Lab Manual- 3

Unit Testing

Unit testing is a common practice where developers write test cases together with regular code. Developers develop the products using programming languages such as Java, JavaScript, C#, and so on. As JavaScript is familiar to all, we write and test our unit testing using JavaScript. Besides, for every programming language, there are many testing frameworks. Here, For JavaScript, we use the Jest testing framework. Jest is a JavaScript testing framework designed to ensure the correctness of any JavaScript codebase. It allows you to write tests with an approachable, familiar, and feature-rich API that gives you results quickly. Jest is well-documented, requires little configuration, and can be extended to match your requirements.

To understand unit testing, we consider an example of a Calculator. Assume, there are two types of calculators, Basic and Advanced calculators. The basic calculator has the following functionalities-

- Add(a, b): It takes two numbers as input and returns the summation (a+b) of these two numbers.
- Subtract(a, b): It takes two numbers as input and returns the subtraction (a-b) of these two numbers.
- Multiply(a, b): It takes two numbers as input and returns the multiplication value (a*b) of these two numbers.
- Divide(a, b): It takes two numbers, dividend and divisor as input and returns the quotient (a/b) of these two numbers.

The advanced calculator has the following functionalities-

- Pow(x, n): It takes two numbers as input and returns the powered value (x^n) of these two numbers.
- Modulo(a, b): It takes two numbers as input and returns the modulo value (a%b) of these two numbers.

To test the calculator, at first, we have to implement the Basic and Scientific calculator. As mentioned earlier, we will implement the calculator using JavaScript. Before implanting it, we need to install some libraries.

Prerequisite:

- Implementation IDE: For writing code, you can use any IDE. Here, we will use VSCode IDE. Download it from this link and install the .exe.
- Node JS: Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser. Download it from this link and install the .msi. Check the version of Node JS

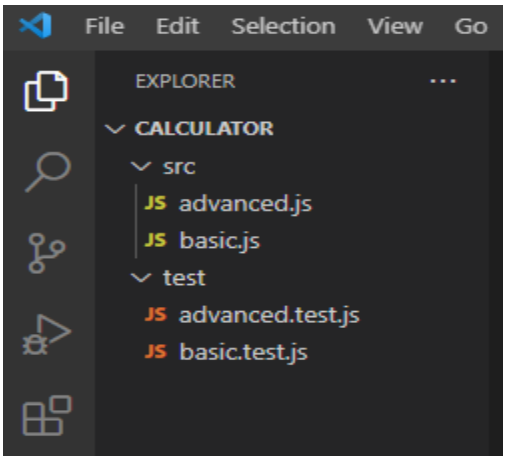
using ***node –v*** command. Also, you can check the version of npm using ***npm –v*** command.

- JEST: Jest is a testing framework. Install it using the following command from the terminal.

