Angular Unit Testing with Jasmine - Learning Plan

A comprehensive, step-by-step learning roadmap for mastering unit testing in Angular applications using Jasmine.

Table of Contents

Learning Path Overview

```
Week 1-2:   Foundations → Testing concepts + Jasmine
syntax
Week 3-4:   Angular Tools → TestBed, fixtures,
debugging
Week 5-6:   Components → Isolated & integrated
component tests
Week 7:     Services → HTTP, state management,
dependencies
Week 8-9:   Advanced → Pipes, directives, signals,
routing
Week 10-12: Real-World → Complete application testing
```

Time Commitment: 10-15 hours per week

Total Duration: 12 weeks

Final Goal: Confidently write comprehensive unit tests for any Angular application

---

Prerequisites

Required Knowledge

- ✅ TypeScript fundamentals (types, interfaces, classes)
- ✅ Angular basics (components, services, dependency injection)
- ✅ RxJS fundamentals (observables, operators)
- ✅ Command line basics

Setup Requirements

```
# Angular CLI
npm install -g @angular/cli

# Create test project
ng new angular-testing-practice --routing=false --
style=scss

# Verify Jasmine is installed (comes with Angular by
default)
npm list jasmine-core karma karma-jasmine
```

Phase 1: Testing Fundamentals (Week 1)

Learning Objectives

- Understand why testing matters
- Learn the testing pyramid concept
- Grasp unit vs integration vs E2E testing
- Understand test-driven development (TDD) basics

Topics to Study

1.1 Why Test? (Day 1)

Concept: Understanding the value of automated testing

The Problem Without Testing:
Imagine you're building a banking application. You implement a transfer function that works perfectly during manual testing. Two months later, a colleague modifies the currency conversion logic. They test their changes, but unknowingly break the transfer function. This bug goes unnoticed until a customer loses money in production.

Automated tests prevent this scenario.

Detailed Benefits:

1. **Catch Bugs Early in Development**
   - Tests act as a safety net, catching issues before code review
   - Finding a bug in development costs ~$100, in production it can cost $10,000+
   - Tests run in seconds, giving instant feedback on code changes
2. **Enable Confident Refactoring**
   - Without tests: "I want to refactor this, but what if I break something?"
   - With tests: "I'll refactor, and tests will tell me immediately if anything breaks"
   - This leads to cleaner, more maintainable code over time
3. **Document Code Behavior**
   - Tests serve as executable documentation
   - `it('should reject invalid email formats')` is clearer than a comment

- New team members can read tests to understand how code should work

4. **Reduce Manual Testing Time**
   - Manual testing the same features repeatedly wastes developer time
   - Automated tests run thousands of scenarios in minutes
   - Frees you to focus on building features instead of clicking buttons
5. **Improve Code Quality and Design**
   - Writing testable code naturally leads to better architecture
   - Forces you to think about separation of concerns
   - Hard-to-test code is often poorly designed code

Real-World Example:

```
// Read and analyze this buggy code
export class Calculator {
  add(a: number, b: number): number {
    return a - b; // Bug: should be addition (using subtraction!)
  }
}

// Without tests: Bug ships to production
// Developer manually tests: calculator.add(5, 3) = 2
// Developer thinks: "That looks right... wait, that's wrong!"
// But they might not test all combinations
// With tests: Bug caught immediately
describe('Calculator', () => {
  it('should add two positive numbers', () => {
    const calc = new Calculator();
    expect(calc.add(2, 2)).toBe(4); // FAILS! Returns 0
    // Test fails immediately, bug never reaches
production
  });

  it('should handle negative numbers', () => {
    expect(calc.add(-5, 3)).toBe(-2); // FAILS! Returns
-8
  });
});
```

The ROI (Return on Investment):

- Time to write test: 2 minutes
- Time saved finding bug later: 30+ minutes
- Customer trust preserved: Priceless

Mindset Shift: Don't think "I don't have time to write tests." Think "I don't have time to NOT write tests."

Exercise:
Take 10 minutes to document a real bug you've encountered. How would a test have caught it? Write that test.

1.2 Testing Pyramid (Day 2)

Concept: Balance different types of tests

```
        /\
       /E2E\       10% - End-to-End (slow, brittle)
      /------\
     /Integr.\    20% - Integration (moderate)
    /----------\
   /   Unit     \ 70% - Unit Tests (fast, reliable)
  /--------------\
```

Key Principle: Write more unit tests, fewer E2E tests

1.3 Arrange-Act-Assert Pattern (Day 3-4)

Concept: Structure every test consistently

Why AAA Matters:
The AAA pattern is like a recipe - it provides consistent structure that makes tests easy to read and maintain. Any developer can look at an AAA test and immediately understand what's being tested, even without prior context.

The Three Phases Explained:

1. ARRANGE (Setup)

- Create the test data and objects you need
- Think: "What ingredients do I need?"
- Set up the initial state of the system
- This is usually the longest section

2. ACT (Execute)

- Call the method/function you're testing

- Think: "What am I actually testing?"
- This should usually be ONE line (testing one thing)
- The actual behavior under test

3. ASSERT (Verify)

- Check that the result matches expectations
- Think: "Did it do what I expected?"
- Can have multiple assertions, but they should all relate to the same act

Basic Example with Detailed Explanation:

```
it('should calculate total with tax', () => {
  // ARRANGE - Set up test data
  // We need: a price, a tax rate, and a calculator
  const price = 100;        // Base price
  const taxRate = 0.1;      // 10% tax
  const calculator = new TaxCalculator();  // Our
system under test

  // ACT - Execute the function being tested
  // This is the ONE thing we're testing
  const result = calculator.calculateTotal(price,
taxRate);

  // ASSERT - Verify the outcome
  // 100 + (100 * 0.1) = 110
  expect(result).toBe(110);
});
```

Complex Example - Why AAA Shines:

```
it('should apply discount before tax for premium
members', () => {
  // ARRANGE
  const cart = new ShoppingCart();
  const premiumUser = {
    id: 1,
    membershipLevel: 'premium',
    discountRate: 0.15  // 15% discount
  };
  const items = [
    { name: 'Item 1', price: 100 },
    { name: 'Item 2', price: 50 }
  ];
  const taxRate = 0.1;  // 10% tax

  // Set up the cart
  items.forEach(item => cart.addItem(item));
  cart.setUser(premiumUser);

  // ACT
  const total = cart.calculateTotal(taxRate);

  // ASSERT
  // Calculation: (100 + 50) = 150
  // With discount: 150 - (150 * 0.15) = 127.50
  // With tax: 127.50 + (127.50 * 0.1) = 140.25
  expect(total).toBe(140.25);
  expect(cart.discountApplied()).toBe(22.50);
  expect(cart.taxApplied()).toBe(12.75);
});
```

Common AAA Mistakes:

❌ Bad - Mixed Phases:

```
it('confusing test', () => {
  const calc = new Calculator();   // Arrange
  const result1 = calc.add(2, 2); // Act
  expect(result1).toBe(4);         // Assert
  const result2 = calc.add(5, 5); // Act again?
  expect(result2).toBe(10);        // Assert again?
});
```

