

## 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

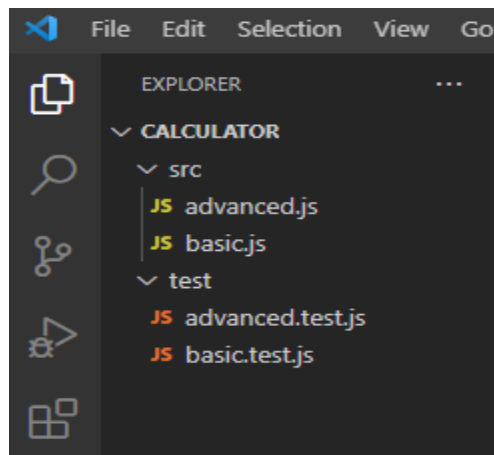
using **node -v** command. Also, you can check the version of npm using **npm -v** command.

- 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.

```
File Edit Selection View Go Run Terminal Help basic.js - calculator - Visual Studio Code

EXPLORER
  CALCULATOR
    src
      JS advanced.js
      JS basicjs
    test
      JS advanced.test.js
      JS basic.test.js

src > JS basicjs > ...
1 function add(a, b) {
2   return a + b;
3 }
4
5 function subtract(a, b) {
6   return a - b;
7 }
8
9 function multiply(a, b) {
10  return a * b;
11 }
12
13 function divide(a, b) {
14  return a / b;
15 }
16
17 console.log(add(2,3));
18
19

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Admin\Desktop\unit\calculator> node .\src\basic.js
5
PS C:\Users\Admin\Desktop\unit\calculator>
```

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.

```
EXPLORER
  CALCULATOR
    src
      JS advanced.js
      JS basic.js
    test
      JS advanced.test.js
      JS basic.test.js
    {} package.json

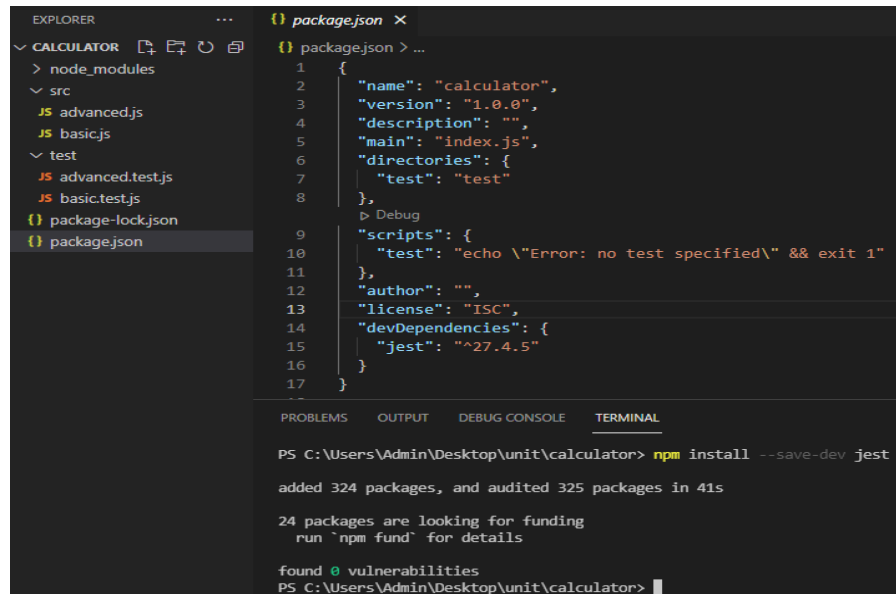
{} package.json > ...
1 {
2   "name": "calculator",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "directories": {
7     "test": "test"
8   },
9   "scripts": {
10    "test": "echo \"Error: no test specified\" && exit 1"
11  },
12   "author": "",
13   "license": "ISC"
14 }
15

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Admin\Desktop\unit\calculator> npm init
```

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-



The screenshot shows the VS Code interface. On the left, the Explorer sidebar displays the project structure for 'CALCULATOR':

- node\_modules
- src
  - advanced.js
  - basic.js
- test
  - advanced.test.js
  - basic.test.js
- package-lock.json
- package.json

The main editor shows the 'package.json' file with the following content:

```
1 {
2   "name": "calculator",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "directories": {
7     "test": "test"
8   },
9   "scripts": {
10    "test": "echo \\\"Error: no test specified\\\" && exit 1"
11  },
12   "author": "",
13   "license": "ISC",
14   "devDependencies": {
15     "jest": "^27.4.5"
16   }
17 }
```

