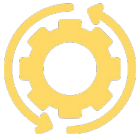
Unit Testing the eCommerce Applications-1

Relevel
by Unacademy

Class Agenda



We will understand Testing – What, Why and How.

What is called Test Driven Development and how it works.

We will be getting started with Unit Testing.

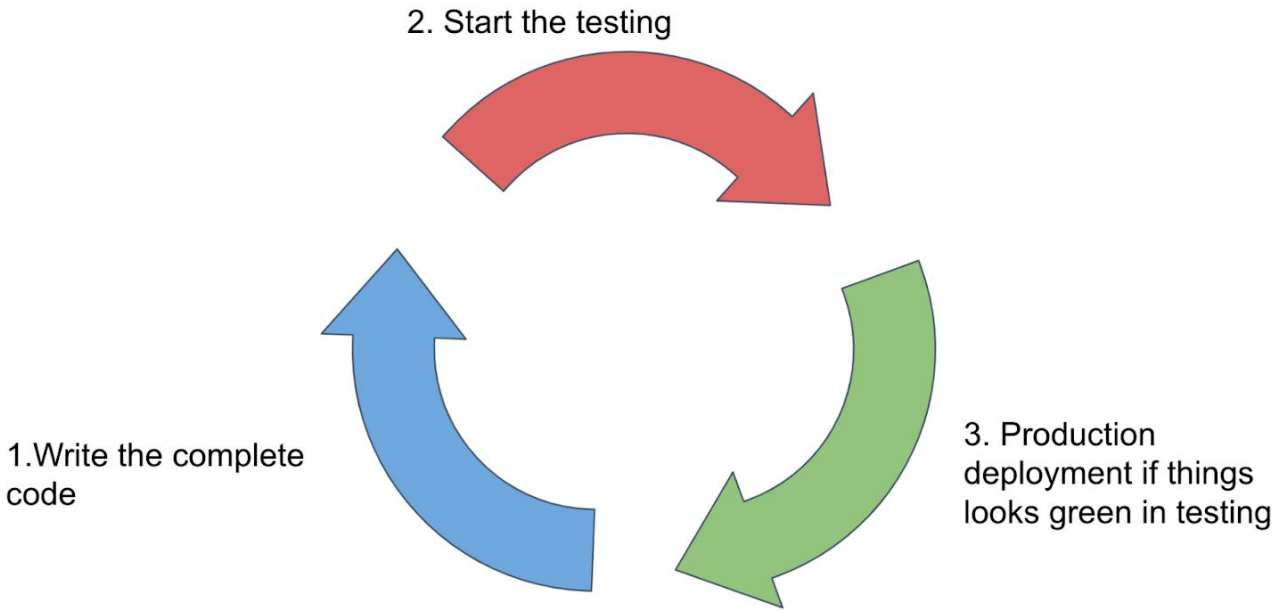
Educator Introduction

What is testing and why is it needed ?

- When you develop an application, it is quite important to test it before deploying it on the production server; otherwise, an unwanted bug can cause a lot of problems.
- No enterprise-level application is ever deployed without testing, whether manual or automated.
- In manual testing, you run the application, provide some input to it and check whether you are obtaining the proper output.
- In automated testing, the application is tested by writing more code that is solely responsible for testing your application–a type of a testing layer.
- So, you know that testing is important, and no one can argue against it, not even experienced software developers.

- So, the fact that testing is extremely important for any piece of software cannot be challenged.
- The main debate is regarding the stage at which you should test your application.

How has testing been done conventionally ?