✅ Good - Clear Separation:

```
it('should add two numbers', () => {
  // Arrange
  const calc = new Calculator();

  // Act
  const result = calc.add(2, 2);

  // Assert
  expect(result).toBe(4);
});

it('should add different numbers', () => {
  // Arrange
  const calc = new Calculator();

  // Act
  const result = calc.add(5, 5);
  // Assert
  expect(result).toBe(10);
});
```

Pro Tips:

1. **Add comments** labeling each section when learning
2. **Keep Act minimal** - usually one line
3. **If Arrange is too long**, extract a helper function
4. **Multiple asserts are OK** if they verify the same action
5. **Each test should test ONE behavior**

Visual Mental Model:

```
ARRANGE → ACT → ASSERT
Set stage → Perform action → Check result
Prepare → Execute → Verify
Given → When → Then (BDD style)
```

Exercise:

Write 5 tests using AAA pattern for these functions. Label each section clearly:

```
function getDiscountPrice(price: number,
discountPercent: number): number
function isValidEmail(email: string): boolean
function mergArrays<T>(arr1: T[], arr2: T[]): T[]
function calculateAge(birthDate: Date): number
function formatCurrency(amount: number, currency:
string): string
```

1.4 Test Coverage (Day 5)

Concept: Measure how much code is tested

Coverage Metrics:

- **Statements**: % of code lines executed
- **Branches**: % of if/else paths tested
- **Functions**: % of functions called
- **Lines**: % of executable lines run

Target Goals:

- Critical code: 90-100%
- Business logic: 80-90%
- Overall project: 70-80%

Exercise: Run coverage on a sample project

```
ng test --code-coverage
```

Week 1 Checkpoint

- [ ] Can explain why testing is important
- [ ] Understand the testing pyramid
- [ ] Can write tests using AAA pattern
- [ ] Knows how to generate coverage reports

Phase 2: Jasmine Basics (Week 2)

Learning Objectives

- Master Jasmine syntax and structure
- Write effective test suites
- Use matchers correctly
- Handle setup and teardown

Topics to Study

2.1 Jasmine Core Concepts (Day 1-2)

Test Suite Structure:

```
describe('Calculator', () => {
  // Test suite - groups related tests

  it('should add two numbers', () => {
    // Test spec - individual test case
    expect(2 + 2).toBe(4);
  });

  it('should subtract two numbers', () => {
    expect(5 - 3).toBe(2);
  });
});
```

Key Functions:

- `describe()` - Test suite (group of tests)
- `it()` - Individual test spec
- `expect()` - Assertion
- `beforeEach()` - Setup before each test
- `afterEach()` - Cleanup after each test
- `beforeAll()` - One-time setup
- `afterAll()` - One-time cleanup

Exercise: Create a test suite with multiple specs

2.2 Jasmine Matchers (Day 3)

Built-in Matchers:

```
describe('Matchers', () => {
  it('equality matchers', () => {
    expect(value).toBe(4);              // === comparison
    expect(value).toEqual({a: 1});      // Deep equality
    expect(value).not.toBe(null);       // Negation
  });

  it('truthiness matchers', () => {
    expect(value).toBeTruthy();
    expect(value).toBeFalsy();
    expect(value).toBeNull();
    expect(value).toBeUndefined();
    expect(value).toBeDefined();
  });

  it('numeric matchers', () => {
    expect(value).toBeGreaterThan(3);
    expect(value).toBeLessThan(10);
    expect(value).toBeCloseTo(4.2, 1); // Precision
  });

  it('string matchers', () => {
    expect(message).toContain('hello');
    expect(message).toMatch(/pattern/);
  });

  it('array matchers', () => {
    expect(array).toContain(item);
    expect(array).toHaveSize(3);
  });

  it('type matchers', () => {
    expect(value).toBeInstanceOf(Class);
  });

  it('exception matchers', () => {
    expect(() => fn()).toThrow();
    expect(() => fn()).toThrowError('error message');
  });
});
```

Exercise: Write tests using 10+ different matchers

2.3 Setup and Teardown (Day 4)

```
describe('UserService', () => {
  let service: UserService;
  let database: Database;

  // Runs once before all tests
  beforeAll(() => {
    database = new Database();
    database.connect();
  });

  // Runs before each test
  beforeEach(() => {
    service = new UserService(database);
    database.clear(); // Clean state for each test
  });

  // Runs after each test
  afterEach(() => {
    service.cleanup();
  });

  // Runs once after all tests
  afterAll(() => {
    database.disconnect();
  });

  it('should fetch user', () => {
    const user = service.getUser(1);
    expect(user).toBeDefined();
  });
});
```

Best Practices:

- Use `beforeEach()` for test isolation
- Keep tests independent (no shared state)
- Clean up resources in `afterEach()`

Exercise: Refactor tests to use proper setup/teardown

2.4 Spies and Mocks (Day 5)

Understanding Spies: The Core Concept

Imagine you're testing a `PaymentProcessor` that calls a `BankAPI.charge()` method. You don't want your tests to actually charge real credit cards! This is where spies come in.

A spy is a test double that:

1. Replaces a real function
2. Tracks how it was called (arguments, call count)
3. Lets you control what it returns
4. Prevents real implementation from running (unless you want it to)

Why Use Spies?

- **Isolation**: Test your code without its dependencies
- **Speed**: Don't make real API calls or database queries
- **Control**: Simulate different scenarios (success, failure, edge cases)
- **Verification**: Ensure functions are called correctly

Spy Basics - Deep Dive:

1. Tracking Calls (The Detective)

```
describe('Spies - Call Tracking', () => {
  it('should track function calls', () => {
    const calculator = {
      add: (a: number, b: number) => a + b
    };

    // Install a spy on the 'add' method
    // This replaces the real function with a spy
    spyOn(calculator, 'add');

    // Call the function
    calculator.add(1, 2);

    // The spy tracked everything!
    expect(calculator.add).toHaveBeenCalled();
    expect(calculator.add).toHaveBeenCalledWith(1, 2);
    expect(calculator.add).toHaveBeenCalledTimes(1);

    // Call it again
    calculator.add(5, 10);

    // Spy still tracking
    expect(calculator.add).toHaveBeenCalledTimes(2);
    expect(calculator.add).toHaveBeenCalledWith(5, 10);
  });
});
```

2. Returning Values (The Impersonator)

```
describe('Spies - Return Values', () => {
  it('should spy and return fake value', () => {
    const apiService = new ApiService();

    // Spy AND control what it returns
    // Real getData() makes HTTP call - we don't want
that!
    spyOn(apiService,
'getData').and.returnValue('mocked data');

    const result = apiService.getData();

    // Returns our fake data, not real API data
    expect(result).toBe('mocked data');
    expect(apiService.getData).toHaveBeenCalled();
  });

  it('should return different values on multiple
calls', () => {
    const service = new RandomService();

    // Return different values each call
    spyOn(service, 'getRandom')
      .and.returnValues(1, 2, 3);

    expect(service.getRandom()).toBe(1);
    expect(service.getRandom()).toBe(2);
    expect(service.getRandom()).toBe(3);
  });
});
```

3. Calling Through (The Transparent Spy)

What is callThrough?

`callThrough()` is a spy behavior that lets the **real method execute** while still tracking all calls. Think of it as a "transparent spy" - it observes without interfering.

