Jasmine

Npm install jasmin-node is an older way. New way is

npm install jasmine --save-dev

**2. Initialize Jasmine:**

After installing **jasmine**, you can initialize it in your project by running the following command:

npx jasmine init

**3. Write Your Jasmine Tests:**

Create your test specs inside the **spec** directory. Each test file should have the **.spec.js** extension. Write your test suites and specs in these files, following the Jasmine testing syntax.

**4. Run Jasmine Tests:**

You can run your Jasmine tests using the following command:

Npx jasmine

Jasmine will discover and run the test specs located in the **spec** directory and report the test results.

Initialize the test environment

The **jasmine init** command is used to initialize a Jasmine project. When you run **jasmine init**, it sets up the basic configuration and directory structure for your Jasmine testing environment. This command is part of the Jasmine testing framework, which is commonly used for writing and running tests in JavaScript applications.

Here's what **jasmine init** typically does:

1. **Directory Structure**: It creates a directory structure for your Jasmine tests, with a default **spec** directory where your test spec files should be placed. The **jasmine.json** configuration file is also created in the root of your project.
2. **jasmine.json Configuration**: The **jasmine.json** file is a configuration file that allows you to specify options and settings for your Jasmine tests. It includes configurations such as the location of your test files, the reporter used for displaying test results, and more.
3. **Spec Helper File**: It creates a **spec/support** directory and a **jasmine.json** file where you can include any helper code or setup that should be available to all your test specs.
4. **Example Specs**: It typically includes some example test specs to help you get started with writing your own tests.

In Jasmine, the **describe** function is a fundamental building block used to define a test suite. A test suite is a collection of related test specs (test cases) that are organized together. The **describe** function provides a structured way to group related test specs and set up a common context for the tests.

Here's how the **describe** function works and how it is used:

**Syntax:**

describe(description, functionDefinition);

, functionDefinition);

* **description**: A string that describes the test suite. This string is typically a human-readable description of what the suite is testing. It should be clear and concise.
* **functionDefinition**: A function that contains the test specs (test cases) for the suite. This function defines the scope of the suite and typically contains one or more **it** blocks.

**Example:**

describe('Math functions', function () {

it('should add two numbers correctly', function () {

// Your test spec code here

});

it('should subtract two numbers correctly', function () {

// Your test spec code here

});

});above, **describe** is used to create a test suite with the description "Math functions." Inside the suite, there are two test specs defined using the **it** function, each describing a specific behavior of math functions. The **describe** block provides a common context for the test specs within it.

The **describe** function is essential for organizing and structuring your tests, especially when you have multiple test specs that relate to the same functionality or component. It helps make your test suite more readable and maintainable. By nesting **describe** blocks, you can create a hierarchy of test suites, which is particularly useful for complex projects with numerous tests.

[Built-In Matchers](https://codecraft.tv/courses/angular/unit-testing/jasmine-and-karma/#_built_in_matchers)

Jasmine comes with a few pre-built matchers like so:

TypeScript

Copyexpect(array).toContain(member);

expect(fn).toThrow(string);

expect(fn).toThrowError(string);

expect(instance).toBe(instance);

expect(mixed).toBeDefined();

expect(mixed).toBeFalsy();

expect(mixed).toBeNull();

expect(mixed).toBeTruthy();

expect(mixed).toBeUndefined();

expect(mixed).toEqual(mixed);

expect(mixed).toMatch(pattern);

expect(number).toBeCloseTo(number, decimalPlaces);

expect(number).toBeGreaterThan(number);

expect(number).toBeLessThan(number);

expect(number).toBeNaN();

expect(spy).toHaveBeenCalled();

expect(spy).toHaveBeenCalledTimes(number);

expect(spy).toHaveBeenCalledWith(...arguments);

Top of Form

Karma

ng g config karma

angular testing package includes two utilities: testbed and async

In the context of testing in Angular, the **async** utility is often used in conjunction with the **TestBed** utility. While **TestBed** is responsible for configuring and creating instances of components and services for unit testing, the **async** utility helps manage asynchronous operations within your test cases. Let's explore the roles of both utilities:

1. **TestBed:**
   * The **TestBed** utility is part of the **@angular/core/testing** package and is used for configuring and creating a testing module.
   * It allows you to set up a testing environment that mimics an Angular module, where you can define the components, services, providers, and dependencies needed for your tests.
   * **TestBed.createComponent** is commonly used to create instances of components for testing.
   * You can configure and override dependencies, services, and providers within the testing module using **TestBed**.
   * **TestBed** is essential for setting up a controlled environment for unit testing Angular components and services.
2. **async:**
   * The **async** utility is used to manage asynchronous operations within your test cases.
   * It's also part of the **@angular/core/testing** package.
   * Angular often involves asynchronous operations, such as HTTP requests and timeouts. The **async** utility helps you handle these scenarios in a test.
   * When you use **async** in your test, it allows you to work with asynchronous code while ensuring that the test runner waits for the asynchronous operations to complete before evaluating your test assertions.
   * The **async** utility is particularly useful when testing components that involve asynchronous tasks like observables and promises.

Here's an example of how **TestBed** and **async** are commonly used together in an Angular test:

import { TestBed, ComponentFixture, async } from '@angular/core/testing';

import { MyComponent } from './my.component';

import { MyService } from './my.service';

describe('MyComponent', () => {

let fixture: ComponentFixture<MyComponent>;

let component: MyComponent;

beforeEach(async(() => {

TestBed.configureTestingModule({

declarations: [MyComponent],

providers: [MyService],

}).compileComponents(); // Compile components asynchronously

fixture = TestBed.createComponent(MyComponent);

component = fixture.componentInstance;

}));

it('should create the component', () => {

expect(component).toBeTruthy();

});

it('should do something asynchronously', async(() => {

// Perform asynchronous tasks and make assertions here

fixture.detectChanges(); // Trigger change detection

fixture.whenStable().then(() => {

// Assertions that rely on asynchronous behavior should go here

});

});

});

it('should make a POST request', waitForAsync(inject([HttpTestingController, DataService],

  (httpClient: HttpTestingController, service: DataService) => {

    const newProduct = { id: 18, name: 'Product 18', description: 'Description 18',category: 'laptop' };

    // Use the service to make the POST request

  service.addProduct(newProduct).subscribe((response) => {

      expect(response).toEqual(newProduct); // Check if the response matches the expected new product

  });

  // Set up an expectation for the POST request

  const req = httpMock.expectOne('http://localhost:3000/products'); // Replace with your API endpoint

  expect(req.request.method).toEqual('POST');

  expect(req.request.body).toEqual(newProduct); // Verify the request data

      // Respond to the request with a mock response

 // Respond to the request with the new product

  req.flush(newProduct, { status: 201, statusText: 'Created'});

Fixture

In Angular testing, a fixture is **a wrapper for a component and its template**. It provides a convenient interface to both the component instance and the rendered DOM. The fixture is created by the TestBed and it can be used to interact with the component and its template in various ways.

In Jasmine, a "spy" is a feature used for mocking or stubbing functions in order to track their calls and behavior. Spies allow you to:

1. **Track Function Calls:** You can create a spy on a function or method and then verify if and how many times it was called.
2. **Control Behavior:** Spies can be configured to return specific values, throw errors, or call through to the original function.

SpyOn() is a function that takes two arguments: the first argument is the object or class that you want to spy on, and the second argument is the function or property that you want to spy on.