Unit Testing With JEST

Jest is a JavaScript testing framework developed by Facebook, designed primarily for unit testing React applications. However, it can be used for testing any JavaScript codebase.

Jest is known for its simplicity, speed, and built-in features, making it one of the most popular choices for testing JavaScript applications.

Prerequisites

- Node.js (LTS) installed and available on command line (node -v).
- npm or yarn available (npm -v or yarn -v).
- VS Code (recommended) for editing and debugging.
- A project folder initialized with package.json .

Installation Steps

- Change directory to your project folder:

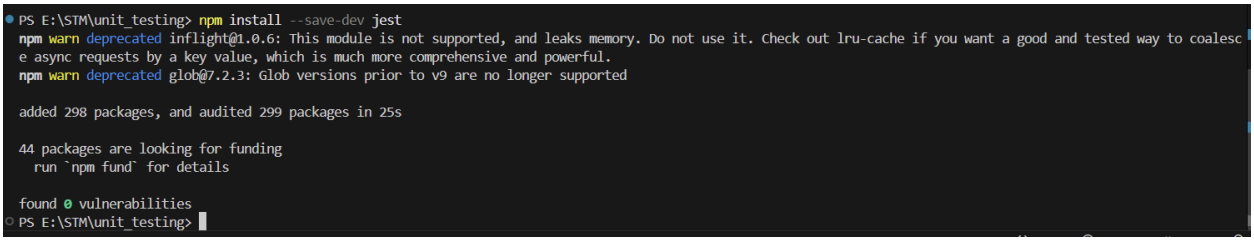
```
PS E:\> cd E:\STM\unit_testing
PS E:\STM\unit_testing>
```

- Initialize the project with npm init -y

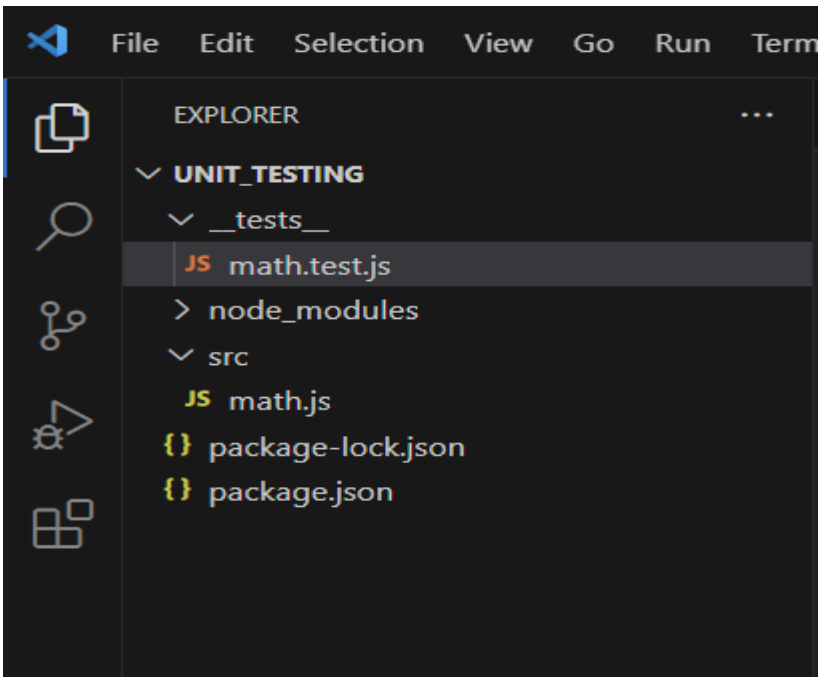
```
PS E:\STM\unit_testing> npm init -y
Wrote to E:\STM\unit_testing\package.json:

{
  "name": "unit_testing",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

- Add scripts to package.json

- Install jest with npm install --save-dev jest



- Set up structure



- Write code for math.js under src and math.test.js under __test__

package.json | math.js | math.test.js

src > math.js > ...