The Key Difference:

- Without `callThrough`: Spy **replaces** the method, returns `undefined` by default
- With `callThrough`: Spy **wraps** the method, lets it run normally

When to Use callThrough:

- ✅ You want to verify a method was called, but need its real logic to run
- ✅ Testing integration between methods
- ✅ You need the actual return value from the method
- ✅ Method has side effects you want to happen (writes to DOM, updates state)

When NOT to Use:

- ❌ Method calls external services (use mock instead)
- ❌ Method has expensive operations (defeats purpose of mocking)
- ❌ You want to control the return value (use `returnValue` instead)

```
describe('Spies - Call Through', () => {
  it('should call real method but still track', () => {
    const calculator = {
      add: (a: number, b: number) => {
        console.log('Real add called!');
        return a + b;
      }
    };

    // Spy but let real method run
    spyOn(calculator, 'add').and.callThrough();

    const result = calculator.add(2, 3);

    // Real method ran - we get real result
    expect(result).toBe(5);
    // But we still tracked the call!
    expect(calculator.add).toHaveBeenCalledWith(2, 3);
  });

  it('demonstrates difference: with vs without
callThrough', () => {
    const service = {
      getData: () => 'real data'
    };

    // WITHOUT callThrough - method is replaced
    spyOn(service, 'getData');
    expect(service.getData()).toBeUndefined(); //
Returns undefined!

    // WITH callThrough - method runs normally
    spyOn(service, 'getData').and.callThrough();
    expect(service.getData()).toBe('real data'); //
Returns real value!
```

```
  });
});
```

Real-World Example:

```
class UserService {
  formatName(firstName: string, lastName: string):
string {
    return `${firstName} ${lastName}`;
  }

  getDisplayName(user: User): string {
    // We want to verify this calls formatName
    return this.formatName(user.firstName,
user.lastName);
  }
}

describe('UserService', () => {
  it('should call formatName with correct params', ()
=> {
    const service = new UserService();

    // Spy on formatName but let it run (we need the
real formatting)
    spyOn(service, 'formatName').and.callThrough();

    const result = service.getDisplayName({
      firstName: 'John',
      lastName: 'Doe'
    });
    // Verify formatName was called
    
expect(service.formatName).toHaveBeenCalledWith('John', 
'Doe');

    // Still get real result from formatName
    expect(result).toBe('John Doe');
  });
});
```

Visual Mental Model:

```
Without callThrough:
─────────────────
Original Method → [BLOCKED by Spy] → Returns 
undefined/mock value

With callThrough:
─────────────────
Original Method → [Spy observes] → Method runs → 
Returns real value
                       ↓
                  Tracks call
```

4. Simulating Errors (The Saboteur)

```
describe('Spies - Throwing Errors', () => {
  it('should spy and throw error', () => {
    const service = new DatabaseService();
    // Make method throw an error
    spyOn(service, 'query').and.throwError('Database
connection failed');

    // Test error handling
    expect(() => service.query('SELECT *'))
      .toThrowError('Database connection failed');
  });
});
```

5. Fake Implementations (The Actor)

```
describe('Spies - Custom Implementations', () => {
  it('should use custom logic', () => {
    const service = new CalculationService();

    // Provide completely custom implementation
    spyOn(service, 'calculate').and.callFake((x:
number) => {
      return x * 2; // Custom logic for testing
    });

    expect(service.calculate(5)).toBe(10);
  });
});
```

Creating Spy Objects with createSpyObj - The Full Mock:

What is createSpyObj?

`createSpyObj` is a Jasmine utility that creates a **fake object** with multiple spy methods in one go. Instead of creating a real service instance and spying on individual methods, you create a complete mock object from scratch.

The Problem it Solves:

Without `createSpyObj`, mocking a service is tedious:

```
// ❌ The hard way - manual mocking
const mockService = {
  getUser: jasmine.createSpy('getUser'),
  saveUser: jasmine.createSpy('saveUser'),
  deleteUser: jasmine.createSpy('deleteUser'),
  updateUser: jasmine.createSpy('updateUser')
};

mockService.getUser.and.returnValue({ id: 1, name:
'John' });
mockService.saveUser.and.returnValue(true);
// ... configure each spy
```

With `createSpyObj`, it's one line:

```
// ✅ The easy way - createSpyObj
const mockService = jasmine.createSpyObj('UserService',
[
  'getUser', 'saveUser', 'deleteUser', 'updateUser'
]);
```

Syntax Breakdown:

```
jasmine.createSpyObj(baseName, methodNames,
propertyNames?)

// baseName: String - Name for debugging (shows in
error messages)
// methodNames: String[] - Array of method names to
create as spies
// propertyNames: String[] - (Optional) Array of
property names to create as spies
```

Basic Example - Step by Step:

```
describe('UserComponent with Mocked Service', () => {
  it('should load user data', () => {
    // STEP 1: Create a spy object
    // This creates an object with 3 spy methods
    const mockUserService =
jasmine.createSpyObj('UserService', [
      'getUser',    // Creates mockUserService.getUser
as a spy
      'saveUser',   // Creates mockUserService.saveUser
as a spy
      'deleteUser'  // Creates
mockUserService.deleteUser as a spy
    ]);

    // At this point, mockUserService looks like:
    // {
    //   getUser: Spy,
    //   saveUser: Spy,
    //   deleteUser: Spy
    // }

    // STEP 2: Configure what each spy returns
    // By default, spies return undefined
    // We configure them to return what we want
    mockUserService.getUser.and.returnValue({
      id: 1,
      name: 'John Doe'
    });

    // STEP 3: Use the mock in your component
    const component = new
UserComponent(mockUserService);
    component.loadUser(1);

    // STEP 4: Verify interactions
    // The spy tracked all the calls!

expect(mockUserService.getUser).toHaveBeenCalledWith(1)
;
    expect(component.user.name).toBe('John Doe');
  });
});
```

What createSpyObj Actually Creates:

```
const mockService = jasmine.createSpyObj('MyService',
['method1', 'method2']);

// Under the hood, this is equivalent to:
const mockService = {
  method1: jasmine.createSpy('method1'),
  method2: jasmine.createSpy('method2')
};

// Each method is a fully functional spy that:
// 1. Can be configured (.and.returnValue,
.and.throwError, etc.)
// 2. Tracks all calls
// 3. Can be verified with expectations
```

Advanced: createSpyObj with Properties:

You can also spy on properties (getters/setters):

```
// Service with both methods and properties
class ConfigService {
  apiUrl = 'https://api.example.com';
  timeout = 5000;

  getConfig(): Config { }
  setConfig(config: Config): void { }
}

// Mock with methods AND properties
const mockConfigService = jasmine.createSpyObj(
  'ConfigService',
  ['getConfig', 'setConfig'],  // Methods
  ['apiUrl', 'timeout']          // Properties (optional
3rd param)
);

// Configure property values
```

```
mockConfigService.apiUrl = 'https://test.com';
mockConfigService.timeout = 1000;

// Configure method behaviors
mockConfigService.getConfig.and.returnValue({ /* config
*/ });
```

Alternative Syntax - Object with Return Values:

You can configure return values during creation:

```
// Configure return values immediately
const mockService = jasmine.createSpyObj('UserService',
{
  getUser: { id: 1, name: 'John' },      // getUser()
returns this
  saveUser: true,                         // saveUser()
returns true
  deleteUser: undefined                   //
deleteUser() returns undefined
});

// No need for .and.returnValue!
```