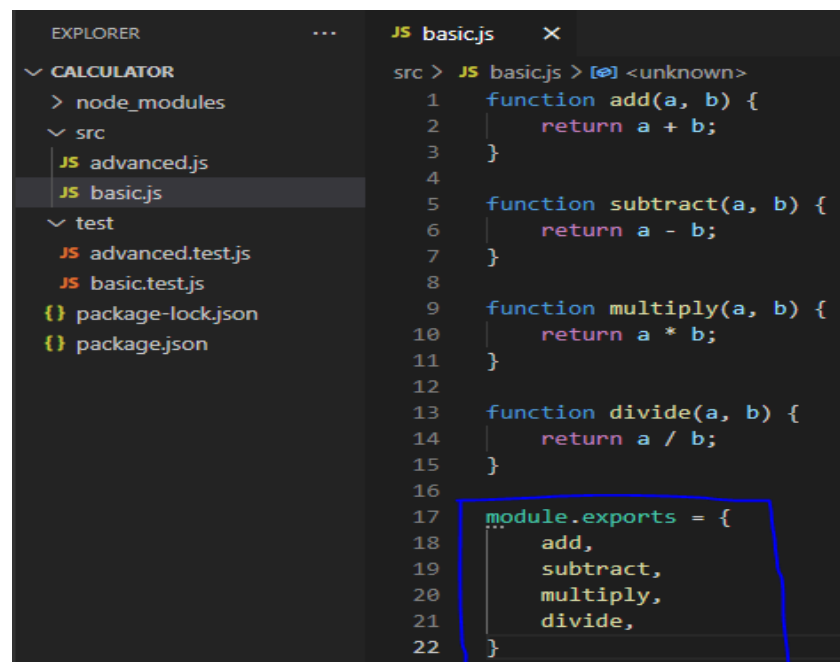
At the bottom, the Terminal panel shows the output of the command `npm install --save-dev jest`:

```
PS C:\Users\Admin\Desktop\unit\calculator> npm install --save-dev jest
added 324 packages, and audited 325 packages in 41s
24 packages are looking for funding
run `npm fund` for details
found 0 vulnerabilities
PS C:\Users\Admin\Desktop\unit\calculator>
```

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.



The screenshot shows the VS Code interface with the 'basic.js' file open in the editor. The Explorer sidebar on the left shows the project structure, with 'basic.js' selected under the 'src' folder.

The main editor displays the code for 'basic.js':

```
src > JS basic.js > [?] <unknown>
1  function add(a, b) {
2    |   return a + b;
3  }
4
5  function subtract(a, b) {
6    |   return a - b;
7  }
8
9  function multiply(a, b) {
10   |   return a * b;
11 }
12
13 function divide(a, b) {
14   |   return a / b;
15 }
16
17 module.exports = {
18   |   add,
19   |   subtract,
20   |   multiply,
21   |   divide,
22 }
```

The `module.exports` object is highlighted with a blue box.

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.