```
1  function add(a, b) {
2      return a + b;
3  }
4
5  function asyncMultiply(a, b) {
6      return new Promise((resolve) => setTimeout(() => resolve(a * b), 20));
7  }
8
9  module.exports = { add, asyncMultiply };
10
```

package.json | math.js | math.test.js

__tests__ > math.test.js > ...

```
1  const { add, asyncMultiply } = require('../src/math');
2
3  describe('math utilities', () => {
4    test('add returns sum', () => {
5      expect(add(2, 3)).toBe(5);
6    });
7
8    test('asyncMultiply resolves correctly', async () => {
9      await expect(asyncMultiply(3, 4)).resolves.toBe(12);
10   });
11 });
12
```

- **Run the Tests**

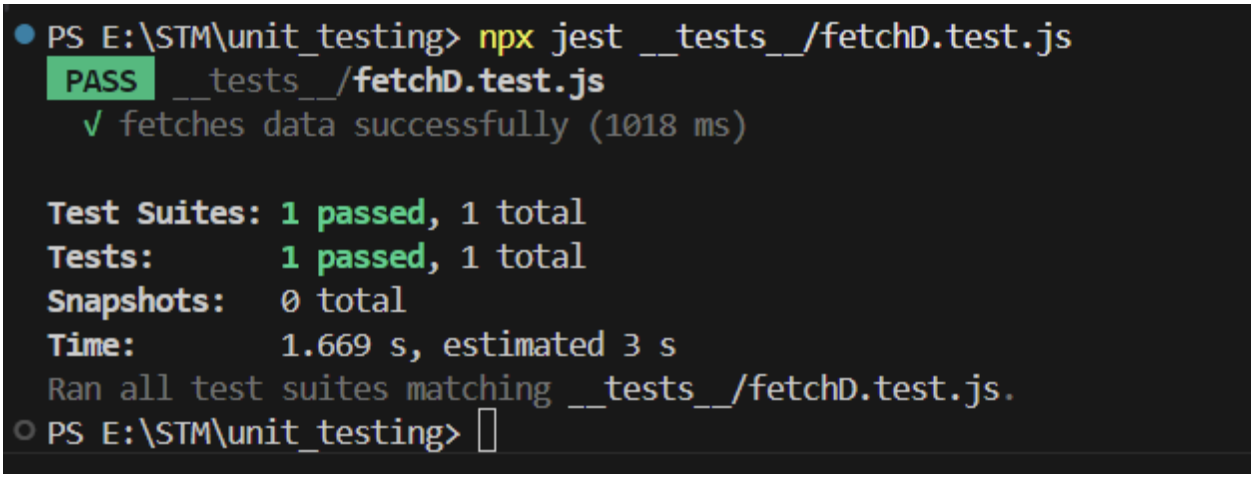
```
PS E:\STM\unit_testing> npm test

> unit_testing@1.0.0 test
> jest

 PASS  __tests__/math.test.js
  math utilities
    √ add returns sum (2 ms)
    √ asyncMultiply resolves correctly (20 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.429 s
Ran all test suites.
PS E:\STM\unit_testing>
```

- Test with some faulty code too to check jest is working perfectly. Here, in math.js return a + b is changed to a – b .

```
PS E:\STM\unit_testing> npm test

> unit_testing@1.0.0 test
> jest

FAIL __tests__/math.test.js
  math utilities
    × add returns sum (5 ms)
    √ asyncMultiply resolves correctly (35 ms)