TypeScript Type Safety:

For better TypeScript support, use type assertion:

```
const mockUserService =
jasmine.createSpyObj<UserService>(
  'UserService',
  ['getUser', 'saveUser', 'deleteUser']
);

// Now TypeScript knows the types!
mockUserService.getUser.and.returnValue({ id: 1, name:
'John' });
```

Common Pattern in Angular Tests:

```
describe('MyComponent', () => {
  let component: MyComponent;
  let fixture: ComponentFixture<MyComponent>;
  let mockUserService: jasmine.SpyObj<UserService>;  //
Type it properly

  beforeEach(() => {
    // Create mock service
    mockUserService = jasmine.createSpyObj<UserService>
(
      'UserService',
      ['getUser', 'saveUser', 'deleteUser']
    );

    // Configure TestBed with mock
    TestBed.configureTestingModule({
      imports: [MyComponent],
      providers: [
        { provide: UserService, useValue:
mockUserService }
      ]
    });

    fixture = TestBed.createComponent(MyComponent);
    component = fixture.componentInstance;
  });

  it('should load user', () => {
    mockUserService.getUser.and.returnValue(of({ id: 1,
name: 'John' }));

    component.loadUser();

    expect(mockUserService.getUser).toHaveBeenCalled();
  });
});
```

createSpyObj vs spyOn - When to Use Which?

Use `createSpyObj` when:

- ✅ Creating a completely fake object from scratch
- ✅ Mocking an entire service/class
- ✅ Testing with dependency injection
- ✅ You don't need the real implementation at all

Use `spyOn` when:

- ✅ You have a real object and want to spy on specific methods
- ✅ You want to keep some methods real and spy on others
- ✅ Testing a specific method in isolation

```
// createSpyObj - Create fake from scratch
const mock = jasmine.createSpyObj('Service',
['method1', 'method2']);

// spyOn - Spy on real object
const real = new RealService();
spyOn(real, 'method1');
```

Real-World Comparison:

```
// Scenario 1: Component depends on UserService
// Use createSpyObj - cleaner for dependency injection
const mockUserService =
jasmine.createSpyObj('UserService', ['getUsers']);
TestBed.configureTestingModule({
  providers: [{ provide: UserService, useValue:
mockUserService }]
});

// Scenario 2: Testing a method that calls another
method in same class
// Use spyOn - spy on the actual instance
it('should call helper method', () => {
  const component = fixture.componentInstance;
  spyOn(component, 'helperMethod');

  component.mainMethod();

  expect(component.helperMethod).toHaveBeenCalled();
});
```

Complete Example - Before and After:

```
// ❌ BEFORE createSpyObj - Verbose and error-prone
describe('PaymentComponent (Manual Mock)', () => {
  it('should process payment', () => {
    const mockPaymentService = {
      processPayment:
jasmine.createSpy('processPayment'),
      validateCard: jasmine.createSpy('validateCard'),
      sendReceipt: jasmine.createSpy('sendReceipt')
    };


mockPaymentService.processPayment.and.returnValue(of({
success: true }));

mockPaymentService.validateCard.and.returnValue(true);

mockPaymentService.sendReceipt.and.returnValue(of(true)
);

    // ... test code
  });
});

// ✅ AFTER createSpyObj - Clean and concise
describe('PaymentComponent (createSpyObj)', () => {
  it('should process payment', () => {
    const mockPaymentService =
jasmine.createSpyObj('PaymentService', {
      processPayment: of({ success: true }),
      validateCard: true,
      sendReceipt: of(true)
    });

    // ... test code (much cleaner!)
  });
});
```

Debugging Tips:

The `baseName` parameter is crucial for debugging:

```
// Good - descriptive name
const mock = jasmine.createSpyObj('UserService',
['getUser']);
// Error: "Expected spy UserService.getUser to have
been called"

// Bad - no name
const mock = jasmine.createSpyObj('', ['getUser']);
// Error: "Expected spy .getUser to have been called"
(confusing!)
```

Summary:

`createSpyObj` is your go-to tool for creating complete mock objects in Jasmine. It:

- Creates a fake object with spy methods
- Saves you from repetitive spy creation
- Works perfectly with Angular's dependency injection
- Provides type safety with TypeScript

- Makes tests cleaner and more maintainable

Key Takeaway: Think of `createSpyObj` as a "mock factory" that instantly creates a fake object with all the methods you need, already set up as spies.

Real-World Scenario - Payment Processing:

```
describe('PaymentProcessor', () => {
  it('should process payment and send confirmation', ()
=> {
    // ARRANGE
    // We need to mock both dependencies
    const mockBankApi = jasmine.createSpyObj('BankAPI',
['charge']);
    const mockEmailService =
jasmine.createSpyObj('EmailService', ['send']);

    // Configure mock responses
    mockBankApi.charge.and.returnValue({
      success: true,
      transactionId: 'TXN-12345'
    });

    const processor = new PaymentProcessor(mockBankApi,
mockEmailService);

    // ACT
    const result = processor.processPayment({
      amount: 100,
      cardNumber: '4111111111111111',
      email: 'customer@example.com'
    });
    // ASSERT
    // Verify bank was charged correctly
    expect(mockBankApi.charge).toHaveBeenCalledWith({
      amount: 100,
      cardNumber: '4111111111111111'
    });

    // Verify email was sent

expect(mockEmailService.send).toHaveBeenCalledWith({
      to: 'customer@example.com',
      subject: 'Payment Confirmation',
      transactionId: 'TXN-12345'
    });

    expect(result.success).toBe(true);
  });

  it('should handle bank errors gracefully', () => {
    const mockBankApi = jasmine.createSpyObj('BankAPI',
['charge']);
    const mockEmailService =
jasmine.createSpyObj('EmailService', ['send']);

    // Simulate bank API failure
    mockBankApi.charge.and.returnValue({
      success: false,
      error: 'Insufficient funds'
    });

    const processor = new PaymentProcessor(mockBankApi,
mockEmailService);
    const result = processor.processPayment({
      amount: 100,
      cardNumber: '4111111111111111',
      email: 'customer@example.com'
    });

    // Should not send confirmation email on failure

expect(mockEmailService.send).not.toHaveBeenCalled();
    expect(result.success).toBe(false);
    expect(result.error).toBe('Insufficient funds');
  });
});
```

Spy Cheat Sheet:

```
// Basic spying
spyOn(obj, 'method')

// Spy behaviors
.and.returnValue(value)      // Return specific value
.and.returnValues(v1, v2)    // Return different values
each call
.and.callThrough()            // Call real method
.and.throwError('error')      // Throw error
.and.callFake((args) => {})   // Custom implementation
.and.stub()                   // Do nothing (default)

// Verification
expect(spy).toHaveBeenCalled()
expect(spy).toHaveBeenCalledWith(args)
expect(spy).toHaveBeenCalledTimes(n)
expect(spy).not.toHaveBeenCalled()

// Create spy object
jasmine.createSpyObj('Name', ['method1', 'method2'])
```

When to Use Spies vs Real Objects:

- ✅ Use spies for: API calls, database queries, external services
- ✅ Use spies for: Expensive operations (file I/O, calculations)
- ✅ Use spies for: Verifying interactions
- ❌ Don't spy on: Simple logic you're actually testing
- ❌ Don't spy on: Everything (over-mocking makes brittle tests)

