing we want to do is test our function. Assuming our script.js file is as follows…..

const word3 = 'brush';  
const word4 = 'pick';  
  
function checkLength(word3) {  
 if (word4.length > 4) {  
 return true;  
 } else {  
 return false;  
 }  
}  
  
function checkIfWordContainsI() {  
 if (word3.includes('i')) {  
 return true;  
 } else {  
 return false;  
 }  
}  
  
function concatTwoWords(word1, word2) {  
 return word1 + word3;  
}

then we can run tests on all three functions, or just focus on the one we want to test. In this case we’ll focus just on concatTwoWords() so we’ll specify this as the function to export:

const word3 = 'brush';  
const word4 = 'pick';  
  
function checkLength(word3) {  
 if (word4.length > 4) {  
 return true;  
 } else {  
 return false;  
 }  
}  
  
function checkIfWordContainsI() {  
 if (word3.includes('i')) {  
 return true;  
 } else {  
 return false;  
 }  
}  
  
function concatTwoWords(word1, word2) {  
 return word1 + word3;  
}  
  
// export the function we want to test  
module.exports = { concatTwoWords };

Next we’ll create a **script.test.js** file, with the following content:

// our script.test.js file  
  
const { concatTwoWords } = require('./script');  
  
test('concatenates two words that are passed into the function', () => {  
 concatTwoWords((word1, word2) => word1 + word2);  
  
 // Test the function  
 expect(concatTwoWords('hello', 'world')).toBe('helloworld');  
});

So we use the word test to define the test we want to make…. the first argument is simply a string that allows you to explain what you are trying to test, and the second argument is a function that expresses the arguments the function should expect, and then what it should actually do, so in the example above, it takes two arguments (word1, word2) and adds them together.

We then write our “expect” code, which says that we expect that if we pass in two strings of “hello” and “world” into our function, then we should get “helloworld” to be returned.

With that in place, lets now run our test….

npm test

Here is the response….

FAIL ./script.test.js  
 × concatenates two words that are passed into the function (9 ms)  
  
 ● concatenates two words that are passed into the function  
  
 expect(received).toBe(expected) // Object.is equality  
  
 Expected: "helloworld"  
 Received: "hellobrush"  
  
 6 |  
 7 | // Test the function  
 > 8 | expect(concatTwoWords('hello', 'world')).toBe('helloworld');  
 | ^  
 9 | });  
  
 at Object.toBe (script.test.js:8:44)  
  
Test Suites: 1 failed, 1 total  
Tests: 1 failed, 1 total  
Snapshots: 0 total  
Time: 1.011 s  
Ran all test suites.

So as you can see, we were expecting “helloworld” but we did in fact get back “hellobrush”

We can now go back to our function to investigate, and at this point we should be able to see why it’s adding “brush” — It’s because the function is attempting to concatenate word1 (correct) with word3 (incorrect).

Lets fix our function, and run the test again:

function concatTwoWords(word1, word2) {  
 return word1 + word2;  
}

npm test  
  
 PASS ./script.test.js  
 √ concatenates two words that are passed into the function (5 ms)  
  
Test Suites: 1 passed, 1 total  
Tests: 1 passed, 1 total  
Snapshots: 0 total  
Time: 0.969 s, estimated 1 s  
Ran all test suites.

So as you can see, the test has helped us find what could potentially be something tricky to troubleshoot, as there were no errors ever reported in VS Code. Hopefully it’s obvious how useful this could be on very large, complex projects.

**.toBe (or .not.ToBe) — that is the question!**

You may have noticed that we used **.toBe** as part of our expect statement. This is known as a “matcher”

.toBe is used to to compare primitive values such as strings or numbers, but will return errors if you wanted to compare arrays or objects.

For example, lets create another simple function called myArray() and then export it:

function concatTwoWords(word1, word2) {  
 return word1 + word2;  
}  
  
function myArray(array) {  
 return [...array];  
}  
  
// export the function we want to test  
module.exports = {  
 concatTwoWords,  
 myArray,  
};

The array parameter is passed to the myArray() function.  
[…array] creates a new array containing all the elements of the array parameter. It effectively duplicates the contents of the original array.