  ● math utilities › add returns sum

    expect(received).toBe(expected) // Object.is equality

    Expected: 5
    Received: -1

      3 | describe('math utilities', () => {
      4 |   test('add returns sum', () => {
    > 5 |     expect(add(2, 3)).toBe(5);
        |                       ^
      6 |   });
      7 |
      8 |   test('asyncMultiply resolves correctly', async () => {

      at Object.toBe (__tests__/math.test.js:5:23)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
```

Testing Asynchronous Functions

Jest makes it easy to test asynchronous functions using async/await or returning promises.



- The test is run using npx jest __tests__/fetchD.test.js

Mocking in Jest

Jest provides built-in functionality to mock functions, helping to isolate tests.

E:\STM\unit_testing\package.json

```
1   // api.js
2   function API() {
3       return 'Real data from API';
4   }
5   module.exports =API;
```

__tests__ > JS api.test.js > ...

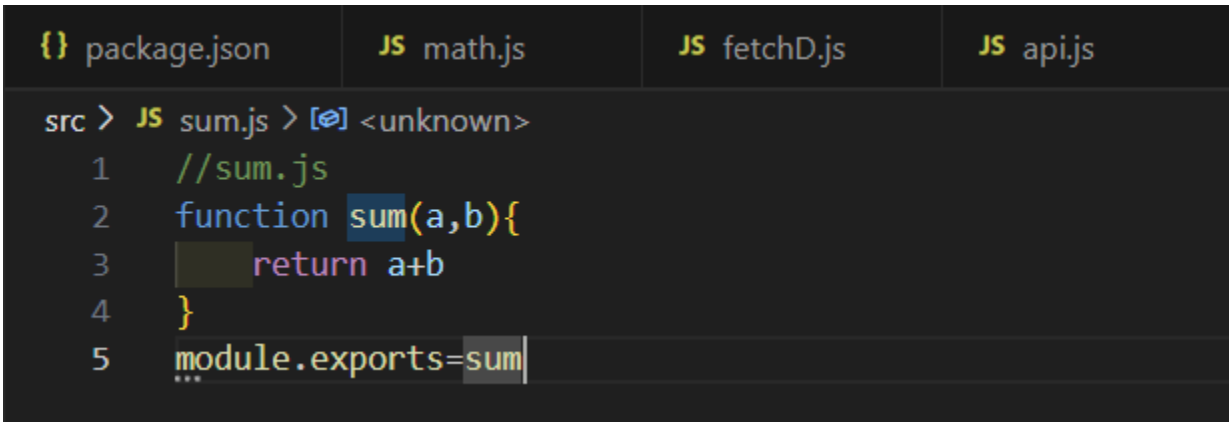
```
1   const API = require('../src/api');
2
3   jest.mock('./api');
4
5   test('mocked fetchDataFromAPI returns mocked data', () => {
6       API.mockReturnValue('Mocked data');
7       const data =API();
8       expect(data).toBe('Mocked data');
9   });
```

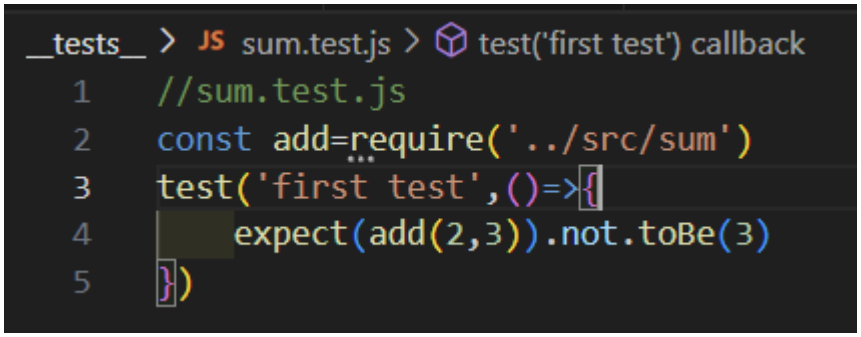
```
PS E:\STM\unit_testing> npx jest __tests__/api.test.js