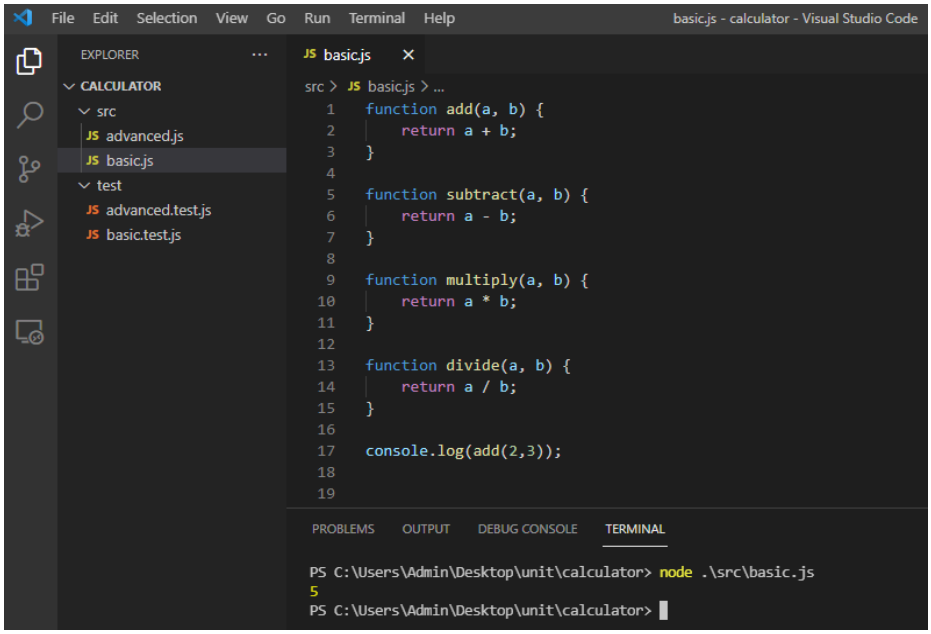
```
npm install --save-dev jest
```

Environment setup:

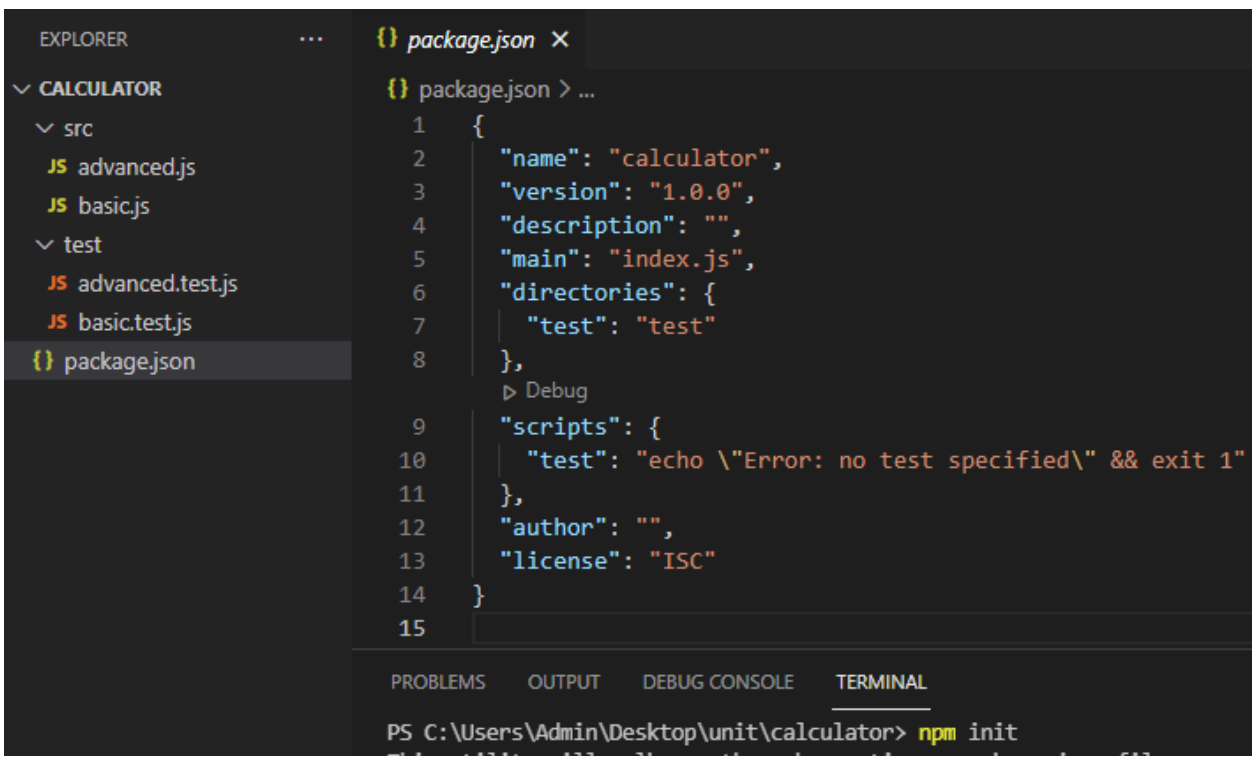
Now, the environment is ready to implement the calculator. At first, a folder named calculator is created to organize the whole project. Then, inside the calculator folder, two folders are created, such as src and test. Now, two .js files are created for the implementation of the functional requirements inside the src folder. In the same way, two .test.js files are created for testing purposes inside the test folder.