If you ran this code normally, you’d get the following:

myArray([1,2,3,4,5])  
(5) [1, 2, 3, 4, 5]

so you can see it’s returned an array of the same values in the same order as what was passed into the function.

Let’s now test this function with the .toBe matcher as another test….

const { concatTwoWords, myArray } = require('./script');  
  
test('concatenates two words that are passed into the function', () => {  
 // Set up the mock implementation for concatTwoWords  
 concatTwoWords((word1, word2) => word1 + word2);  
  
 // Test the function  
 expect(concatTwoWords('hello', 'world')).toBe('helloworld');  
});  
  
  
  
test('correctly clones the array passed in', () => {  
  
 //define an array to pass into the function  
 const array = [1, 2, 3, 4, 5];  
  
 // Test the function  
 expect(myArray(array)).toBe(array);  
});

Here is the return after we run npm test…

FAIL ./script.test.js  
 √ concatenates two words that are passed into the function (4 ms)  
 × correctly clones the array passed in (7 ms)  
  
 ● correctly clones the array passed in  
  
 expect(received).toBe(expected) // Object.is equality  
  
 If it should pass with deep equality, replace "toBe" with "toStrictEqual"  
  
 Expected: [1, 2, 3, 4, 5]  
 Received: serializes to the same string  
  
 15 |  
 16 | // Test the function  
 > 17 | expect(myArray(array)).toBe(array);  
 | ^  
 18 | });  
 19 |  
  
 at Object.toBe (script.test.js:17:26)  
  
Test Suites: 1 failed, 1 total  
Tests: 1 failed, 1 passed, 2 total  
Snapshots: 0 total  
Time: 0.981 s, estimated 1 s  
Ran all test suites.

So we can see that it passed the first test, but not the second. The second test expected [1, 2, 3, 4, 5] but the return was a shallow **copy** of the same array, not THE original array. The two arrays are basically objects stored in different memory locations, and so are different, even though they look identical.

In order to get this type of test to pass, we’d have to replaced the matcher or .toBe with .toEqual which will check if the values in the array are the same, not necessarily that the arrays are the same array. We’ll run again with .toEqual

const { concatTwoWords, myArray } = require('./script');  
  
test('concatenates two words that are passed into the function', () => {  
 // Set up the mock implementation for concatTwoWords  
 concatTwoWords((word1, word2) => word1 + word2);  
  
 // Test the function  
 expect(concatTwoWords('hello', 'world')).toBe('helloworld');  
});  
  
test('correctly clones the array passed in', () => {  
  
 //define an array to pass into the function  
 const array = [1, 2, 3, 4, 5];  
  
 // Test the function  
 expect(myArray(array)).toEqual(array);  
});

which now returns…

PASS ./script.test.js  
 √ concatenates two words that are passed into the function (4 ms)  
 √ correctly clones the array passed in (1 ms)  
  
Test Suites: 1 passed, 1 total  
Tests: 2 passed, 2 total  
Snapshots: 0 total

There are plenty of other “matchers” that can be used, with more details available on Jest’s website: <https://jestjs.io/docs/getting-started>

**jsdom**

I mentioned jsdom earlier because I ran into an issue when trying out Jest, and it took me a while to figure out what the problem was.

Initially my script.js file also contained other things, not just the three functions I was planning on testing. This caused the following error to be generated whenever I ran tests….