Challenges of the above approach :

- Testing is added very late in the cycle of development
- Last minute issues, might lead to the delay in the release because of re-development
- Compromised testing because of the rush of the release

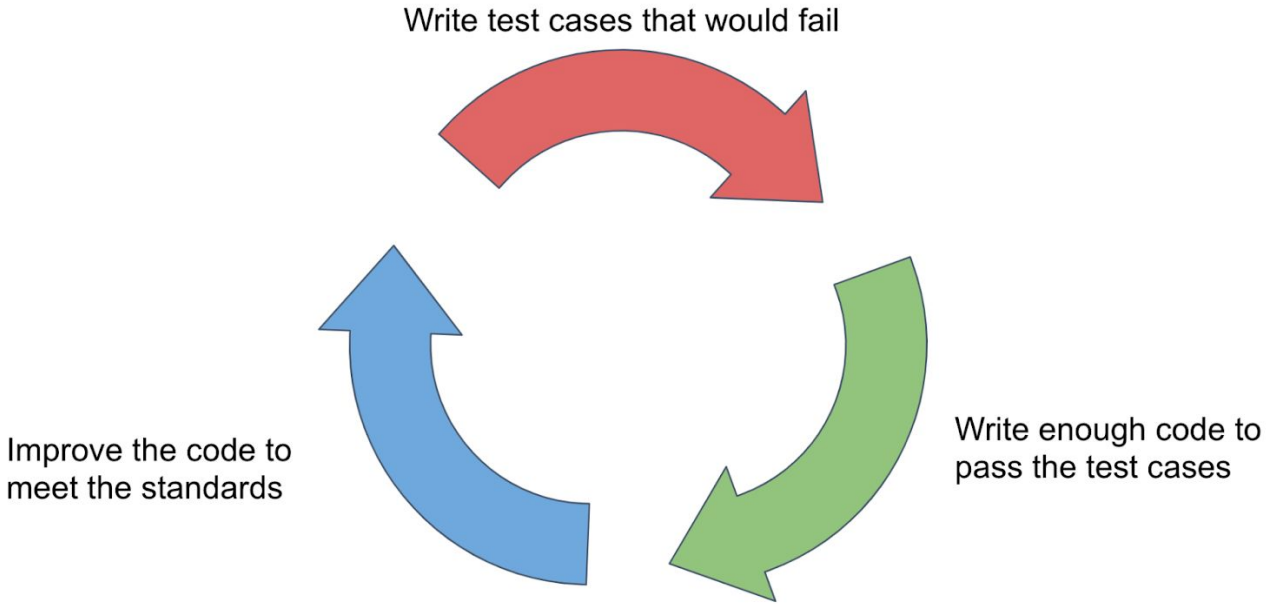
What is Test Driven Development

- Another option is that you can write the testing code even before you start developing the application and then write the code to pass all the test cases and carry out refactoring as required. This is called ***test-driven development*** or ***TDD***.

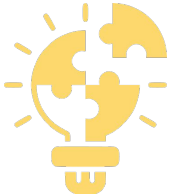
TDD is an iterative process, where:

- The developer first writes the test cases based on the requirements. If you are building a calculator, then the developer will write test cases that assert that 3 + 5 should be 8, 9 * 2 should be 18, and so on.
- Then, the developer writes only enough code to pass all the test cases.
- Once all the test cases are passed, the developer can perform other refactoring to meet the standards, such as making the code clean, removing the duplicate code, and improving the performance.

- This cycle is also called the ***test-code-refactor*** cycle or the ***red-green-refactor*** cycle.



Advantages of TDD over the traditional approach :



- It builds your confidence when it comes to the code, as you know that the code written so far is working perfectly fine.
- It helps you achieve 100% test coverage.
- It helps you write minimal code to achieve the desired outcome.
- It helps you make your code cleaner without introducing bugs.
- It helps you find bugs at the earliest stage with the root cause for that bug instead of finding it at a later stage and debugging the code for several hours or days.
- It improves productivity. You are only writing the test cases and fixing the bugs instead of manually testing and debugging the application and then fixing the bugs; hence, skipping the debugging step.

Behaviour Driven Development

- BDD is only an extension of TDD.
- In BDD, you write test cases before writing the production code.
- It differs from TDD in terms of what we are testing and what type of test cases we are writing.
- In TDD, you test the written code by writing test cases for each method.
- In BDD, you test the **BEHAVIOR** of the written code by writing test cases for components or classes.
- For example, suppose you have an AuthenticationService class with several methods.
- This class would be used to perform authentication while logging in.