At first, we implement the basic calculator functionalities.



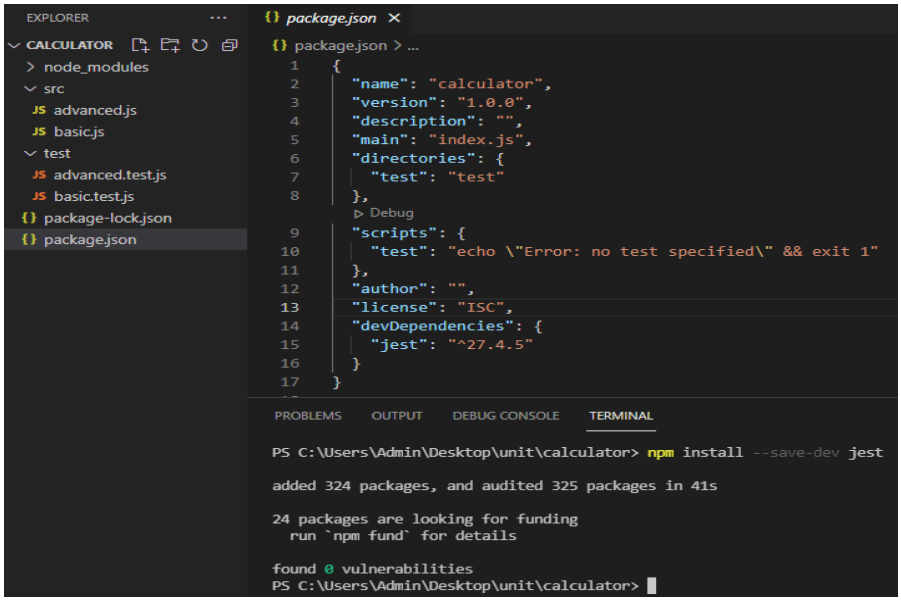
Now, from the calculator folder, node modules are needed to be initiated by typing `npm init` from the command line and clicking ENTER with the default value. After that, a package.json file is created, which looks like the following.



Now, we will install the JEST using the following command line.

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

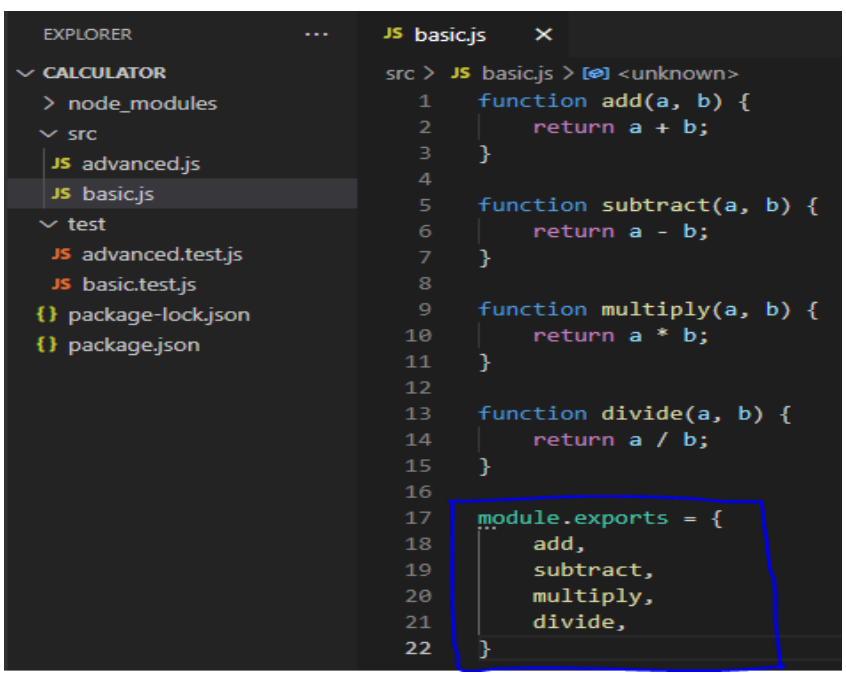
After that, the folder structure looks like the following-



Here, the package.json file is also updated by adding JEST dependency. Also, a package-lock.json file is added where all the information about the node-modules folder is recorded.