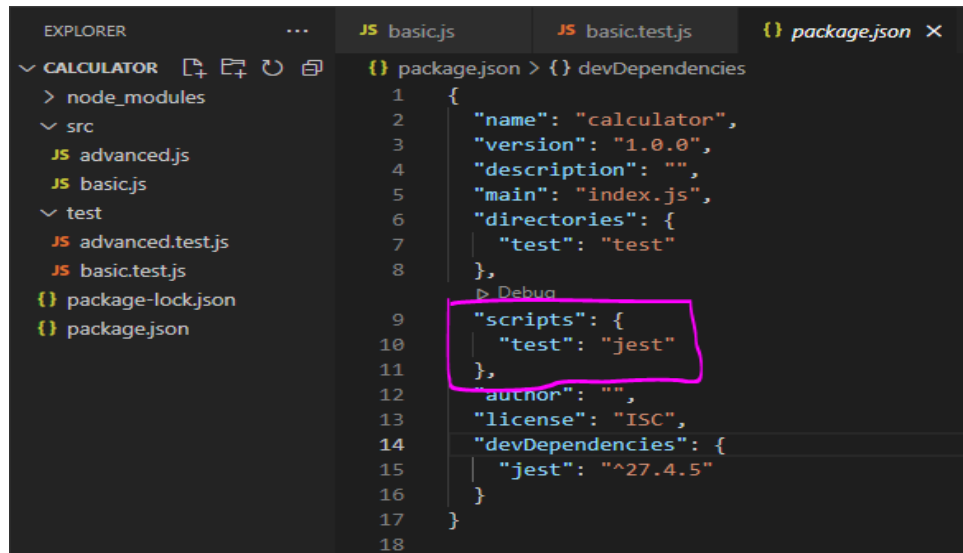
```

EXPLORER    ...    JS basic.js    JS basic.test.js X
└─ CALCULATOR
  └─ node_modules
  └─ src
    └─ JS advanced.js
    └─ JS basic.js
    └─ test
      └─ JS advanced.test.js
      └─ JS basic.test.js
      └─ {} package-lock.json
      └─ {} package.json

test > JS basic.test.js > describe('Add') callback > [x] DTdata
1  const calculator = require("../src/basic");
2
3  describe('Add', () => {
4    var BVAdata = [
5      [1, 2, 3],
6      [4, 5, 9],
7      [3, 12, 15],
8      [4, 6, 10]
9    ]
10   describe.each(BVAdata)('BVA: add(%i, %i), Expected: %i', (a, b, expected) => {
11     test(`returns ${calculator.add(a, b)}`, () => {
12       expect(calculator.add(a, b)).toBe(expected);
13     });
14   });
15
16   var DTdata = [
17     [0, 89, 89],
18     [-17, -35, -52],
19     [65, -12, 53],
20     [-78, 24, -54]
21   ]
22   describe.each(DTdata)('DT: add(%i, %i), Expected: %i', (a, b, expected) => {
23     test(`returns ${calculator.add(a, b)}`, () => {
24       expect(calculator.add(a, b)).toBe(expected);
25     });
26   });
27 });

```

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.



The screenshot shows the VS Code interface with the Explorer on the left and the package.json file open in the editor. The Explorer shows a project named 'CALCULATOR' with a 'test' directory containing 'advanced.test.js' and 'basic.test.js'. The package.json file is open, and the 'scripts' section is highlighted with a pink box. The 'scripts' section contains a 'test' script set to 'jest'. The 'devDependencies' section also lists 'jest' as a dependency.

```
1 {
2   "name": "calculator",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "directories": {
7     "test": "test"
8   },
9   "scripts": {
10    "test": "jest"
11  },
12  "author": "",
13  "license": "ISC",
14  "devDependencies": {
15    "jest": "^27.4.5"
16  }
17 }
```

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-

```
test > JS basic.test.js > describe('Add') callback > DTdata
1   const calculator = require("../src/basic");
2
3   describe('Add' () => {
    > 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
        BVA: add(3, 12), Expected: 15
        ✓ returns 15
        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 (5 ms)

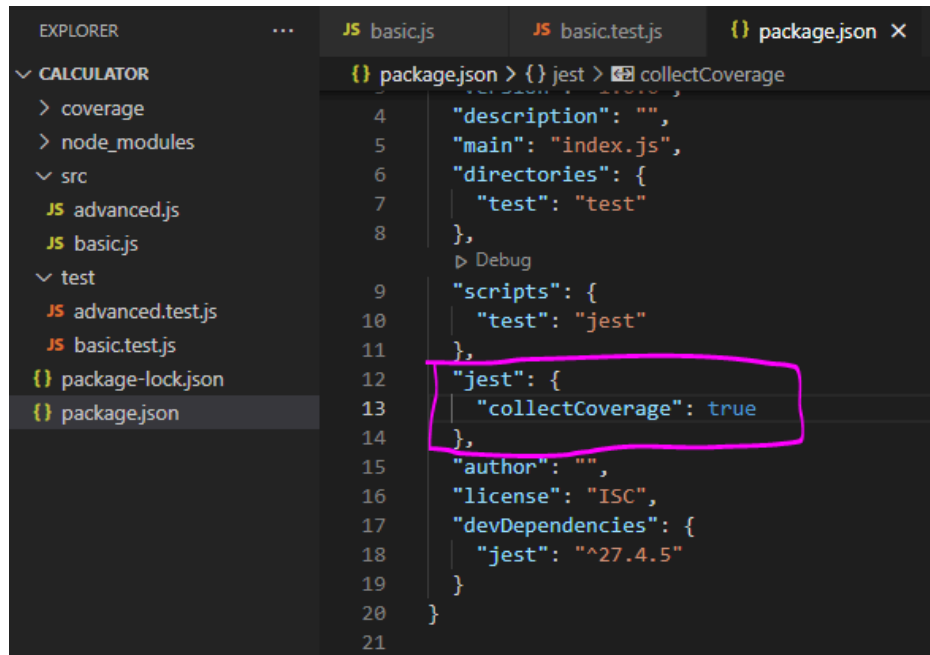
    Test Suites: 1 passed, 1 total
    Tests:      8 passed, 8 total
    Snapshots:  0 total
    Time:       0.419 s, estimated 1 s
    Ran all test suites matching /\.\\test\\basic.test.js/i.
    PS C:\Users\Admin\Desktop\unit\calculator>
```