This was a snippet of my script.js file, with the functions I wanted to test with jest near the bottom….

document.querySelector('.again').addEventListener('click', () => {  
 let guessValue = parseInt(document.querySelector('.guess').value);  
  
 if (guessValue === secretNumber) {  
 document.querySelector('.number').textContent = '?';  
 document.querySelector('body').style.backgroundColor = '#000000';  
 if (Number(score) > Number(highScore)) {  
 highScore = score;  
 document.querySelector('.highscore').textContent = highScore; // Update the high score in the UI  
 }  
 score = 10;  
 document.querySelector('.guess').value = '';  
 document.querySelector('.message').textContent = 'Make a guess';  
 }  
});  
  
const word3 = 'brush';  
const word4 = 'pick';  
  
function checkLength(word3) {  
 if (word4.length > 4) {  
 return true;  
 } else {  
 return false;  
 }  
}  
  
function checkIfWordContainsI() {  
 if (word3.includes('i')) {  
 return true;  
 } else {  
 return false;  
 }  
}  
  
function concatTwoWords(word1, word2) {  
 return word1 + word2;  
}  
  
function myArray(array) {  
 return [...array];  
}  
  
function myObject(object) {  
 return { ...object };  
}  
  
// export the function we want to test  
module.exports = {  
 concatTwoWords,  
 myArray,  
 myObject,  
};

As you can see, at the top of the file, I had some code for something else, not just pure functions…. whenever I ran npm test, I was seeing this error:

FAIL ./script.test.js  
 ● Test suite failed to run  
  
 The error below may be caused by using the wrong test environment, see https://jestjs.io/docs/configuration#testenvironment-string.  
 Consider using the "jsdom" test environment.  
  
 ReferenceError: document is not defined  
  
 46 | // });  
 47 |  
 > 48 | document.querySelector('.again').addEventListener('click', () => {  
 | ^  
 49 | let guessValue = parseInt(document.querySelector('.guess').value);  
 50 |  
 51 | if (guessValue === secretNumber) {  
  
 at Object.document (script.js:48:1)  
 at Object.require (script.test.js:1:47)  
  
Test Suites: 1 failed, 1 total  
Tests: 0 total  
Snapshots: 0 total  
Time: 1.002 s  
Ran all test suites.

So I could see a reference to using a jsdom test environment instead of via node.

In order to do this, I first had to install the jsdom package (see at the start of this medium posting) and then had to modify my package.json file to be as follows:

"scripts": {  
 "test": "jest --watchAll --config=jest.config.json"  
 },

So I’m basically saying that I want jest to use a **jest.config.json** file

Next I had to create the **jest.config.json** file, and add the following to it:

"scripts": {  
 "test": "jest --watchAll --config=jest.config.json"  
 },

now when I ran npm test I was seeing a different error:

FAIL ./script.test.js  
 ● Test suite failed to run  
  
 TypeError: Cannot read properties of null (reading 'addEventListener')  
  
 46 | // });  
 47 |  
 > 48 | document.querySelector('.again').addEventListener('click', () => {  
 | ^  
 49 | let guessValue = parseInt(document.querySelector('.guess').value);  
 50 |  
 51 | if (guessValue === secretNumber) {  
  
 at Object.<anonymous> (script.js:48:33)  
 at Object.require (script.test.js:1:47)  
  
Test Suites: 1 failed, 1 total  
Tests: 0 total  
Snapshots: 0 total  
Time: 2.938 s  
Ran all test suites.

This had me stuck for a while, but evenbtually I got past this by ensuring that the code causing the problem was wrapped with a listener for ‘DOMContentLoaded’

document.addEventListener('DOMContentLoaded', () => {  
 document.querySelector('.again').addEventListener('click', () => {  
 let guessValue = parseInt(document.querySelector('.guess').value);  
  
 if (guessValue === secretNumber) {  
 document.querySelector('.number').textContent = '?';  
 document.querySelector('body').style.backgroundColor = '#000000';  
 if (Number(score) > Number(highScore)) {  
 highScore = score;  
 document.querySelector('.highscore').textContent = highScore; // Update the high score in the UI  
 }  
 score = 10;  
 document.querySelector('.guess').value = '';  
 document.querySelector('.message').textContent = 'Make a guess';  
 }  
 });  
});

PASS ./script.test.js  
 √ concatenates two words that are passed into the function (11 ms)  
 √ correctly clones the array passed in (1 ms)  
 √ correctly clones the object passed in  
  
Test Suites: 1 passed, 1 total  
Tests: 3 passed, 3 total  
Snapshots: 0 total  
Time: 2.836 s  
Ran all test suites.  
  