Test the basic.js file:

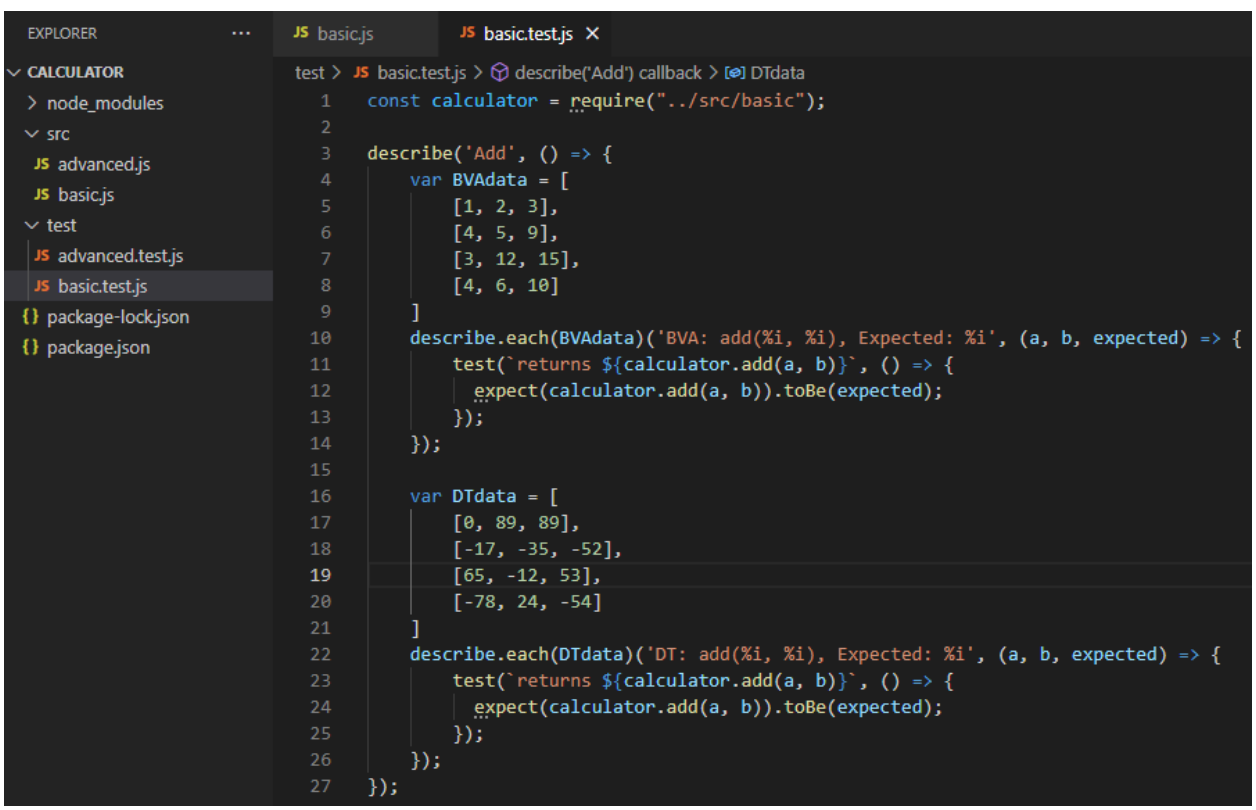
To test the basic calculator, we implement the basic.test.js file. Here, we follow JEST terminology to test individual functions having different types of test cases. Before implementing the test file, first, we have to export the methods of the basic calculator.