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:

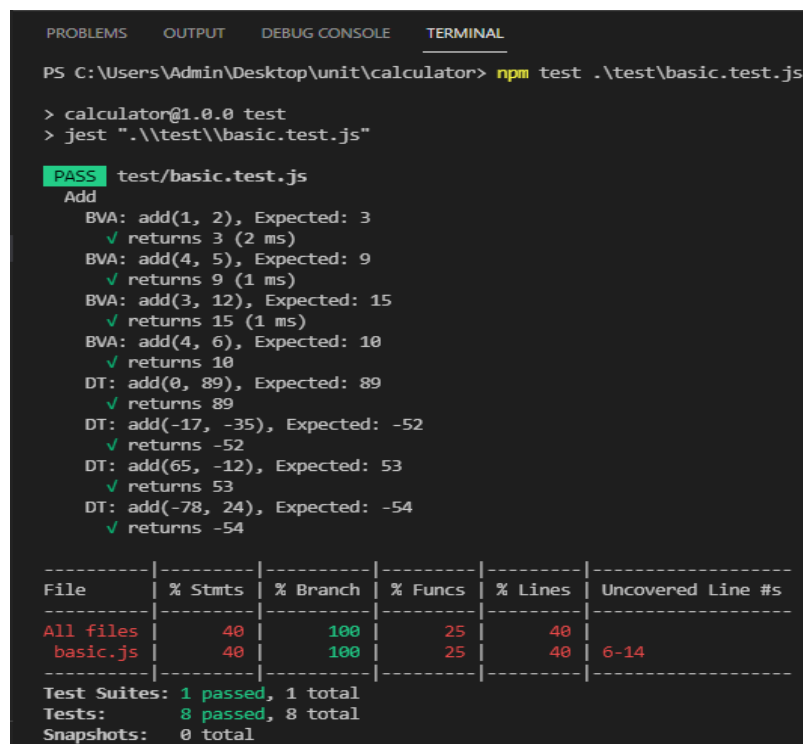
The test coverage report is generated using lcov and text reporters. To do this, we need to add the following code snippet in the package.json file.



The screenshot shows the VS Code interface with the Explorer sidebar on the left displaying a project structure. The main editor shows the `package.json` file. A pink rectangle highlights the `"jest": { "collectCoverage": true }` section. The file content is as follows:

```
1  {
2    "name": "calculator",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "directories": {
7      "test": "test"
8    },
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 }
```

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



The screenshot shows the terminal output of the `npm test` command. It displays the test results for `test/basic.test.js`, including a list of tests and their expected/actual values. Below the test results, a table provides coverage statistics for the files.