- In TDD, we will write test cases to check whether all the methods are working as desired.
- In BDD, we will write test cases to check whether the AuthenticationService is working as desired.
- In BDD as well, the test cases will be written for the methods. However, the motive behind those test cases would be to test the behavior of the component instead of testing the individual methods.

TDD Vs BDD

TDD

You start the development by writing test cases.
TDD focuses on code implementation.
Test cases are written from the developer's perspective.

BDD

You start the development by writing behaviors (same as TDD but the motive is different).
BDD focuses on code behavior.
Test cases are written from the end user's perspective.

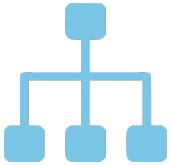
TDD	BDD
Test coverage is 100%, as every method is tested.	Test coverage may not be 100%, as every method may not be tested.
It is a better approach when other developers will be using your code, for example, writing libraries or developing frameworks.	It is a better approach when other users will be using your code, such as when creating websites or desktop applications.
Here, there are almost 0% chances of having bugs.	Here, the chances of having bugs are not 0%.

Introduction to the Testing Framework

Let us understand in detail about JEST.

What is Jest

Jest is a delightful JavaScript Testing Framework with a focus on simplicity. It is maintained by Facebook. It is fast and with minimum steps to set up.



Getting Started with Jest

How to install jest:

```
npm install --save-dev jest
```

Testing script in package.json file:

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


package.json
package.json > ...
{
"name": "testing",
"version": "1.0.0",
"description": "",
"main": "index.js",
Debug
"scripts": {
"test": "jest"
},
"author": "",
"license": "ISC",
"devDependencies": {
"jest": "^27.5.1"
}
}

Some Basic Test Cases:

Case 1: Testing Object

```
/**
 * Testing objects
 */
test('testing the objects' , ()=>{
    const data = {
        one :1
    };
    data.two = 2;

    expect(data).toEqual({two:2,one:1});  // Recursively checks all the fields of
the JS object
})
```

Case 2: Testing with wrong value:

```
/**
 * Testing not to be
 */
test('testing the sum method', ()=>{
    expect(sum(4,5)).not.toBe(7);
})
```

Case 3: Testing undefined, null and false:

```
 * Testing truthiness :
 * testing :
 * 1.undefined
 * 2.null
 * 3.false
 */

test('null', ()=>{
    let n = null;
    expect(n).toBeNull();
})

test('undefined', ()=>{
    let n = undefined;
    expect(n).toBeUndefined();
})
```

```
test('testing toBeTruthy and toBeFalsy', ()=>{
    let a= 7;
    let b = true;
    let c = undefined;
    expect(a).toBeTruthy();
    expect(b).toBeTruthy();
    expect(c).toBeFalsy();

})
```

Case 4: Testing numbers

```
/**
 * Testing numbers
 */

test('two plus two', () => {
   const value = 2 + 2;
   expect(value).toBeGreaterThan(3);
   expect(value).toBeGreaterThanOrEqual(3.5);
   expect(value).toBeLessThan(5);
   expect(value).toBeLessThanOrEqual(4.5);

   // toBe and toEqual are equivalent for numbers
   expect(value).toBe(4);
   expect(value).toEqual(4);
 });
```

Case 5: Testing string

```
/**
  * String testing
  */
  test('there is no I in team', () => {
   expect('team').not.toMatch(/I/);
 });

 test('but there is a "stop" in Christoph', () => {
   expect('Christoph').toMatch(/stop/);
 });
```

Case 6: Testing arrays and iterables

```
/**
  * Testing Arrays and Iterables
  */

  const shoppingList = [
    'diapers',
    'kleenex',
    'trash bags',
    'paper towels',
    'milk',
  ];

  test('the shopping list has milk on it', () => {
    expect(shoppingList).toContain('milk');
    expect(new Set(shoppingList)).toContain('milk');
  });
```

