**Chapter 13: Testing & Debugging**

**1.1 Why Testing Matters in Angular Applications**

Modern web applications are complex, with many components, services, and business logic working together. Without testing, it's hard to ensure your code behaves correctly over time.

**✳Key Reasons to Test Your Angular Code**

* **Detect bugs early** before they reach production
* **Ensure code quality** with confidence during refactoring
* **Enable team collaboration** and shared ownership of features
* **Support CI/CD** pipelines with automated test runs
* **Ensure maintainability** for large projects over time

**1.2 Types of Testing in Angular**

Angular supports several layers of testing. Each serves a different purpose.

| **Type** | **Purpose** | **Tools** |
| --- | --- | --- |
| **Unit Testing** | Tests individual functions, classes, or components | Jasmine, Jest |
| **Integration Testing** | Tests interaction between components/services | Angular TestBed |
| **E2E Testing** | Tests app behavior from the user’s perspective | Playwright, Cypress |

**1.3 Angular Testing Ecosystem Overview**

When you create an Angular project using the CLI, it sets up a robust testing environment for you.

**Tools Included by Default**

| **Tool** | **Role** |
| --- | --- |
| **Jasmine** | Testing framework (describe, it, expect) |
| **Karma** | Test runner (launches browser, runs tests) |
| **TestBed** | Angular's test harness for creating/testing components |
| **HttpClientTestingModule** | Helps test HTTP requests |
| **Jasmine Spy Functions** | Used to mock and observe function calls |

**1.4 What Are Spec Files?**

In Angular, test files are placed next to the source files, ending with .spec.ts.

Example:

login.component.ts // Main component

login.component.spec.ts // Test for the component

**Anatomy of a Spec File­­**

Here’s what a typical test file contains:

* **TestBed configuration** (creates the test module)
* **Fixture and component instance**
* **beforeEach()** to set up test logic
* **Test cases using it()**
* **Assertions using expect()**

**1.5 Jasmine Basics (Core Testing Syntax)**

Jasmine is the default testing framework Angular uses. Here are the core functions you’ll use:

**describe() – Group Tests**

describe('MyComponent', () => {

// test cases go here

});

**it() – A Single Test Case**

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

expect(true).toBe(true);

});

**Example:**

describe('Math', () => {

it('should add numbers correctly', () => {

expect(2 + 2).toBe(4);

});

});

**expect() – Assertions**

expect(value).toBe(expectedValue);

**Spies – Monitor or Mock Functions**

spyOn(service, 'methodName').and.returnValue(mockValue);

**1.6 Running Tests with Angular CLI**

You can run all your tests using:

ng test

If we are working on a workspace with multiple projects

ng test –project <projectname>

This will:

* Compile your test files
* Launch a browser via Karma
* Run all \*.spec.ts files
* Show results live

**1.7 Code Coverage**

Angular makes it easy to track how much of your code is tested:

ng test --code-coverage

This creates a coverage/ folder with an HTML report showing:

* Covered lines
* Missed lines
* Branch and function coverage

**1.8 Testing in Angular 19+ (Standalone Components)**

With Angular 15+, you can use **standalone components** (no NgModules). In Angular 19, this is the default.

**Testing Standalone Components**

Instead of using declarations in TestBed, use imports:

TestBed.configureTestingModule({

imports: [MyStandaloneComponent] // ✅ not declarations

});

This is cleaner and more modular than before.

**Chapter 2: Writing Your First Angular Test (With Standalone Component)**

**2.1 Introduction**

Now that you understand the **fundamentals of testing in Angular**, it’s time to write your **first real unit test**. In this chapter, you’ll build a **simple standalone Angular component**, and write test cases to verify its behavior.

We’ll go step-by-step to cover:

* Creating a component using Angular CLI
* Understanding how TestBed works
* Writing unit tests for user interaction
* Verifying DOM updates
* Spying on @Output() events

**2.2 Scenario: ReactiveForm Component**

Angular CLI generates a .spec.ts file when you create a component:

ng generate component user-form --standalone

Creates:

user-form.component.ts

user-form.component.spec.ts ✅ (test file)

**1. Component: UserFormComponent**

import { Component } from '@angular/core';

import { ReactiveFormsModule, FormGroup, FormBuilder, Validators } from '@angular/forms';

@Component({

standalone: true,

selector: 'app-user-form',

template: `

<form [formGroup]="form" (ngSubmit)="submit()">

<label>

Name: <input formControlName="name" />

</label>

<label>

Email: <input formControlName="email" />

</label>

<button type="submit">Submit</button>

</form>

`,

imports: [ReactiveFormsModule]

})

export class UserFormComponent {

form: FormGroup;

constructor(private fb: FormBuilder) {

this.form = this.fb.group({

name: ['', Validators.required],

email: ['', [Validators.required, Validators.email]]

});

}

submit() {

if (this.form.valid) {

console.log('Form Submitted', this.form.value);

}

}

}

**2. Test File: user-form.component.spec.ts**

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

import { ReactiveFormsModule } from '@angular/forms';

import { UserFormComponent } from './user-form.component';

describe('UserFormComponent', () => {

let component: UserFormComponent;

let fixture: ComponentFixture<UserFormComponent>;

beforeEach(async () => {

await TestBed.configureTestingModule({

imports: [UserFormComponent] // because it's standalone

}).compileComponents();

fixture = TestBed.createComponent(UserFormComponent);

component = fixture.componentInstance;

fixture.detectChanges();

});

it('should create the form with 2 controls', () => {

expect(component.form.contains('name')).toBeTrue();

expect(component.form.contains('email')).toBeTrue();

});

it('should mark form as invalid when fields are empty', () => {

component.form.setValue({ name: '', email: '' });

expect(component.form.valid).toBeFalse();

});

it('should mark form as valid with proper values', () => {

component.form.setValue({ name: 'Alice', email: 'alice@example.com' });

expect(component.form.valid).toBeTrue();

});

it('should invalidate email field if not a proper email', () => {

component.form.controls['email'].setValue('invalid-email');

expect(component.form.controls['email'].valid).toBeFalse();

});

it('should log form value on submit if valid', () => {

spyOn(console, 'log');

component.form.setValue({ name: 'Bob', email: 'bob@example.com' });

component.submit();

expect(console.log).toHaveBeenCalledWith('Form Submitted', { name: 'Bob', email: 'bob@example.com' });

});

});

**How to Run the Tests**

* Project in workspace.

ng test –project p10-testing

* Not in Workspace

ng test

* This opens a browser and runs all test cases.
* Results are shown in both the terminal and the browser.

✅ Green = passed, ❌ Red = failed.

**✅ Test Tips for Beginners**

* Use TestBed.configureTestingModule() to configure your test setup.
* Use fixture.detectChanges() to trigger Angular lifecycle hooks.
* Use spyOn() to mock and verify functions like console.log, services, etc.
* Always test:
  + **Structure** (component exists, form controls)
  + **Behavior** (button clicks, validation)
  + **Edge cases** (invalid inputs, empty states)

**Practice Exercises**

1. Create a LoginFormComponent with email + password and write tests for:
   * Valid form submission
   * Invalid email
   * Missing password
2. Test a service that fetches users from an API (mock HttpClient)
3. Write unit tests for a custom pipe (e.g., truncate)