Watch Usage: Press w to show more.

**DOMContentLoaded** is an event in the Document Object Model (DOM) of web browsers. It is fired when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and other external resources to finish loading. In other words, it signals that the HTML structure of a web page is **ready to be manipulated and interacted with using JavaScript.**

So what was happening was that the event listener code was being executed before the DOM had been fully loaded in the test environment.

# Testing MongoDB Document Return Objects in Jest

[[Ibiyemi Pedro](https://ibiyemipedro.medium.com/?source=post_page-----62f1546315af--------------------------------)](https://ibiyemipedro.medium.com/?source=post_page-----62f1546315af--------------------------------)

[Ibiyemi Pedro](https://ibiyemipedro.medium.com/?source=post_page-----62f1546315af--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F3da532abc1a6&operation=register&redirect=https%3A%2F%2Fibiyemipedro.medium.com%2Ftesting-mongodb-document-return-objects-in-jest-62f1546315af&user=Ibiyemi+Pedro&userId=3da532abc1a6&source=post_page-3da532abc1a6----62f1546315af---------------------post_header-----------)

2 min read

·

Jul 7

I had to write this because I tried to look for resources to help me test some functions in a repository for a MongoDB project. Typically most of the functions did a find() findAll() aggregate() and then some of these methods had a .exec() as in find().exec() .

Most of my tests involved testing the return value of these methods which were of type MongoDocument , and Mongo attaches extra methods to the document like .toObject() .save() to the returned object

A simple solution I found was instead of using a mock data of your desired response,

const myObjectMock = {  
 name: 'test name',  
 desc: 'test desc',  
}

you should use a mock of the MongoDB object as such

const myObjecMongoDocumentMock = {  
 ...myObjectMock,  
 \_id: 'object-id',  
 \_v: 1,  
 save: jest.fn(),  
 toObject: jest.fn(),  
 exec: jest.fn(),  
}

This way you can mock the extra MongoDB methods and return a desired response for your mock

An example of this in an application :

## Imagine we want to test this repository method

public async getAll(filter: FilterQuery<ProjectDocument>): Promise<PartialProject[]> {  
 const projects = await this.projectsModel.find(filter).exec()  
  
 return projects.map((project) => modelWithId(project.toObject()))  
 }

## The test function would look this function below

describe('getAll method', () => {  
 it('should return all objects', async () => {  
   
 // mock the mongoDB find method to return a mock of mongoDB object with the extra methods you need  
 objectModel.find.mockReturnValueOnce({  
 exec: () => Promise.resolve([myObjecMongoDocumentMock]),  
 })  
  
 // mocking the .toObject method to return desired response  
 myObjecMongoDocumentMock.toObject = jest  
 .fn()  
 .mockReturnValue({ ...fullProjectMock, \_id: 'doc-id', \_v: 1 })  
  
 // call the repository function  
 const result = await repository.getAll({})  
  
 // your assertion  
 expect(result.length).toBeGreaterThan(0)  
 })  
 })

Hope this was helpful.

# Mocking with Jest in Typescript/Javascript

[[Vivek Murarka](https://medium.com/@vivek.murarka?source=post_page-----d203699cc617--------------------------------)](https://medium.com/@vivek.murarka?source=post_page-----d203699cc617--------------------------------)

[Vivek Murarka](https://medium.com/@vivek.murarka?source=post_page-----d203699cc617--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Faf05810e69d9&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40vivek.murarka%2Fmocking-with-jest-in-typescript-javascript-d203699cc617&user=Vivek+Murarka&userId=af05810e69d9&source=post_page-af05810e69d9----d203699cc617---------------------post_header-----------)

7 min read

·

Aug 9

3

In this article I will be discussing about Jest mocking, different type of use cases where mocking become tricky and testing become daunting. Aim of this article is to guide with example for most common testing scenarios and some not so common ones.