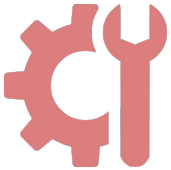
Case 7: Testing exceptions

```
/**
 * Testing exceptions
 */
function compileAndroidCode() {
  throw new Error('you are using the wrong JDK');
}

test('compiling android goes as expected', () => {
  expect(() => compileAndroidCode()).toThrow();
  expect(() => compileAndroidCode()).toThrow(Error);

  // You can also use the exact error message or a regexp
  expect(() => compileAndroidCode()).toThrow('you are using the wrong JDK');
  expect(() => compileAndroidCode()).toThrow(/JDK/);
});
```

Test Result:

```
PS D:\Mohit\Relevel\Backend\TestingUsingJest> npm test file1.test.js

> testing@1.0.0 test
> jest "file1.test.js"

 PASS  ./file1.test.js
  √ testing the objects (3 ms)
  √ testing the sum method (1 ms)
  √ null
  √ undefined
  √ undefined (1 ms)
  √ testing toBeTruthy and toBeFalsy
  √ two plus two (3 ms)
  √ there is no I in team (1 ms)
  √ but there is a "stop" in Christoph
  √ the shopping list has milk on it (1 ms)
  √ compiling android goes as expected (16 ms)

Test Suites: 1 passed, 1 total
Tests:       11 passed, 11 total
Snapshots:   0 total
Time:        0.65 s, estimated 1 s
Ran all test suites matching /file1.test.js/i.
PS D:\Mohit\Relevel\Backend\TestingUsingJest>
```

Setup and Teardown



There are many steps we have to do each time such as starting a database, adding, resetting the database as well.

Create new file named async.test.js and add below functions in it:

BeforeEach

```
/**
 * Running some setup before every test
 */
beforeEach(()=>{
    console.log("before every test this will be executed");
})
```

AfterEach

```
/**
 * Running some cleanup/resets after every test
 */
afterEach(()=>{
    console.log("after every test this will be executed");
})
```

BeforeAll

```
/**
 * Runnin once before the beginning of the file
 */
beforeAll(()=>{
    console.log("Before all");
})
```

AfterAll

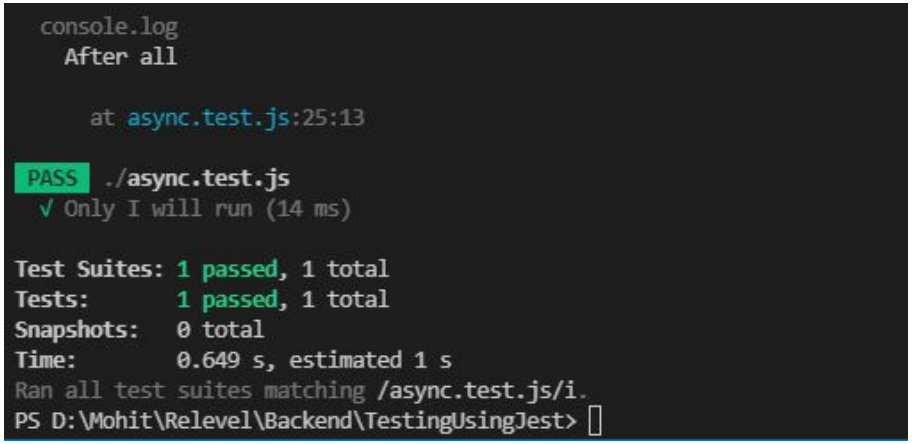
```
21  /**
22   * Runnin once at the end of the file
23   */
24  afterAll(()=>{
25      console.log("After all");
26  })
27
```

beforeEach and afterEach can handle asynchronous code in the same ways that tests can handle asynchronous code.

We can do some operations with beforeAll and afterAll as well.
It can be used for one time setup as well.