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

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

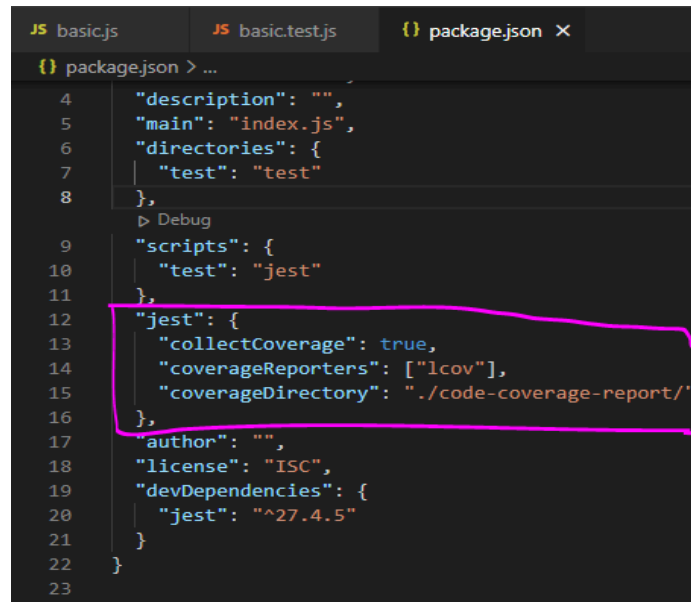
PASS test/basic.test.js
  Add
    BVA: add(1, 2), Expected: 3
    ✓ returns 3 (2 ms)
    BVA: add(4, 5), Expected: 9
    ✓ returns 9 (1 ms)
    BVA: add(3, 12), Expected: 15
    ✓ returns 15 (1 ms)
    BVA: add(4, 6), Expected: 10
    ✓ returns 10
    DT: add(0, 89), Expected: 89
    ✓ returns 89
    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 X
{} package.json > ...
4   "description": "",
5   "main": "index.js",
6   "directories": {
7     "test": "test"
8   },
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-reporter` 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
```

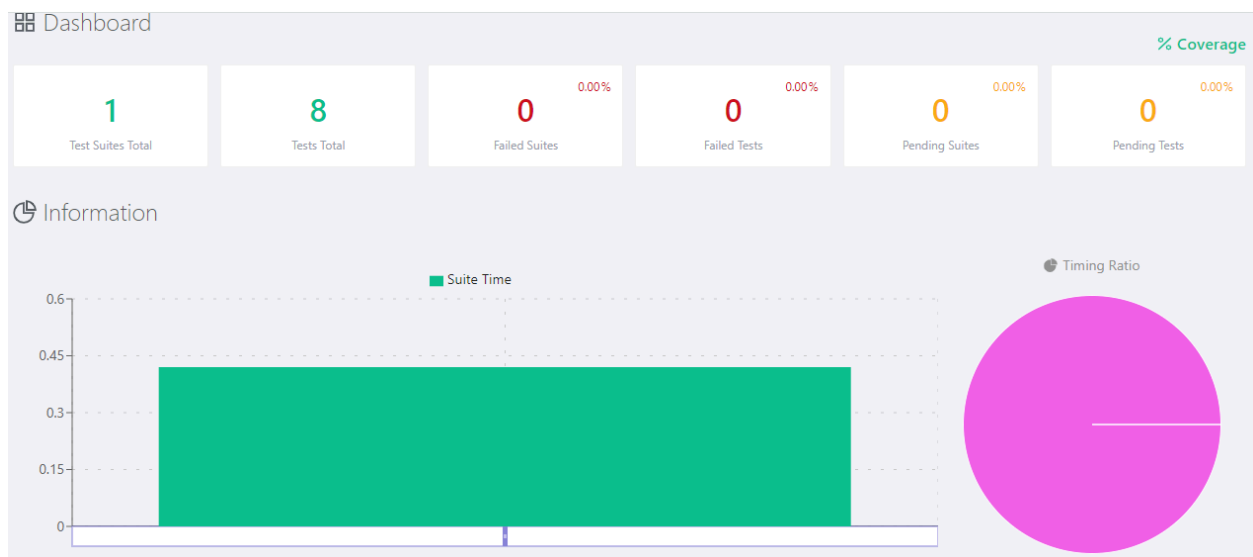
Now, we need to add this into `package.json` file.