Exercise:

Create comprehensive tests using spies for:

1. A `NotificationService` that calls `EmailAPI` and `SMSApi`
2. A `FileUploader` that calls `StorageService.upload()`
3. A `CacheManager` that calls `LocalStorage` and `SessionStorage`
4. Test both success and failure scenarios for each

Week 2 Checkpoint

- ☐ Can write describe/it test suites
- ☐ Know 10+ Jasmine matchers
- ☐ Can use beforeEach/afterEach correctly

- [ ] Understand spies and mocking

Week 2 Mini-Project

Create a `StringUtils` class with 5 methods and write comprehensive unit tests:

```
class StringUtils {
  capitalize(str: string): string { }
  reverse(str: string): string { }
  isPalindrome(str: string): boolean { }
  countWords(str: string): number { }
  truncate(str: string, maxLength: number): string { }
}
```

Phase 3: Angular Testing Utilities (Week 3-4)

Learning Objectives

- Master TestBed for Angular testing
- Understand component fixtures
- Debug tests effectively
- Work with Angular's testing APIs

Topics to Study

3.1 TestBed Fundamentals (Day 1-2)

What is TestBed? Understanding the Foundation

Think of TestBed as a "mini Angular application" created just for your tests.

The Challenge Without TestBed:
Angular components don't work in isolation. They need:

- Dependency injection to work
- Change detection to update views
- Compiler to process templates
- Zone.js for async operations
- The entire Angular framework context

You can't just do `const component = new MyComponent()` like a regular class because Angular components rely on framework features.

TestBed's Solution:
TestBed creates a complete Angular testing module with all the infrastructure your components need, but isolated from your real application.

Mental Model:

```
Your Real App:
├── AppModule (production)
├── All your modules
└── Real services

TestBed (per test file):
├── TestingModule (isolated)
├── Only what you configure
└── Mock services
```

Basic Setup - Step by Step:

```
import { TestBed } from '@angular/core/testing';
import { MyComponent } from './my-component';

describe('MyComponent', () => {
  // Run BEFORE each test
  beforeEach(() => {
    // Create a testing module (like NgModule, but for
tests)
    TestBed.configureTestingModule({
      imports: [MyComponent] // Standalone component
      // For non-standalone: declarations:
[MyComponent]
    });

    // Angular compiles templates, sets up DI, etc.
    // This happens automatically when you call
createComponent
  });

  it('should create', () => {
    // Create an instance of the component
    // This gives us a ComponentFixture
    const fixture =
TestBed.createComponent(MyComponent);

    // Get the actual component instance
    const component = fixture.componentInstance;

    // Verify component was created
    expect(component).toBeTruthy();
  });
});
```

Why beforeEach?

We configure TestBed in `beforeEach()` so each test gets a fresh, isolated testing environment. This prevents tests from affecting each other.

```
describe('MyComponent', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [MyComponent]
    });
  });

  it('test 1', () => {
    // Gets fresh TestBed
  });

  it('test 2', () => {
    // Gets fresh TestBed (test 1 didn't contaminate
it)
  });
});
```

TestBed Methods - Detailed Explanation:

1. configureTestingModule() - The Setup

```
TestBed.configureTestingModule({
  // Import standalone components, pipes, directives,
modules
  imports: [
    HttpClientTestingModule,  // Mock HTTP
    MyComponent,              // Standalone component
    ReactiveFormsModule       // For reactive forms
  ],

  // Provide services (or mock them)
  providers: [
    // Use real service
    RealService,

    // Replace service with mock
    { provide: MyService, useValue: mockService },

    // Use a factory function
    {
      provide: ConfigService,
      useFactory: () => new ConfigService({ apiUrl:
'test.com' })
    },

    // Use a class
    { provide: ApiService, useClass: MockApiService }
  ],

  // For non-standalone components
  declarations: [LegacyComponent],
  // Override component schemas
  schemas: [NO_ERRORS_SCHEMA] // Ignore unknown
elements
});
```

Why Configure?

Without configuration, your component has no dependencies, no modules, nothing. You're telling TestBed "here's what my component needs to work."

2. compileComponents() - The Async Compiler

```
// Only needed for components with external
templates/styles
beforeEach(async () => {
  await TestBed.configureTestingModule({
    imports: [ComponentWithExternalTemplate]
  }).compileComponents();  // Returns a Promise

  // Now templates and styles are loaded
});

// For inline templates, compileComponents() is not
needed
beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [ComponentWithInlineTemplate]
  });
  // No compileComponents needed!
});
```

Why Async?

When Angular CLI builds for production, it inlines templates. But in tests, if a component uses `templateUrl`, Angular needs to fetch that file. `compileComponents()` does this and returns a Promise.

Modern Pattern (Async/Await):

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    imports: [MyComponent]
  }).compileComponents();

  fixture = TestBed.createComponent(MyComponent);
});
```

3. createComponent() - The Instance Factory

```
// Create a component instance
const fixture = TestBed.createComponent(MyComponent);

// What you get back (ComponentFixture):
const component = fixture.componentInstance;  // The
component class
const element = fixture.nativeElement;         // Root
DOM element
const debugElement = fixture.debugElement;     //
Angular wrapper
const changeDetector = fixture.changeDetectorRef; //
Change detection
```

Behind the Scenes:

```
// When you call createComponent():
TestBed.createComponent(MyComponent)
  ↓
1. Angular creates component instance
2. Creates DOM from template
3. Sets up change detection
4. Injects dependencies
5. Wraps everything in ComponentFixture
  ↓
Returns fixture (your handle to control the component)
```

4. inject() - The Service Injector

```
it('should use service', () => {
  // Get service from TestBed's injector
  const service = TestBed.inject(MyService);

  // Now you can test the service
  expect(service).toBeTruthy();

  // Or spy on its methods
  spyOn(service, 'getData').and.returnValue('test');
});
```

Common Pattern - Inject in beforeEach:

```
describe('Component with Service', () => {
  let component: MyComponent;
  let fixture: ComponentFixture<MyComponent>;
  let service: MyService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [MyComponent],
      providers: [MyService]
    });

    // Inject service once
    service = TestBed.inject(MyService);

    // Create component
    fixture = TestBed.createComponent(MyComponent);
    component = fixture.componentInstance;
  });

  it('should call service', () => {
    spyOn(service, 'getData');
    component.loadData();
    expect(service.getData).toHaveBeenCalled();
  });
});
```

Complete Real-World Example:

```
import { TestBed, ComponentFixture } from
'@angular/core/testing';
import { HttpClientTestingModule } from
'@angular/common/http/testing';
import { UserProfileComponent } from './user-
profile.component';
import { UserService } from './user.service';
import { of } from 'rxjs';

```

```
describe('UserProfileComponent', () => {
  let component: UserProfileComponent;
  let fixture: ComponentFixture<UserProfileComponent>;
  let userService: UserService;
  let mockUserService: jasmine.SpyObj<UserService>;

  beforeEach(async () => {
    // Create mock service
    mockUserService =
jasmine.createSpyObj('UserService', ['getUser']);