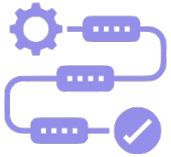
Test Result

```
PS D:\Mohit\Relevel\Backend\TestingUsingJest> npm test async.test.js

> testing@1.0.0 test
> jest "async.test.js"

  console.log
    Before all

      at async.test.js:18:13

  console.log
    before every test this will be executed

      at Object.<anonymous> (async.test.js:5:13)

  console.log
    Only this executes

      at Object.<anonymous> (async.test.js:81:13)

  console.log
    after every test this will be executed

      at Object.<anonymous> (async.test.js:11:13)
```

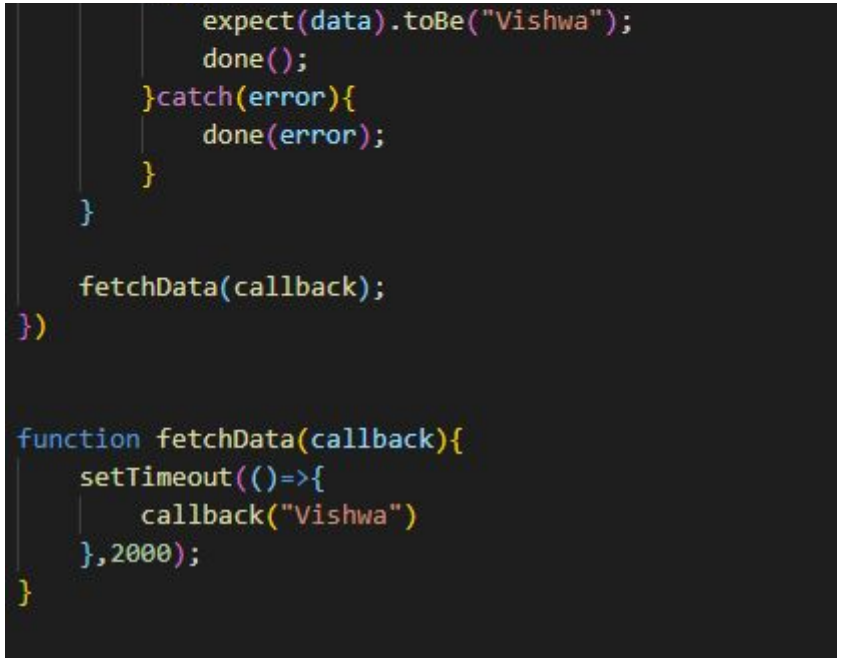

console.log
After all
at async.test.js:25:13
PASS ./async.test.js
√ Only I will run (14 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 0.649 s, estimated 1 s
Ran all test suites matching /async.test.js/i.
PS D:\Mohit\Relevel\Backend\TestingUsingJest>

Testing Callback function

We start by adding below function in async.test.js file



```
/**
 * Testing the call back function - improperly
 */
test("testing callback", ()=>{
    function callback(data){
        expect(data).toBe("Vishwa");
    }
    fetchData(callback);
})

/**
 *
 * Testing the callback function properly
 */
test("testing callback properly", done=>{
    function callback(data) {
        try{
```


expect(data).toBe("Vishwa");
done();
}catch(error){
done(error);
}
}
fetchData(callback);
})
function fetchData(callback){
setTimeout(()=>{
callback("Vishwa")
},2000);
}

Test Result

```
PS D:\Mohit\Relevel\Backend\TestingUsingJest> npm test async.test.js

> testing@1.0.0 test
Snapshots:   0 total
Time:        0.649 s, estimated 1 s
> testing@1.0.0 test
> jest "async.test.js"

  console.log
    Before all

      at async.test.js:18:13

  console.log
    before every test this will be executed

      at Object.<anonymous> (async.test.js:5:13)

  console.log
    after every test this will be executed

      at Object.<anonymous> (async.test.js:11:13)

  console.log
    before every test this will be executed
```

```
      at Object.<anonymous> (async.test.js:5:13)

  console.log
    after every test this will be executed

      at Object.<anonymous> (async.test.js:11:13)

  console.log
    After all

      at async.test.js:25:13

 PASS  ./async.test.js
  √ testing callback (9 ms)
  √ testing callback properly (2010 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        2.96 s
Ran all test suites matching /async.test.js/i.
PS D:\Mohit\Relevel\Backend\TestingUsingJest>
```