As mentioned in the theory class, there are different types of testing such as boundary value analysis (BVA), decision table (DT) based testing, and so on. Every type of testing has several test cases that need to be tested. For example, considering the add function, a sample test case can be the following-

Add			
Method Name	a	b	Expected
BVA	1	2	3
	4	5	9
	3	12	15
	4	6	10
DT	0	89	89
	-17	-35	-52
	65	-12	53
	-78	24	-54

Here are the implementation details to test the add function of the basic calculator.

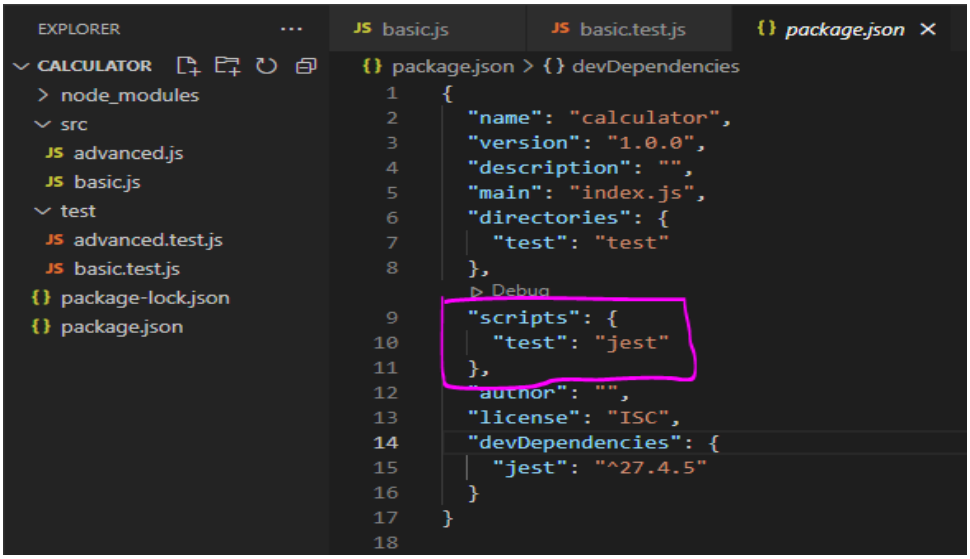



```
const calculator = require("../src/basic");

describe('Add', () => {
    var BVAdata = [
        [1, 2, 3],
        [4, 5, 9],
        [3, 12, 15],
        [4, 6, 10]
    ]
    describe.each(BVAdata)('BVA: add(%i, %i), Expected: %i', (a, b, expected) => {
        test(`returns ${calculator.add(a, b)}`, () => {
          expect(calculator.add(a, b)).toBe(expected);
        });
    });

    var DTdata = [
        [0, 89, 89],
        [-17, -35, -52],
        [65, -12, 53],
        [-78, 24, -54]
    ]
    describe.each(DTdata)('DT: add(%i, %i), Expected: %i', (a, b, expected) => {
        test(`returns ${calculator.add(a, b)}`, () => {
          expect(calculator.add(a, b)).toBe(expected);
        });
    });
});
```