    // Configure TestBed
    await TestBed.configureTestingModule({
      imports: [
        UserProfileComponent,      // Component to test
        HttpClientTestingModule    // Mock HTTP
      ],
      providers: [
        // Replace real service with mock
        { provide: UserService, useValue:
mockUserService }
      ]
    }).compileComponents();  // Compile if needed

    // Create component
    fixture =
TestBed.createComponent(UserProfileComponent);
    component = fixture.componentInstance;

    // Get the mocked service
    userService = TestBed.inject(UserService);
  });

  it('should load user on init', () => {
    // Arrange
    const mockUser = { id: 1, name: 'John' };

mockUserService.getUser.and.returnValue(of(mockUser));

    // Act
    fixture.detectChanges(); // Triggers ngOnInit

    // Assert
    expect(component.user).toEqual(mockUser);
    expect(userService.getUser).toHaveBeenCalled();
  });
});
```

TestBed Lifecycle:

```
Test File Loads
  ↓
describe() block runs
  ↓
beforeEach() runs → Configure TestBed
  ↓
it() test 1 runs
  ↓
afterEach() runs (if any) → Cleanup
  ↓
beforeEach() runs again → Fresh TestBed
  ↓
```

```
it() test 2 runs
  ↓
And so on...
```

Common Mistakes:

❌ **Creating component before configuring TestBed**:

```
describe('MyComponent', () => {
  const fixture = TestBed.createComponent(MyComponent);
// ERROR!

  beforeEach(() => {
    TestBed.configureTestingModule({...});
  });
});
```

✅ **Correct order**:

```
describe('MyComponent', () => {
  let fixture: ComponentFixture<MyComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({...});
    fixture = TestBed.createComponent(MyComponent); //
After config
  });
});
```

Exercise:

Set up TestBed for these scenarios:

1. Component with no dependencies
2. Component that uses HttpClient
3. Component with a service dependency (mock it)
4. Component with external template (use compileComponents)
5. Component with multiple service dependencies

3.2 Component Fixtures (Day 3)

Understanding Component Fixtures - Your Test Control Panel

When you create a component with `TestBed.createComponent()`, you get back a `ComponentFixture`. Think of it as a **control panel** for your component in tests.

What's in a Fixture?

A fixture is a wrapper around your component that gives you:

1. **Access to the component** (the TypeScript class)
2. **Access to the DOM** (the rendered HTML)
3. **Control over change detection** (when Angular updates the view)
4. **Debugging tools** (Angular's DebugElement)

The Fixture API - Deep Dive:

```
describe('CounterComponent', () => {
  let fixture: ComponentFixture<CounterComponent>;
  let component: CounterComponent;
  let compiled: HTMLElement;
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [CounterComponent]
    });

    // Create the fixture - our control panel
    fixture =
TestBed.createComponent(CounterComponent);

    // Extract what we need
    component = fixture.componentInstance;  // The
TypeScript class
    compiled = fixture.nativeElement;       // The
actual DOM
  });

  it('should display counter value', () => {
    // 1. ARRANGE - Set component property
    component.count = 5;

    // At this point, the DOM is NOT updated yet!
    // Angular doesn't automatically sync (for
performance)

    // 2. ACT - Trigger change detection manually
    fixture.detectChanges();

    // Now Angular runs change detection and updates
the DOM

    // 3. ASSERT - Query DOM and verify
    const display = compiled.querySelector('.count');
    expect(display?.textContent).toBe('5');
  });
});
```

Key Fixture Properties Explained:

1. componentInstance - The Brain

```
// This is your actual component class
fixture.componentInstance

// Examples:
component.count = 10;                 // Set properties
component.increment();                // Call methods
expect(component.total).toBe(15);  // Assert on values

// It's like having:
const component = new CounterComponent();
// But with Angular's DI and lifecycle
```

2. nativeElement - The Body

```
// The real DOM element (HTMLElement)
fixture.nativeElement

// Examples:
const button =
fixture.nativeElement.querySelector('button');
button.click();  // Simulate real click

const text = fixture.nativeElement.textContent;
```

```
expect(text).toContain('Hello');

// This is the ACTUAL browser DOM
```

3. debugElement - The X-Ray Vision

```
// Angular's wrapper with extra testing powers
fixture.debugElement

// Why use it?
// - Platform-agnostic (works in server-side rendering)
// - More powerful queries
// - Better event triggering
// - Access to component/directive instances

import { By } from '@angular/platform-browser';

// Query by CSS
const button =
fixture.debugElement.query(By.css('button'));

// Query by directive
const input =
fixture.debugElement.query(By.directive(MyDirective));

// Trigger events properly
button.triggerEventHandler('click', null);

// Access injected component
const child =
fixture.debugElement.query(By.directive(ChildComponent)
);
const childInstance = child.componentInstance;
```

4. changeDetectorRef - The Updater

```
// Manual control over change detection
fixture.changeDetectorRef

// Useful for OnPush components
fixture.changeDetectorRef.markForCheck();
fixture.detectChanges();

// Check if destroyed
fixture.changeDetectorRef.destroyed; // boolean
```

Key Fixture Methods Explained:

1. detectChanges() - The Most Important Method

```
fixture.detectChanges();

// What it does:
// 1. Runs change detection
// 2. Updates bindings ({{ }}, [property])
// 3. Re-renders template
// 4. Triggers lifecycle hooks (ngOnChanges, ngDoCheck)

// Example flow:
component.name = 'Alice';  // Change data
// Template still shows old name
```

```
fixture.detectChanges();   // Update view
// Template now shows 'Alice'
```

When to call detectChanges():

```
it('demonstrates when to call detectChanges', () => {
  // Initial render - call it once
  fixture.detectChanges();  // Triggers ngOnInit

  // After changing component properties
  component.title = 'New Title';
  fixture.detectChanges();  // Update view

  // After triggering events (usually automatic in real
app)
  button.click();
  fixture.detectChanges();  // Update view after click

  // After async operations complete
  tick();
  fixture.detectChanges();  // Update view after async
});
```

2. whenStable() - Wait for Async

```
// Returns a Promise that resolves when all async
operations complete
fixture.whenStable()

// Use case: component has async initialization
it('should wait for async data', waitForAsync(() => {
  component.loadData();  // Starts async operation

  fixture.whenStable().then(() => {
    // All async done
    fixture.detectChanges();
    expect(component.data).toBeDefined();
  });
}));

// Modern async/await syntax:
it('should wait for async data', async () => {
  component.loadData();

  await fixture.whenStable();  // Wait for async
  fixture.detectChanges();

  expect(component.data).toBeDefined();
});
```

3. destroy() - Cleanup

```
// Destroys component and calls ngOnDestroy
fixture.destroy();

// Usually in afterEach
afterEach(() => {
  fixture.destroy();  // Clean up
});

// Or test that ngOnDestroy was called
it('should cleanup on destroy', () => {
  spyOn(component, 'ngOnDestroy');

  fixture.destroy();

  expect(component.ngOnDestroy).toHaveBeenCalled();
});
```

4. isStable() - Check Async State

```
// Returns true if no pending async operations
fixture.isStable()

// Example:
it('should wait for stability', fakeAsync(() => {
  component.loadData();

  expect(fixture.isStable()).toBe(false);  // Async in
progress

  tick();

  expect(fixture.isStable()).toBe(true);   // Async
complete
}));
```

Real-World Example - Complete Flow:

```
@Component({
  selector: 'app-user-profile',
  standalone: true,
  template: `
```