So, let’s dive write into the article

What is mocking and why we need to mock?  
Keeping it short, mocking any function, object or class is creating a dummy duplicate implementation, so that we could avoid calling actual dependency which is beyond the testing scope.

## Test Scenario 1

To test the first class, we mock a second class on which the first class depends.

Here, we want to ensure that the `area` function in the `Circle` module functions correctly, regardless of what the `multiply` function returns.

import {multiply} from './maths'  
  
export function area(radius:number) {  
 return multiply(Math.PI, multiply(radius, radius))  
}

So in our test case we will mock multiple function

import {area} from './circle'  
  
jest.mock('./maths', () => ({  
 multiply: jest.fn().mockReturnValue(1)  
}));   
  
describe('area', () => {  
 test("calculates the area of a circle", () => {   
 expect(area(1)).toBe(1)  
 })  
});

## Test Scenario 2:

What if we don’t want to mock all the function inside ‘maths’ module and only want multiply to be mocked, we could use jest.requireActual to keep all the original implementation except for multiply

import {area} from './circle'  
import {multiply} from './maths'  
  
jest.mock('./maths', () => ({  
 ...jest.requireActual('./maths'),  
 multiply: jest.fn().mockReturnValue(1)  
}));   
  
describe('area', () => {  
 test("calculates the area of a circle", () => {   
 expect(area(1)).toBe(1)  
 expect(multiply).toHaveBeenCalled();  
 })  
});

## Test Senario 3:

Mocking variables, such as environment variables and global variables, is sometimes necessary for test setup. One interesting use case is mocking the `NODE\_ENV` environment variable to test code behavior in different environments:

export function config(){  
 if(process.env.NODE\_ENV === "production"){  
 return "production"  
 }else if(process.env.NODE\_ENV === "development"){  
 return "development"   
 }else{  
 return "test"  
 }  
}

To test we could follow following approach, here we have not used mock from jest but instead we have changed the real values

describe('config', () => {  
 const env = {...process.env} ; // save current env  
  
 afterAll(() => {  
 process.env = env;  
 });  
  
 it("returns production when NODE\_ENV is production", () => {  
 process.env.NODE\_ENV = "production";  
 expect(config()).toBe("production")  
   
 });  
 it("returns development when NODE\_ENV is development", () => {  
 process.env.NODE\_ENV = "development";  
 expect(config()).toBe("development")  
 });  
  
});

There is a better and more detailed approach discussed here:  
[How to mock process.env when writing unit tests with Jest?](https://medium.com/weekly-webtips/how-to-mock-process-env-when-writing-unit-tests-with-jest-80940f367c2c)

## Test Scenario 4

When dealing with chained functions, such as methods that are called in a chain like `response.status().send()`, we need to mock each chained method:

const mockResponse = () => {  
 const res = {} as unknown as Response;  
 res.status = jest.fn().mockReturnValue(res);  
 res.json = jest.fn().mockReturnValue(res);  
 res.send = jest.fn().mockReturnValue(res);  
 return res;  
 };  
await testMyFunction(mockRequest, mockResponse, mockNext);  
expect(mockResponse().status).toHaveBeenCalledWith(200);

## Test Scenario 5

Testing async method which are called inside another method  
If calling method is not async. In this case it can’t await on another async method, for example:

async function getSomeInformation( x: any) {  
 let n: ReturnType<typeof setTimeout>;  
 n = setTimeout(() =>{console.log("Hello From Async function")}, 1000);  
 clearTimeout(n);  
}  
  
export function testA(req:any,res:any,next:any){  
 if(req?.header?.("test") == "true"){  
 console.log("Inside If");  
 next()  
 }  
 // couln't await for getSomeInformation to complete  
 getSomeInformation("test").then(() => {  
 console.log("Inside Then");  
 next();  
 }).catch(() => {  
 console.log("Inside Catch");  
 next();  
 });  
}

now when we test method ‘testA’ we couldn’t test because our next operation is called but jest couldn’t confirm that before shutting down.

