Angular Testing with Jest

Khalil Themri

What is Unit Testing?

Unit testing is a software testing technique where individual components or functions of a software application are tested in isolation. The primary goal is to ensure that each unit of the software code performs as expected.

A "unit" refers to the smallest testable part of an application, such as a function, method, or class. By testing these units independently, developers can identify and fix bugs early in the development process, leading to more robust and maintainable code.

Why Unit Testing is Important

- **Early Detection of Bugs:** Unit tests help identify issues at an early stage, before they become harder and more expensive to fix.
- **Facilitates Refactoring:** With a comprehensive suite of unit tests, developers can refactor code confidently, knowing that existing functionality will be preserved.
- **Improves Code Quality:** Writing unit tests encourages developers to write modular, clean, and well-documented code.
- **Simplifies Integration:** By ensuring each unit works correctly, integrating these units into larger systems becomes more manageable.

Overview of Unit Testing Tools

Running Jest Files in Visual Studio Code

Before diving into the examples, it's important to know how to run Jest tests in your development environment. If you're using Visual Studio Code (VSCode), there are a couple of plugins that can enhance your experience:

1. **Jest Plugin by Orta:**
    - Install the *Jest* plugin by orta from the VSCode extensions marketplace.
    - This plugin automatically detects Jest tests in your project and runs them in the background. You can see the test results directly in the editor.
    - It provides helpful features like inline test results and coverage visualization.
2. **Jest Runner:**
    - Install the *Jest Runner* plugin from the VSCode extensions marketplace.

- Jest Runner allows you to run or debug a single test or a whole test file with just a click. This is particularly useful when you're working on a specific part of your test suite.
- To use it, right-click on a test file or test function and select "Run Jest" or "Debug Jest".

With these tools, running and debugging your Jest tests becomes much more efficient, allowing you to focus on writing and improving your tests.

Basic Jest Examples

Let's start with some simple Jest examples to understand the fundamentals of testing.

Listing 1: A super basic test example.

```
describe('super basic test', () => {
  it('true is true', () => {
    expect(true).toEqual(true);
  });
});
```

The above example is a basic test that simply checks if 'true' is equal to 'true'. This is useful for ensuring that your testing environment is set up correctly.

Listing 2: Testing the Cat class getters and setters.

```
import { Cat } from './cat';

describe('Test Cat getters and setters.', () => {
  it('The cat name should be Gracie', () => {
    const cat = new Cat();
    cat.name = 'Gracie';
    expect(cat.name).toEqual('Gracie');
  });
});
```

In this example, we test a simple getter and setter in a 'Cat' class to verify that the 'name' property is set and retrieved correctly.

In this example, tests are grouped by chapters, which allows for a clear organization of tests as you progress through different sections or chapters of your project or tutorial.

Understanding Jest Test Structure

In this section, we'll explore a common structure for testing an Angular component using Jest. The code is organized into blocks with 'describe', 'beforeEach', 'afterEach', and 'it' functions, each serving a distinct purpose in testing.

Example Jest Test Suite

Listing 3: Jest Test Suite for an Angular Component

```
describe('ComponentName', () => {
  // Declarations and variable initializations
  // e.g., let service, component;

  beforeEach(() => {
    // Optional: Preparing the test environment
    // e.g., initTestEnvironment();
  });

  beforeEach(() => {
    // Initialization logic specific to each test
    // e.g., component = new ComponentName();
  });

  afterEach(() => {
    // Cleanup logic after each test execution
    // e.g., service.reset();
  });

  describe('firstFunction', () => {
    // Tests specific to the firstFunction
    it('should work', () => {
      // Test logic for a successful case
    });

    it('should fail', () => {
      // Test logic for a failing case
    });
  });

  describe('SecondFunction', () => {
    // Tests specific to the SecondFunction
    it('should work', () => {
      // Test logic for a successful case
    });

    it('should fail', () => {
      // Test logic for a failing case
    });
  });
});
```

Explanation of the Test Structure

- **describe:** This function is used to group related tests. Here, ‘ComponentName‘ is the main group, and ‘firstFunction‘ and ‘SecondFunction‘ are sub-groups for specific functions.
- **beforeEach:** This function runs before each ‘it‘ block within its ‘describe‘ block. It’s useful for setting up any test-specific initialization.
- **afterEach:** This function runs after each ‘it‘ block within its ‘describe‘ block. It’s typically used to clean up or reset states that were modified during the test. Even though we have ‘beforeEach‘, ‘afterEach‘ ensures that any changes made during a test don’t affect other tests.
- **it:** This function represents an individual test case. It’s where you write assertions to check if the code behaves as expected.