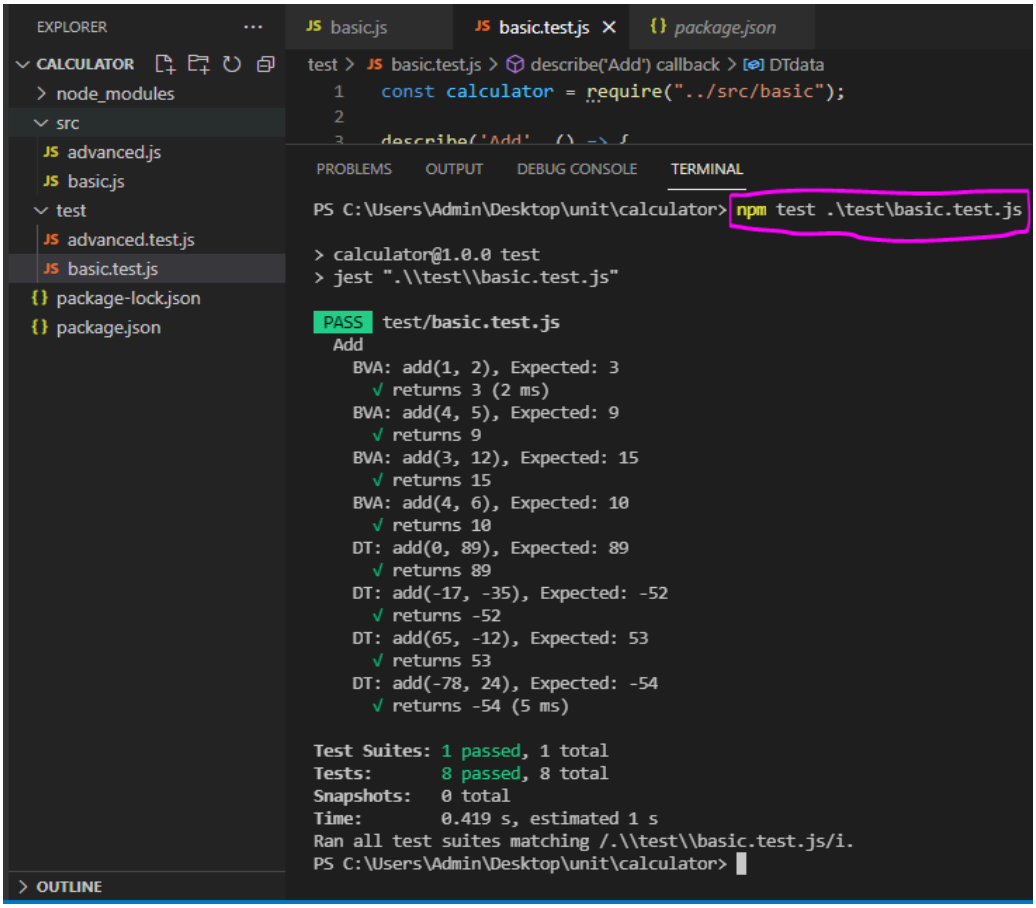
Now, it’s time to test our first add function with two testing methods, having several test cases. Before running the test, we have to add the JEST in the package.json file by updating the script's member.



As we want to test the only basic.test.js file, the following command is needed to be executed.

```
:\calculator> npm test .\test\basic.test.js
```

The output of the test cases are following-



Here, there are 8 test cases and each of the test cases is passed.

However, if the test cases are huge, it will be difficult for us to comprehend all the results from the terminal (console). So, it is needed to generate a report considering all the test cases. Here, we will generate two types of reports-

Test Coverage Report:

The test coverage report is generated using lcov and text reporters. To do this, we need to add the following code snippet in the package.json file.

```
EXPLORER
CALCULATOR
  coverage
  node_modules
  src
    advanced.js
    basic.js
  test
    advanced.test.js
    basic.test.js
  package-lock.json
  package.json

basic.js | basic.test.js | package.json
package.json > jest > collectCoverage

4   "description": "",
5   "main": "index.js",
6   "directories": {
7     "test": "test"
8   },
    Debug
9   "scripts": {
10    "test": "jest"
11  },
12  "jest": {
13    "collectCoverage": true
14  },
15  "author": "",
16  "license": "ISC",
17  "devDependencies": {
18    "jest": "^27.4.5"
19  }
20 }
21
```

After that, run the previous `npm test` command again.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Users\Admin\Desktop\unit\calculator> npm test .\test\basic.test.js

> calculator@1.0.0 test
> jest ".\\test\\basic.test.js"

 PASS  test/basic.test.js
  Add
    BVA: add(1, 2), Expected: 3
      √ returns 3 (2 ms)
    BVA: add(4, 5), Expected: 9
      √ returns 9 (1 ms)
    BVA: add(3, 12), Expected: 15
      √ returns 15 (1 ms)
    BVA: add(4, 6), Expected: 10
      √ returns 10
    DT: add(0, 89), Expected: 89
      √ returns 89
    DT: add(-17, -35), Expected: -52
      √ returns -52
    DT: add(65, -12), Expected: 53
      √ returns 53
    DT: add(-78, 24), Expected: -54
      √ returns -54

----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |      40 |      100 |      25 |      40 |
 basic.js |      40 |      100 |      25 |      40 | 6-14
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
```

Here, in addition to the previous output, we have found some interesting stats such as how many statements are covered by executing this test file. However, the previous problem (result is shown in console) exists. We want to generate a report separately. To do so, we need to add two more lines to the package.json file.