Testing Promise

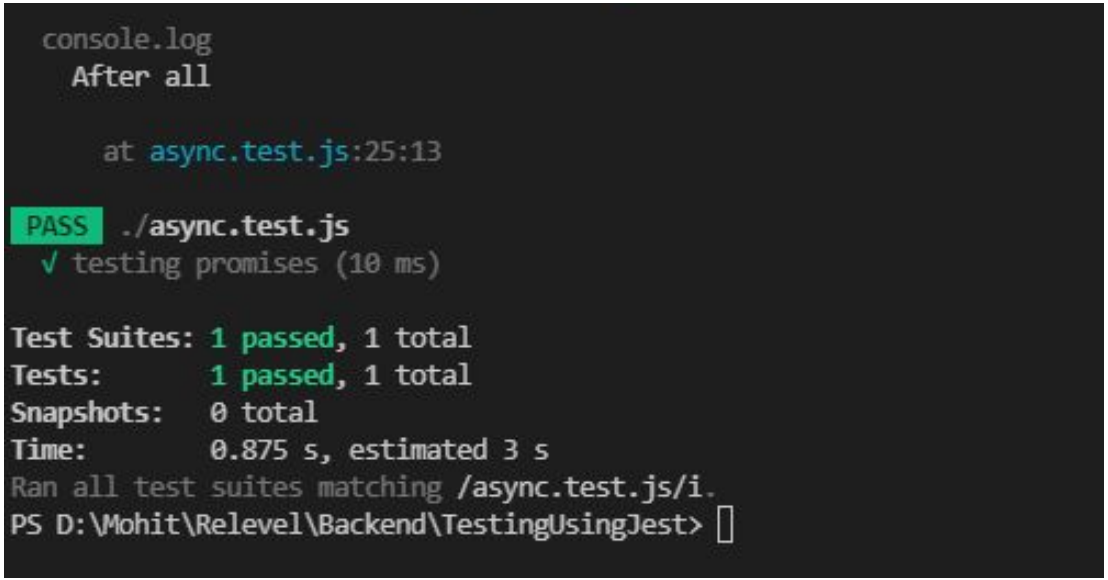
We will write the below code in async.test.js

```
/**
 * Testing promises
 */

function promiseFunc(){
    return new Promise((resolve, reject) =>{
          resolve("Vishwa");
    });
}


test("testing promises", ()=>{
    return promiseFunc().then((msg)=>{
        expect(msg).toBe("Vishwa");
    })
```

```
})

test.only("Only I will run", ()=>{
    console.log("Only this executes");
})
```

Test Result

```
PS D:\Mohit\Relevel\Backend\TestingUsingJest> npm test async.test.js

> testing@1.0.0 test
> jest "async.test.js"

  console.log
    Before all

      at async.test.js:18:13

  console.log
    before every test this will be executed

      at Object.<anonymous> (async.test.js:5:13)

  console.log
    after every test this will be executed

      at Object.<anonymous> (async.test.js:11:13)
```

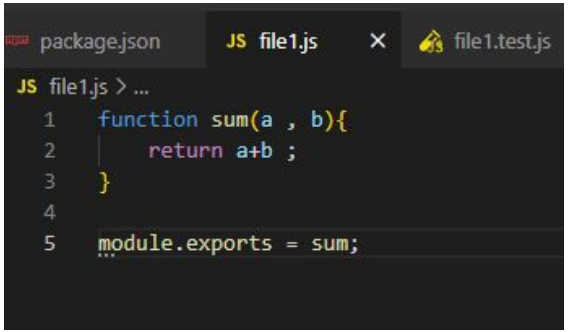

console.log
After all
at async.test.js:25:13
PASS ./async.test.js
√ testing promises (10 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 0.875 s, estimated 3 s
Ran all test suites matching /async.test.js/i.
PS D:\Mohit\Relevel\Backend\TestingUsingJest>

Testing by Importing a file

Here, we will understand how to test a method by importing the js file

Step 1: Create new file named file1.js and add a simple function in it as shown below:

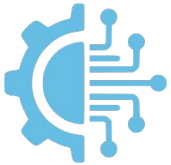
Step 2: Create test file named file1.test.js and write below code to test it.