 PASS  __tests__/api.test.js
   √ mocked fetchDataFromAPI returns mocked data (8 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.383 s
Ran all test suites matching __tests__/api.test.js.
PS E:\STM\unit_testing>
```

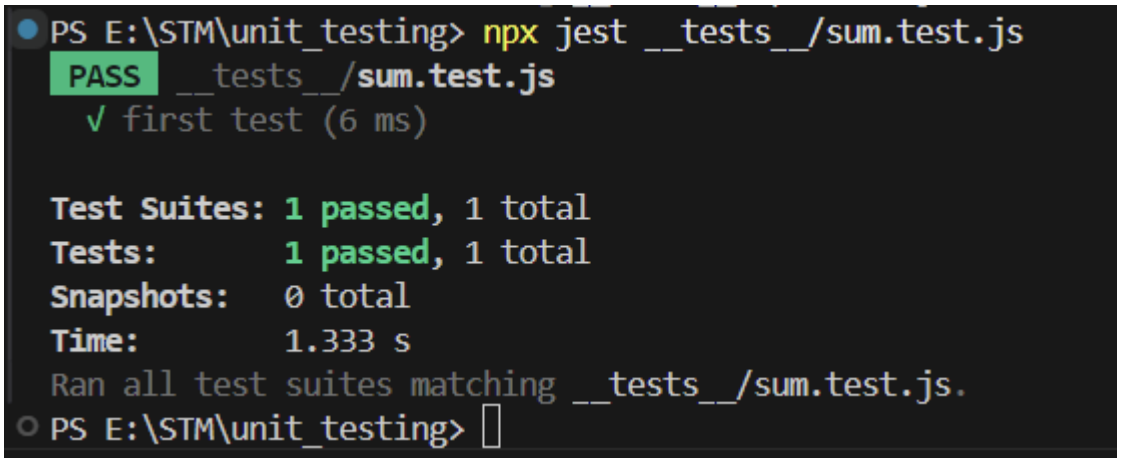
- The test is run using npx jest __tests__/api.test.js

Testing a function with parameters in jest

In this function we will test function with parameters in a dynamic way.



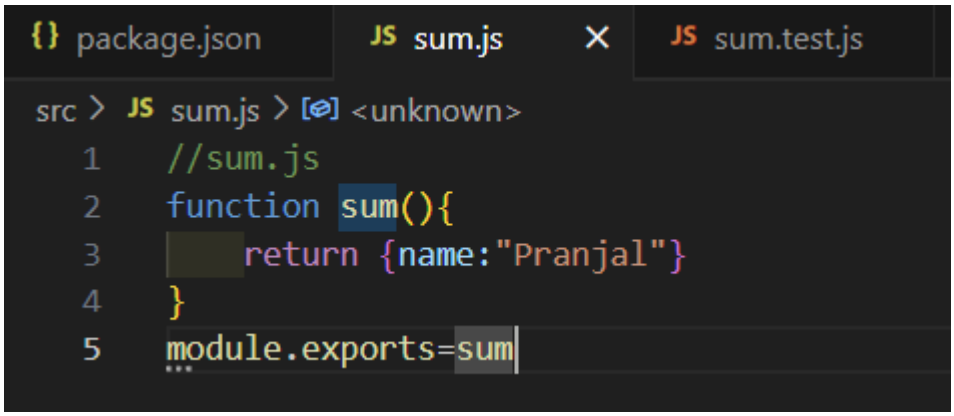


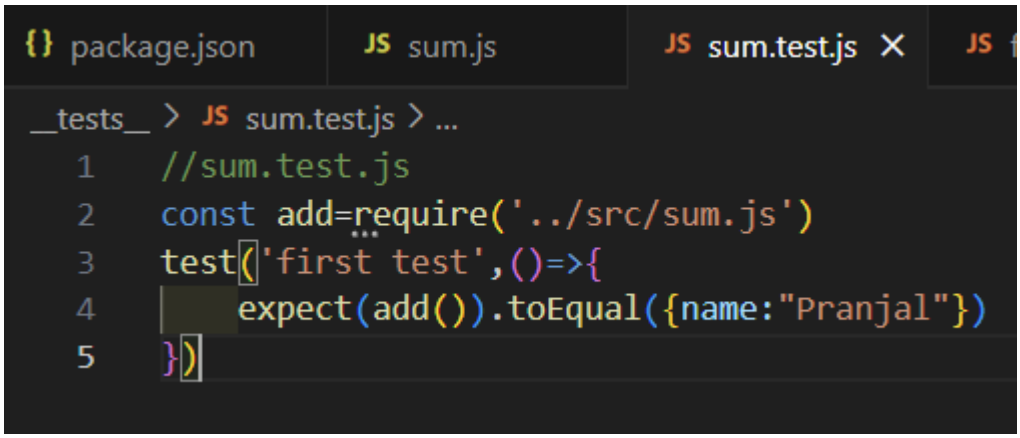


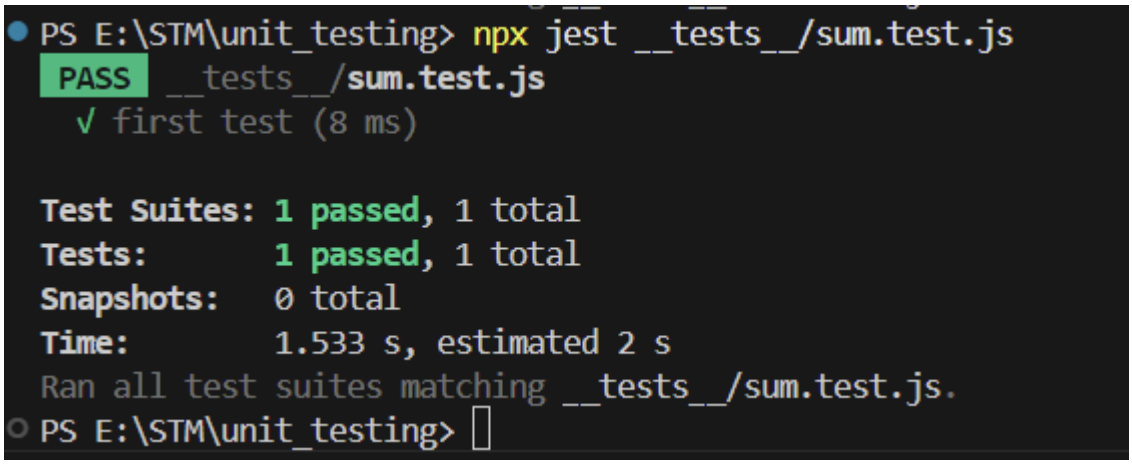
- The sum function returns the sum of its two parameters a and b.