```
JS basic.js    JS basic.test.js    {} package.json ×
{} package.json > ...
4     "description": "",
5     "main": "index.js",
6     "directories": {
7       "test": "test"
8     },
        ▷ Debug
9     "scripts": {
10      "test": "jest"
11    },
12    "jest": {
13      "collectCoverage": true,
14      "coverageReporters": ["lcov"],
15      "coverageDirectory": "./code-coverage-report/"
16    },
17    "author": "",
18    "license": "ISC",
19    "devDependencies": {
20      "jest": "^27.4.5"
21    }
22  }
23
```

Now, we run the same command (`npm test`) again. Now, a folder named code-coverage-report is created, where all the information related to code coverage is reported in an HTML file (index.html).

However, we want to generate a report considering the statistics of test cases, such as how many test cases are passed or failed. To do this, we need jest-html-reported which is described below section.

HTML Report:

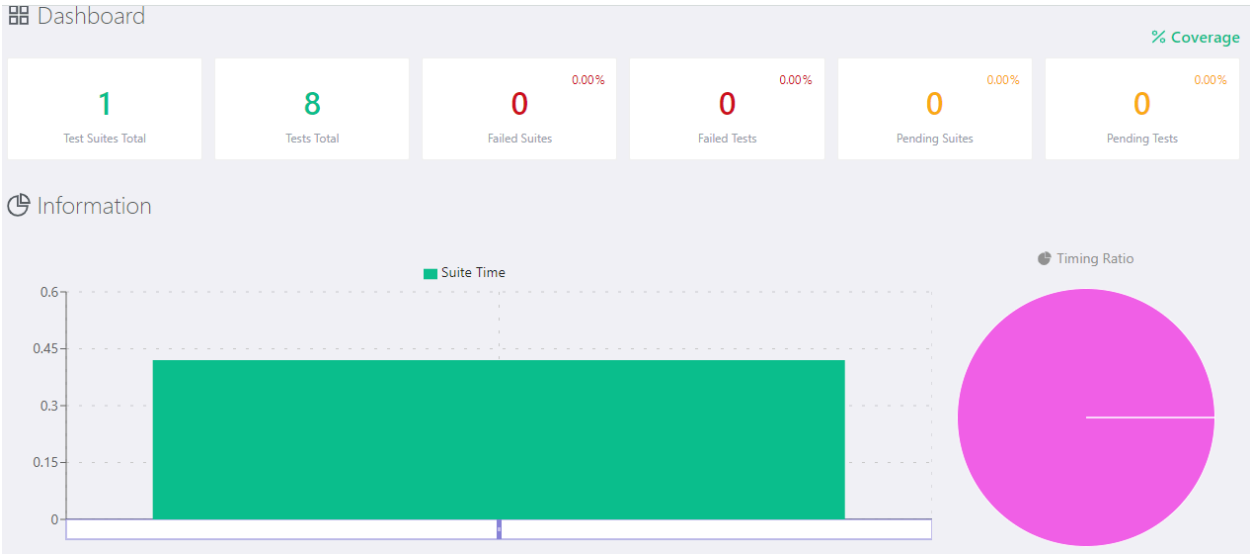
HTML report is generated using jest-html-reporter package. So, at first, we have to install it by the following command.

```
npm install --save-dev jest-html-reporters
```

Now, we need to add this into package.json file.