Test Result

```
PS D:\Mohit\Relevel\Backend\TestingUsingJest> npm test file1.test.js

> testing@1.0.0 test
> jest "file1.test.js"

  console.log
    hello testing

      at Object.<anonymous> (file1.test.js:4:13)

 PASS  ./file1.test.js
  √ Hello testing (32 ms)
  √ testing the sum method (2 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.766 s, estimated 1 s
Ran all test suites matching /file1.test.js/i.
PS D:\Mohit\Relevel\Backend\TestingUsingJest>
```

Mock Functions



Sometimes it's impossible to test a function because it is dependent upon some other function. These functions determine the testing response. Also, to make it modular we have to separate out the function by mocking the behaviour of the dependent function.

How to do mocking:

Here we are trying to mock the callback.

Step 1: Create the function in new file named mock.test.js file

Step 2: Create a mock function to be used while testing

Step 3: Create the test function and use mock function in it.

Step 4: Testing mock with return value

mock.test.js > ...

```
/***
 * Using a mock function
 */
function forEach(items, callback){
    for(let index =0;index <items.length;index++){
        callback(items[index]);
    }
}

/**
 * In order to test the above function, we can mock the
 callback function
 * to test if the callback is invoked
 */
const mockCallback = jest.fn(x=>x+7);
forEach([0,1],mockCallback);

test("testing the mock function", ()=>{
    //check if the mockfunction is called twice
    expect(mockCallback.mock.calls.length).toBe(2);

    //Testing the outcome of the first call of the mock
    function
    expect(mockCallback.mock.results[0].value).toBe(7);

```

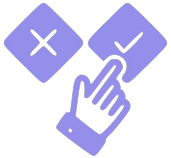
```
24      //Test the first argument of the second call of the mock
        function
25      expect(mockCallback.mock.calls[1][0]).toBe(1);
26  })
27
28  /**
29   * Mocking the return value
30   */
31  const myMock = jest.fn();
32  console.log(myMock());
33
34  myMock.mockReturnValueOnce(10).mockReturnValueOnce('X').
    mockReturnValueOnce(true).mockReturnValueOnce('Vishwa');
35
36  console.log(myMock(), myMock(), myMock(), myMock());
```

Test Result

```
PS D:\Mohit\Relevel\Backend\TestingUsingJest> npm test mock.test.js

> testing@1.0.0 test
> jest "mock.test.js"

  console.log
    undefined

      at Object.<anonymous> (mock.test.js:32:10)

  console.log
    10 X true Vishwa

      at Object.<anonymous> (mock.test.js:36:10)

 PASS  ./mock.test.js
  √ testing the mock function (3 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.758 s, estimated 3 s
Ran all test suites matching /mock.test.js/i.
PS D:\Mohit\Relevel\Backend\TestingUsingJest>
```

MCQs

1. Which command do we use to run tests in a node application?



A. npm test
B. npm run test
C. npm start test
D. npm run jest

2. Which one is true about TTD?

A. It’s mainly focused on the behaviour of code.
B. Here, test cases are written from the end user’s perspective.
C. The chances of having bugs here is not 0%.
D. The chances of having bugs are almost 0%.

3. Which of the following is not a unit testing framework?

A. Mocha
B. Jest
C. AVA
D. uuid

4. Jest uses _____ and _____ matchers to test a value with exact quality.

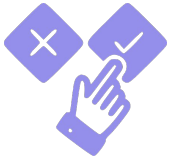
A. toBeTruthy, toBeFalsy

B. toBeEqual, toMatch

C. toBe, toEqual

D. toBeUndefined, toBeDefined

5. Jest will wait until the _____ callback is called before finishing the test.



A. finish

B. do

C. done

D. None of the options

Practice Problems

A. Create a promise and callback functions and also their test function and test it.
B. Create another file with a function in it and also it’s test file and run the test.

Next session

Unit Testing of the eCommerce Application

THANK YOU