```

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

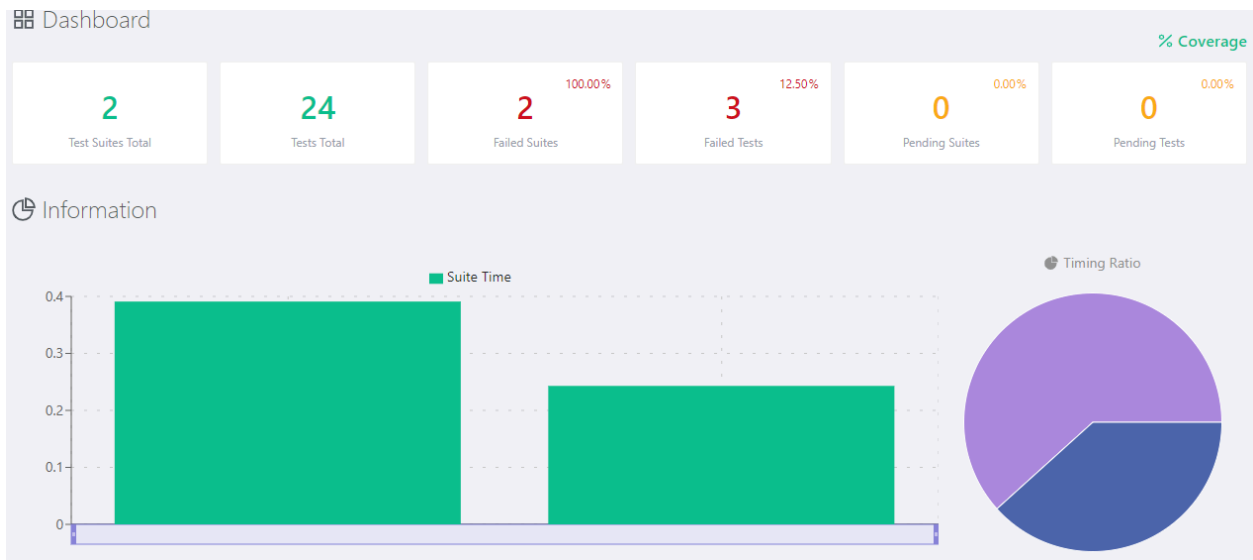


Details

Expand All ☒

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



File	UseTime	Status	Action
C:\Users\Admin\Desktop\user\calculator\test\advanced.test.js	0:00:00.001	✓ Passed	1 <a href="#">info</a>
Pow > Pow(1, 2), Expected: 1 > returns 1	0:00:00.000	✓ Passed	
Pow > Pow(4, 5), Expected: 1024 > returns 1024	0:00:00.000	✓ Passed	
Pow > Pow(10, 12), Expected: 531441 > returns 531441	0:00:00.000	✓ Passed	
Pow > Pow(4, 6), Expected: 4096 > returns 4096	0:00:00.000	✓ Passed	
Pow > DT: pow(5, 89), Expected: 0 > returns 0	0:00:00.000	✓ Passed	
Pow > DT: pow(17, 4), Expected: 85521 > returns 85521	0:00:00.000	✓ Passed	
Pow > DT: pow(6, 7), Expected: 46522789624 > returns 46522789625	0:00:00.000	✗ Failed	<a href="#">info</a>
Pow > DT: pow(76, 6), Expected: 22519960704 > returns 22519960704	0:00:00.000	✓ Passed	
C:\Users\Admin\Desktop\user\calculator\test\basic.test.js	0:00:00.049	✓ Passed	2 <a href="#">info</a>
Merge Data			
Add > Add(1, 2), Expected: 3 > returns 3	0:00:00.001	✓ Passed	
Add > Add(4, 5), Expected: 9 > returns 9	0:00:00.001	✓ Passed	
Add > Add(10, 12), Expected: 22 > returns 22	0:00:00.001	✓ Passed	
Add > Add(4, 6), Expected: 10 > returns 10	0:00:00.000	✓ Passed	
Add > DT: add(5, 89), Expected: 89 > returns 89	0:00:00.000	✓ Passed	
Add > DT: add(17, 35), Expected: 52 > returns 52	0:00:00.000	✓ Passed	
Add > DT: add(5, 10), Expected: 55 > returns 53	0:00:00.000	✗ Failed	<a href="#">info</a>
Add > DT: add(76, 24), Expected: 100 > returns 104	0:00:00.000	✓ Passed	
Subtract > Sub(1, 2), Expected: -1 > returns -1	0:00:00.001	✓ Passed	
Subtract > Sub(4, 5), Expected: -1 > returns -1	0:00:00.000	✓ Passed	
Subtract > Sub(10, 12), Expected: -2 > returns -2	0:00:00.000	✓ Passed	
Subtract > Sub(4, 6), Expected: -2 > returns -2	0:00:00.000	✓ Passed	
Subtract > DT: subtract(5, 89), Expected: 89 > returns 89	0:00:00.000	✓ Passed	
Subtract > DT: subtract(17, 35), Expected: 18 > returns 18	0:00:00.000	✓ Passed	
Subtract > DT: subtract(55, 12), Expected: 77 > returns 77	0:00:00.001	✓ Passed	
Subtract > DT: subtract(76, 24), Expected: -102 > returns -102	0:00:00.000	✓ Passed	

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.