```
basic.js    basic.test.js    package.json
package.json > ...
1  {
2    "name": "calculator",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "directories": {
7      "test": "test"
8    },
     Debug
9    "scripts": {
10     "test": "jest"
11   },
12   "jest": {
13     "collectCoverage": true,
14     "coverageReporters": [
15       "lcov"
16     ],
17     "coverageDirectory": "./code-coverage-report/",
18     "reporters": ["default", "./node_modules/jest-html-reporters"]
19   },
20   "author": "",
21   "license": "ISC",
22   "devDependencies": {
23     "jest": "^27.4.5",
24     "jest-html-reporters": "^2.1.6"
25   }
26 }
27
```

After that, run the previous `npm test` command again. Now, we will see an HTML file (jest_html_reporters.html) where a dashboard will be seen. Here, all the information related to test cases is shown.



Details

Expand All

File	UseTime	Status	Action
C:\Users\Admin\Desktop\unit\calculator\test\basic.test.js	00:00.420	All Passed 8	
Add > BVA: add(1, 2), Expected: 3 > returns 3	00:00.002	Passed	
Add > BVA: add(4, 5), Expected: 9 > returns 9	00:00.000	Passed	
Add > BVA: add(3, 12), Expected: 15 > returns 15	00:00.002	Passed	
Add > BVA: add(4, 6), Expected: 10 > returns 10	00:00.000	Passed	
Add > DT: add(0, 89), Expected: 89 > returns 89	00:00.000	Passed	
Add > DT: add(-17, -35), Expected: -52 > returns -52	00:00.000	Passed	
Add > DT: add(65, -12), Expected: 53 > returns 53	00:00.000	Passed	
Add > DT: add(-78, 24), Expected: -54 > returns -54	00:00.001	Passed	

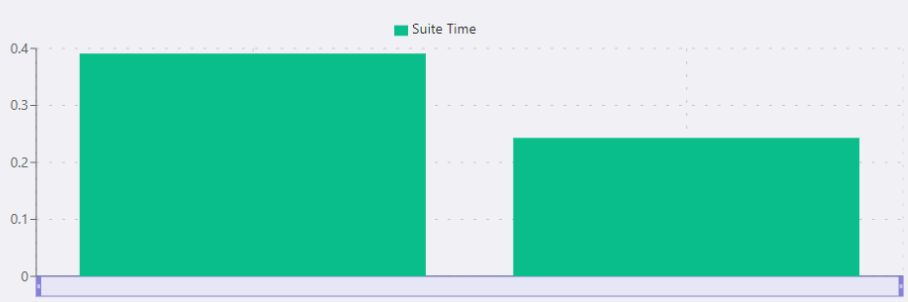
That's it. Now, we can easily add another testing method of the basic calculator into the test file. After implementing the advanced calculator and corresponding file, the output will be the following.

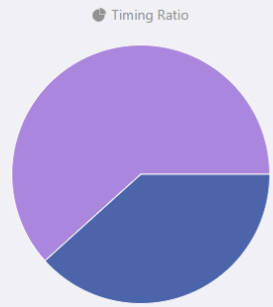
Dashboard

% Coverage

2	24	100.00% 2	12.50% 3	0.00% 0	0.00% 0
Test Suites Total	Tests Total	Failed Suites	Failed Tests	Pending Suites	Pending Tests

Information





By clicking the info button, you will find the error message in detail.

Tasks:

In the same way, **implement and test** all the functions of basic and advanced calculators.

- Add(a, b): It takes two numbers as input and returns the summation (a+b) of these two numbers.
- Subtract(a, b): It takes two numbers as input and returns the subtraction (a-b) of these two numbers.
- Multiply(a, b): It takes two numbers as input and returns the multiplication value (a*b) of these two numbers.
- Divide(a, b): It takes two numbers, dividend and divisor as input and returns the quotient (a/b) of these two numbers.

The advanced calculator has the following functionalities-

- Pow(x, n): It takes two numbers as input and returns the powered value (x^n) of these two numbers.
- Modulo(a, b): It takes two numbers as input and returns the modulo value (a%b) of these two numbers.