- In sum.test.js, test() and expect() check if the result matches the expected value.
- The test is run using npx jest __tests__/sum.test.js

Testing a object in jest

Objects can also be checked with the use of toEqual() function in jest.



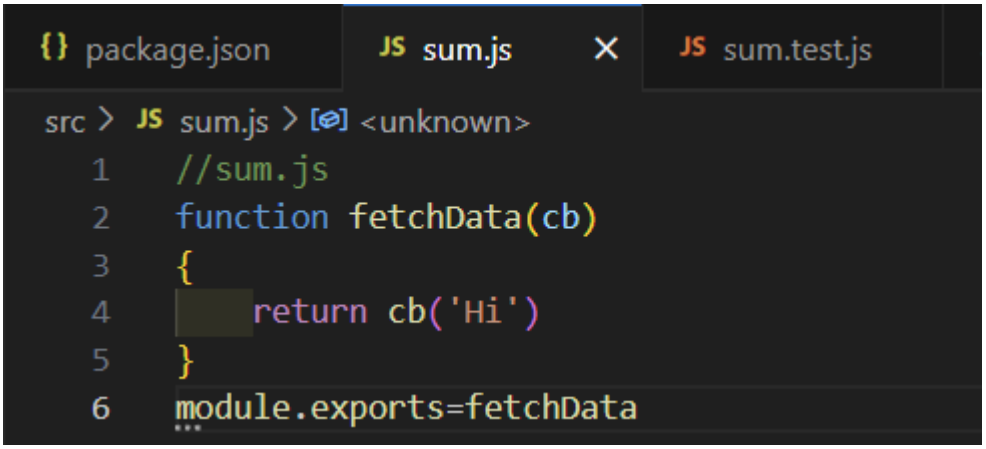


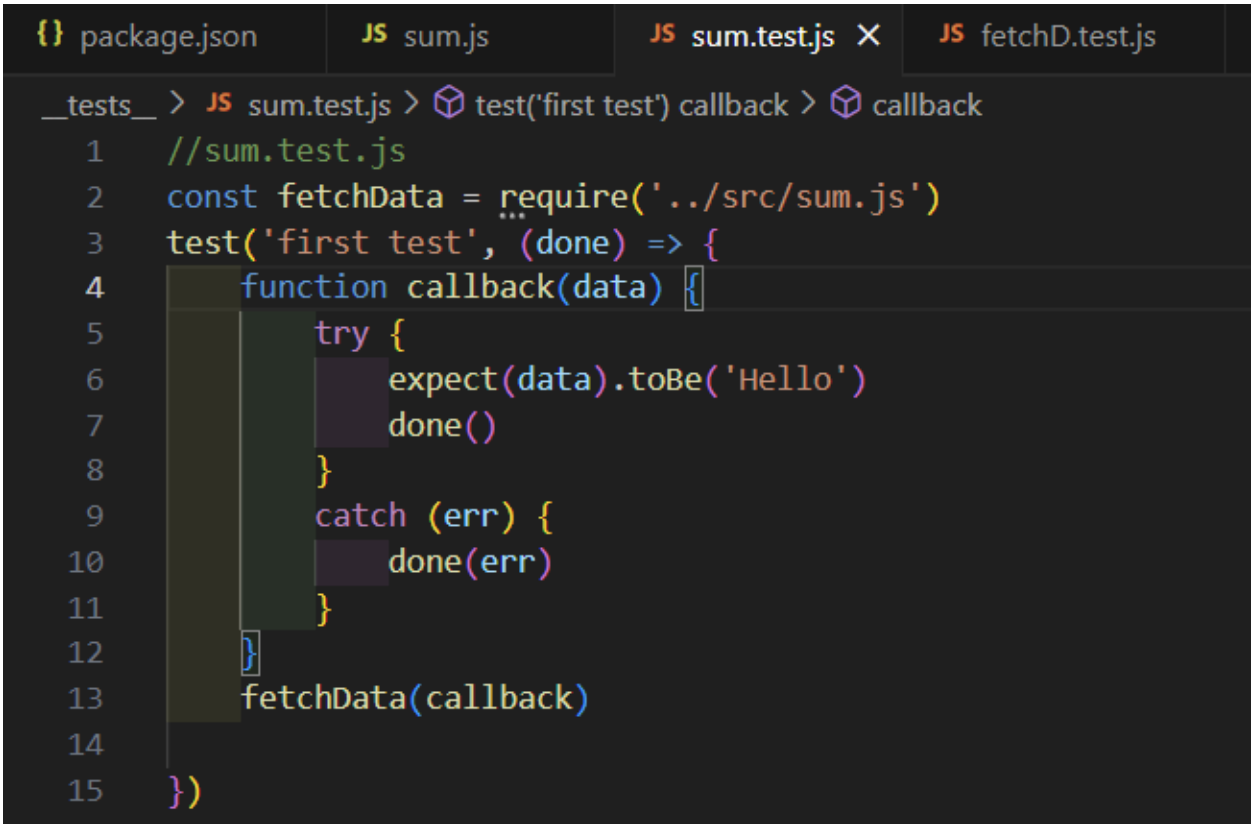


- The sum function returns an object { name: "Pranjal" }.
- In sum.test.js, test() and expect() with toEqual() check if the returned object matches the expected value.
- The test passes if the values match, otherwise it fails.

Testing callback functions

To test callback functions, use Jest's test() with done, pass the callback, and call done() after the assertion.