import {testA} from './examples'  
  
describe('testA', () => {  
 it("testA when ", async () => {  
 const req = {  
 header: jest.fn().mockReturnValue("true")  
 }  
 const res = {}  
 const next = jest.fn(()=>{console.log("Inside Next")})  
 await testA(req,res,next)  
 //following will fail, Although our next is called twice,   
 // but jest environment is torn before compliler can return from getSomeInformation  
 expect(next).toHaveBeenCalledTimes(2);  
 })  
});

To correct this we shall await for getSomeInformation inside testA method, making this also async function.

async function getSomeInformation( x: any) {  
 let n: ReturnType<typeof setTimeout>;  
 n = setTimeout(() =>{console.log("Hello From Async function")}, 1000);  
  
 clearTimeout(n);  
   
}  
  
export async function testA(req:any,res:any,next:any){  
 if(req?.header?.("test") == "true"){  
 console.log("Inside If");  
 next()  
 }  
 await getSomeInformation("test").then(() => {  
 console.log("Inside Then");  
 next();  
 }).catch(() => {  
 console.log("Inside Catch");  
 next();  
 });  
}

## Test Scenario 6

Testing codes which are using third party library whose functionality is beyond the scope of testing. Some common library that requires mocking in test environment.

1. Testing method which calls axios method, while testing we don’t want to do actual api call.

import axios from "axios";  
  
export async function getUserDetails() {  
  
 const response = await axios.get("https://example.com/users", {  
 headers: {  
 "Content-Type": "application/json",  
 },  
 })  
 return response.data;  
}

To test this above method we could mock axios and then check if our axios get method was called and what argument were passed. We also mock the response so that we can continue testing users function without making call.

import axios from "axios";  
import { getUserDetails } from "./users";  
  
jest.mock("axios");  
  
describe("mocking axios", () => {  
  
 it("should fetch users", async () => {  
 const users = [{name: "Bob"}];  
 const resp = {data: users};  
 (axios.get as jest.Mock).mockResolvedValue(resp);  
 // asume getUserDetails which calls axios.get  
 const actualOutput = await getUserDetails();  
 expect(axios.get).toHaveBeenCalledTimes(1);  
 expect(axios.get).toHaveBeenCalledWith("https://example.com/users", {  
 headers: {  
 "Content-Type": "application/json",  
 },  
 });  
 });  
  
});

Similarly we could test other REST methods.

2. Testing methods which depends upon jwt verification, for example:

import jwt from "jsonwebtoken";  
const mySecretKey= "adminuserKey"  
  
export async function testMySecurity(req,res,next: any) {  
  
 const token = req?.header?.authorization?.split(" ")[1];  
 jwt.verify(token, mySecretKey, (err, authData) => {  
 if (err) {  
 console.log("Error in verifying token");  
 res.status(403).send("Forbidden");  
 } else {  
 next(authData);  
 }  
 });  
}

To test this we can’t actually provide a JWT token signed with private key for testing, instead we will spy on verify method and bypass actual verification process and instead check what our function would do if verification failed or passed in real time.

import jwt from "jsonwebtoken";  
import { testMySecurity } from "./security";  
import { Request,Response } from "express";  
  
  
const mockResponse = () => {  
 const res = {} as unknown as Response;  
 res.status = jest.fn().mockReturnValue(res);  
 res.json = jest.fn().mockReturnValue(res);  
 res.send = jest.fn().mockReturnValue(res);  
 return res;  
 };  
  
describe("testMySecurity", () => {  
  
 it("should call next by bypassing verification", async () => {  
 jest.spyOn(jwt, "verify")  
 .mockImplementation(jest.fn((token, secretOrPublicKey, callback) => callback(null, { sub: "user\_id" })));  
  
 const mockRequest = {  
 header: {  
 authorization: "BearerToken testToken",  
 } as unknown as Request,  
 };  
  
 const mockNext = jest.fn(() => "next");  
 await testMySecurity(mockRequest, mockResponse, mockNext);  
 expect(mockNext).toHaveBeenCalledTimes(1);  
  