```
PS E:\STM\unit_testing> npx jest __tests__/sum.test.js
 PASS  __tests__/sum.test.js
  √ first test (8 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.449 s
Ran all test suites matching __tests__/sum.test.js.
PS E:\STM\unit_testing>
```

Change the data in sum.js with “Hi” and the result will show failure.

```
PS E:\STM\unit_testing> npx jest __tests__/sum.test.js
 FAIL  __tests__/sum.test.js
  × first test (20 ms)

  ● first test

    expect(received).toBe(expected) // Object.is equality

    Expected: "Hello"
    Received: "Hi"

      4 |     function callback(data) {
      5 |         try {
    > 6 |             expect(data).toBe('Hello')
        |                          ^
      7 |             done()
      8 |         }
      9 |         catch (err) {

      at toBe (__tests__/sum.test.js:6:26)
      at cb (src/sum.js:4:12)
      at Object.fetchData (__tests__/sum.test.js:13:5)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.391 s, estimated 2 s
Ran all test suites matching __tests__/sum.test.js.
```

- The fetchData function in sum.js takes a callback and passes 'Hi' as a string.
- In sum.test.js, fetchData is imported, and the callback is tested with done to check if the data matches the expected value.
- The test passes if the data matches, otherwise it fails.

Advantages of Jest:

1. Zero Configuration

- Works out of the box for most JavaScript and TypeScript projects. No need for complex setup or extra tools for basic usage.

2. Fast and Efficient

- Runs tests **in parallel** across multiple workers, improving performance.
- Supports **watch mode**, which reruns only changed tests — saves time during development.

3. Snapshot Testing

- Built-in snapshot feature is great for **UI testing** (e.g., React components).
- Helps you detect unexpected changes in output or UI structure.

4. Powerful Mocking Capabilities

- Allows **mocking of functions, modules, and timers** easily.
- Lets you isolate components and test logic independently from dependencies (e.g., APIs, databases).

5. Code Coverage Built-in

- Generates **coverage reports** (--coverage flag).
- Quickly identifies untested parts of your codebase.

6. Wide Ecosystem & Community Support

- Maintained by Facebook and the open-source community.
- Well-documented, with a large collection of tutorials and integrations (React, TypeScript, Node.js).

7. Rich Assertions and Matchers

- Provides clear and expressive assertion methods (toBe, toEqual, toContain, toThrow, etc.).
- Readable test output and colored diff comparisons.

Disadvantages of Jest:

1. Not Ideal for All Environments

- Primarily optimized for **Node.js and React**.
- May need extra configuration for **non-Node** environments (like Vue, Angular, or ESM modules).

2. Slow in Large Projects

- On very large codebases, test startup or watch mode can become slower.
- Parallel execution sometimes increases memory usage.

3. Limited Integration Testing Features

- Designed mainly for **unit and snapshot testing**.
- Not ideal for **end-to-end (E2E)** or **UI automation** — tools like Cypress or Playwright are better for that.

4. Configuration Can Get Complex

- While basic setup is easy, integrating Jest with **TypeScript, Babel, or ES modules** sometimes requires detailed configuration.
- Custom environments (e.g., browser + Node mix) may need additional plugins.

5. Snapshots Can Be Misused

- Large or frequent snapshot updates can hide unintended changes.
- Developers might accept snapshot updates without reviewing differences carefully.

6. Learning Curve for Beginners

- Features like mocks, spies, and fake timers can be confusing for new developers.
- May require time to understand advanced mocking behavior and asynchronous testing patterns.