 });  
  
 it("should call next by bypassing verification", async () => {  
  
 jest.spyOn(jwt, "verify")  
 .mockImplementation(jest.fn((token, secretOrPublicKey, callback) => callback( new Error("Forbidden"), null)));  
  
 const mockRequest = {  
 header: {  
 authorization: "",  
 } as unknown as Request,  
 };  
  
 const mockNext = jest.fn(() => "next");  
 await testMySecurity(mockRequest, mockResponse, mockNext);  
 expect(mockResponse().status).toHaveBeenCalledWith(403);  
 });  
  
});

3. Any other library for example assume admin is a library and we need to mock auth method inside that, we can mock by spying on the method we intent to bypass, thus spying help us mocking modules inside it statement and we can change the expectation every time as per unit cases

import adminMock from "admin";  
const mockAuth = {} as unknown as adminMock.auth.Auth;  
  
describe(" testAuthorisation",() =>{  
  
 it("test authorisation withod using admin library", async() =>{  
 mockAuth.verifyIdToken = jest.fn(mockRejectedIdToken);  
 //spy on auth method of admin, and mock the impelementation to mockAuth  
 jest.spyOn(adminMock, "auth").mockImplementation(() => mockAuth);  
 });  
});

## Test Scenario 7

Testing async method, in many cases one method might depend on another async method, and to test the functionality of first method we would need to mock the second function. For example here getUserDetails depends on getUserBalance

import axios from "axios";  
export async function getUserDetails() {  
 const userBalance: number= await getUserBalance("123");  
 if(userBalance >= 100) {  
 return "rich";  
 }else {  
 return "poor";  
 }  
}  
  
async function getUserBalance(id: string) {  
 const userBalance:number = await new Promise(() => {  
 return axios.get(`https://example.com/users/${id}/balance`, {  
 headers: {  
 "Content-Type": "application/json",  
 },  
 });  
 })  
 return userBalance;  
 }

To do this we need to create seperate files for each such method that needs to be mocked. Hence, we will create another file which will have

import axios from "axios";  
export default async function getUserBalance(id: string) {  
 const userBalance:number = await new Promise(() => {  
 return axios.get(`https://example.com/users/${id}/balance`, {  
 headers: {  
 "Content-Type": "application/json",  
 },  
 });  
 })  
 return userBalance;  
 }

And to mock this we will create seperate sub folder names **\_\_mocks\_\_,** and create a file of same name that returns the mock functionality**for**getUserBalance, like

const getUserBalance = () =>  
 new Promise((resolve) => {  
 resolve( 200 );  
 });  
export default getUserBalance;

Now we can create our test case like below

import { getUserDetails } from "./index";  
  
jest.mock("./getUserBalance",()=>{  
 return jest.fn().mockReturnValueOnce(200).mockReturnValueOnce(99);  
});  
describe("test async method", () => {  
 it("should return rich", async () => {  
 const userBalance = 200;  
 const resp = {data: userBalance};  
   
 const actualOutput = await getUserDetails();  
   
 expect(actualOutput).toEqual("rich");  
 });  
 it("should return poor", async () => {  
   
 const actualOutput = await getUserDetails();  
   
 expect(actualOutput).toEqual("poor");  
 });  
});

Thus we are able to test our dependent methods using mocks.

## Note:

*jest.mock will be executed first and will mock all the occurrence of the modules.*

In conclusion, utilizing Jest’s mocking capabilities is crucial for effective unit testing. By understanding and applying these techniques, you can ensure reliable and comprehensive testing of your codebase. Remember that `jest.mock` is executed first and mocks all occurrences of the specified modules. This enables you to tailor your tests to various scenarios and focus on the behavior you intend to verify.