```
    <div class="profile">
      <h1>{{ user().name }}</h1>
      <p>{{ user().email }}</p>
      <button
(click)="editMode.set(true)">Edit</button>

      @if (editMode()) {
        <input [(ngModel)]="editName" />
        <button (click)="save()">Save</button>
      }
    </div>
  `
})
export class UserProfileComponent {
  user = signal({ name: 'John', email:
'john@example.com' });
  editMode = signal(false);
  editName = '';
}

describe('UserProfileComponent - Complete Fixture
Example', () => {
  let fixture: ComponentFixture<UserProfileComponent>;
  let component: UserProfileComponent;
  let compiled: HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [UserProfileComponent, FormsModule]
    });

    fixture =
TestBed.createComponent(UserProfileComponent);
    component = fixture.componentInstance;
    compiled = fixture.nativeElement;
  });

  it('should display user info', () => {
    // Initial render
    fixture.detectChanges();

    const h1 = compiled.querySelector('h1');
    expect(h1?.textContent).toBe('John');
  });

  it('should enter edit mode on button click', () => {
    fixture.detectChanges();  // Initial render

    // Find and click button
    const button = compiled.querySelector('button');
    button?.click();

    fixture.detectChanges();  // Update after click

    // Edit input should now be visible
    const input = compiled.querySelector('input');
    expect(input).toBeTruthy();
  });

  it('should save changes', () => {
    fixture.detectChanges();

    // Enter edit mode
    component.editMode.set(true);
    fixture.detectChanges();

    // Change input
    component.editName = 'Jane';

    // Click save
    const saveButton =
Array.from(compiled.querySelectorAll('button'))
      .find(btn => btn.textContent === 'Save');
    saveButton?.click();

    fixture.detectChanges();

    // Verify update (you'd implement this in real
component)
    expect(component.editName).toBe('Jane');
  });
});
```

Fixture Mental Model:

```
ComponentFixture
├── componentInstance     → Your TypeScript class
├── nativeElement        → Real DOM (HTMLElement)
├── debugElement         → Angular wrapper
(DebugElement)
│   ├── query()          → Find elements
│   ├── triggerEvent()   → Simulate events
│   └── componentInstance → Get component from element
├── detectChanges()      → Update view
├── whenStable()            → Wait for async
└── destroy()               → Cleanup
```

Best Practices:

✅ **Always call detectChanges() after changing data**
✅ **Use debugElement.query() for reliable element selection**
✅ **Call detectChanges() once initially to trigger ngOnInit**
✅ **Clean up with destroy() in afterEach if needed**
✅ **Use whenStable() for async operations**

❌ **Don't forget detectChanges() and wonder why view doesn't update**
❌ **Don't over-call detectChanges() (once per change is enough)**
❌ **Don't test implementation details through fixture**

Exercise:
Practice with fixtures:

1. Create a counter component, test incrementing updates the view
2. Test a toggle component that shows/hides content
3. Test a form component that validates on input change
4. Test a component that loads data on init (use whenStable)
5. Test a component with OnPush change detection

3.3 DebugElement & Querying (Day 4)

DebugElement vs NativeElement:

```
// NativeElement - direct DOM access
const button =
fixture.nativeElement.querySelector('button');

// DebugElement - Angular wrapper (platform-agnostic)
const buttonDE =
fixture.debugElement.query(By.css('button'));
const button = buttonDE.nativeElement;
```

Query Methods:

```
import { By } from '@angular/platform-browser';

// CSS selector
const element = fixture.debugElement.query(By.css('.my-
class'));
const elements =
fixture.debugElement.queryAll(By.css('li'));

// Directive/Component
const directive =
fixture.debugElement.query(By.directive(MyDirective));

// Trigger events
const button =
fixture.debugElement.query(By.css('button'));
button.triggerEventHandler('click', null);
```

Best Practice: Use `By.css()` for better abstraction

Exercise: Write tests using both query approaches

3.4 Change Detection Strategies (Day 5)

Understanding Change Detection in Tests

In a real Angular application, change detection runs automatically when:

- User events occur (click, input, etc.)
- HTTP requests complete
- Timers fire (setTimeout, setInterval)
- Any async operation completes

In tests, change detection is MANUAL by default for better control and determinism.

Manual Change Detection - The Default:

```
it('should update view', () => {
  // ARRANGE
  component.value = 10;

  // At this point, Angular hasn't synced the view
  // The component property changed, but template
hasn't updated

expect(fixture.nativeElement.textContent).not.toContain
('10');

  // ACT - Trigger change detection manually
  fixture.detectChanges();

  // ASSERT - Now DOM is updated

expect(fixture.nativeElement.textContent).toContain('10
');
});
```

Why Manual Control?

- **Deterministic tests**: You control exactly when updates happen
- **Better debugging**: See exactly what changed and when
- **Performance**: Don't run unnecessary change detection cycles
- **Precision**: Test exactly what you want, when you want

Automatic Change Detection - Less Common:

```
import { ComponentFixtureAutoDetect } from
'@angular/core/testing';

describe('Component with Auto Detection', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [MyComponent],
      providers: [
        // Enable automatic change detection
        { provide: ComponentFixtureAutoDetect,
useValue: true }
      ]
    });

    fixture = TestBed.createComponent(MyComponent);
  });

  it('should update automatically', () => {
    component.value = 10;
    // Change detection runs automatically
    // No fixture.detectChanges() needed


expect(fixture.nativeElement.textContent).toContain('10
');
  });
});
```

When to Use Auto Detection?

- ✅ Testing complex async flows
- ✅ Integration tests with many components
- ❌ Most unit tests (prefer manual control)
- ❌ When you need precise timing

Testing OnPush Components - Special Handling:

OnPush components only run change detection when:

1. Input references change
2. Events fire from the component
3. Observables emit (with async pipe)
4. Manually marked for check

```
import { ChangeDetectionStrategy, Component, input }
from '@angular/core';

@Component({
  selector: 'app-user-card',
  standalone: true,
  changeDetection: ChangeDetectionStrategy.OnPush, //
OnPush!
  template: `
    <div>
      <h3>{{ name() }}</h3>
      <p>{{ email() }}</p>
    </div>
  `
})
export class UserCardComponent {
  name = input.required<string>();
  email = input<string>('');
}

describe('UserCardComponent (OnPush)', () => {
  let fixture: ComponentFixture<UserCardComponent>;
  let component: UserCardComponent;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [UserCardComponent]
    });

    fixture =
TestBed.createComponent(UserCardComponent);
    component = fixture.componentInstance;
  });

  it('should update when input changes', () => {
    // Set input (must use setInput for input signals)
    fixture.componentRef.setInput('name', 'John');
    fixture.componentRef.setInput('email',
'john@example.com');

    // For OnPush, detectChanges() should work
    fixture.detectChanges();

    const h3 =
fixture.nativeElement.querySelector('h3');
    expect(h3?.textContent).toBe('John');
  });

  it('should force update with markForCheck', () => {
    // If detectChanges() doesn't work, use
markForCheck
    fixture.componentRef.setInput('name', 'Jane');

    // Force OnPush to check
    fixture.changeDetectorRef.markForCheck();
    fixture.detectChanges();

    const h3 =
fixture.nativeElement.querySelector('h3');
    expect(h3?.textContent).toBe('Jane');
  });
});
```

Common OnPush Pitfalls:

```
describe('OnPush Component Issues', () => {
  it('WRONG - mutating object reference', () => {
    const user = { name: 'John' };
    fixture.componentRef.setInput('user', user);
    fixture.detectChanges();

    // Mutation - OnPush won't detect this!
    user.name = 'Jane';
    fixture.detectChanges();

    // Still shows 'John' because reference didn't
change
  });

  it('CORRECT - new object reference', () => {
    const user = { name: 'John' };
    fixture.componentRef.setInput('user', user);
    fixture.detectChanges();

    // New object - OnPush will detect this
    fixture.componentRef.setInput('user', { name:
'Jane' });
    fixture.detectChanges();

    // Now shows 'Jane' because reference changed
  });
});
```

Change Detection with Signals (Modern Angular):

Signals automatically mark components for check, making OnPush easier:

```
@Component({
  selector: 'app-counter',
  standalone: true,
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `<div>{{ count() }}</div>`
})
export class CounterComponent {
  count = signal(0);

  increment(): void {
    this.count.update(c => c + 1); // Auto marks for
check!
  }
}

describe('CounterComponent with Signals', () => {
  it('should update with signal changes', () => {
    fixture.detectChanges();

    // Signal update automatically marks for check
    component.increment();
    fixture.detectChanges();


expect(fixture.nativeElement.textContent).toContain('1'
);
  });
});
```

detectChanges() vs markForCheck() vs detectChanges():

```
// detectChanges() - Run change detection for this
component and children
```

```
fixture.detectChanges();

// markForCheck() - Mark this component and ancestors
as needing check
fixture.changeDetectorRef.markForCheck();

// Common pattern for OnPush:
fixture.changeDetectorRef.markForCheck(); // Mark
fixture.detectChanges();                   // Then
detect
```

Performance Tip - Minimize detectChanges() Calls:

```
// ❌ BAD - Multiple calls
it('inefficient test', () => {
  component.firstName = 'John';
  fixture.detectChanges();

  component.lastName = 'Doe';
  fixture.detectChanges();

  component.age = 30;
  fixture.detectChanges();
});

// ✅ GOOD - Single call
it('efficient test', () => {
  component.firstName = 'John';
  component.lastName = 'Doe';
  component.age = 30;
  fixture.detectChanges(); // One call for all changes
});
```

Debugging Change Detection:

```
it('should debug change detection', () => {
  // Log before
  console.log('Before:',
fixture.nativeElement.textContent);

  component.value = 10;

  // Log after change (but before detection)
  console.log('After change:',
fixture.nativeElement.textContent);

  fixture.detectChanges();

  // Log after detection
  console.log('After detection:',
fixture.nativeElement.textContent);

  // Check if component is marked as dirty
  console.log('Needs check:',
fixture.changeDetectorRef['_view'].state & 8);
});
```

Exercise:

1. Test a default component with manual change detection
2. Test an OnPush component with signal inputs

3. Test an OnPush component with object inputs (test both mutation and new reference)
4. Enable automatic change detection and observe the difference
5. Create a component that updates multiple properties and optimize detectChanges() calls

3.5 Async Testing (Day 6-7)

Understanding Async Testing - The Challenge

JavaScript is single-threaded and uses the event loop for async operations:

- `setTimeout` / `setInterval` (timers)
- `Promise` / `async/await` (microtasks)
- HTTP requests
- DOM events

In regular code, these operations take real time. In tests, **we want them instant**.

The Problem:

```
// ❌ This test will fail!
it('broken async test', () => {
  let value = false;

  setTimeout(() => {
    value = true;
  }, 1000);

  expect(value).toBe(true); // FAILS! Runs immediately,
before timeout
});
```

Angular provides two solutions: **fakeAsync** and **waitForAsync**

Solution 1: fakeAsync & tick - Control Time Itself

`fakeAsync` creates a special zone where you control time. `tick()` advances the virtual clock.

Basic Example - setTimeout:

```
import { fakeAsync, tick } from
'@angular/core/testing';

it('should handle setTimeout', fakeAsync(() => {
  let value = false;

  // Start a 1-second timer
  setTimeout(() => {
    value = true;
  }, 1000);

  // Time hasn't passed yet
  expect(value).toBe(false);

  // Advance time by 999ms
  tick(999);
  expect(value).toBe(false); // Still false
  // Advance time by 1ms more (total 1000ms)
  tick(1);
  expect(value).toBe(true); // Now true!
}));
```

What tick() Does:

```
Real World:            Test with fakeAsync:
─────────────          ─────────────

Start                  Start
  ↓ (1000ms wait)        ↓ tick(1000) - instant!
Callback runs          Callback runs
```

Testing Intervals:

```
it('should handle setInterval', fakeAsync(() => {
  let count = 0;

  const intervalId = setInterval(() => {
    count++;
  }, 100);

  expect(count).toBe(0);

  tick(100); // First interval
  expect(count).toBe(1);

  tick(200); // Two more intervals
  expect(count).toBe(3);

  clearInterval(intervalId);
  tick(1000); // No more changes
  expect(count).toBe(3);
}));
```

Testing Debounce:

```
import { debounceTime } from 'rxjs/operators';

it('should debounce search', fakeAsync(() => {
  const searchResults: string[] = [];

  component.search$
    .pipe(debounceTime(300))
    .subscribe(term => searchResults.push(term));

  // Type quickly
  component.search$.next('a');
  tick(100);
  component.search$.next('ab');
  tick(100);
  component.search$.next('abc');

  // Nothing emitted yet (debounced)
  expect(searchResults.length).toBe(0);

  // Wait for debounce
  tick(300);

  // Now it emits
  expect(searchResults).toEqual(['abc']);
}));
```

flush() - Fast Forward to the End:

```
it('should flush all pending timers', fakeAsync(() => {
  let value = false;

  setTimeout(() => { value = true; }, 5000);

  // Instead of tick(5000), just flush everything
  flush();

  expect(value).toBe(true);
}));
```

Testing Promises (Microtasks):

```
it('should handle promises', fakeAsync(() => {
  let resolved = false;

  Promise.resolve().then(() => {
    resolved = true;
  });

  // Promises are microtasks - need to flush them
  tick(); // Or flushMicrotasks()

  expect(resolved).toBe(true);
}));
```

```
it('should handle async/await', fakeAsync(async () => {
  const promise = Promise.resolve('data');

  const result = await promise;
  tick(); // Flush microtasks after await

  expect(result).toBe('data');
}));
```

Real Component Example:

```
@Component({
  selector: 'app-notification',
  standalone: true,
  template: `
    @if (visible()) {
      <div class="notification">{{ message() }}</div>
    }
  `
})
export class NotificationComponent {
  visible = signal(false);
  message = signal('');

  show(msg: string, duration: number = 3000): void {
    this.message.set(msg);
    this.visible.set(true);

    setTimeout(() => {
      this.visible.set(false);
    }, duration);
  }
}

describe('NotificationComponent', () => {
  it('should auto-hide after duration', fakeAsync(() =>
{
    fixture.detectChanges();

    // Show notification for 3 seconds
    component.show('Test message', 3000);
    fixture.detectChanges();

    // Should be visible

expect(fixture.nativeElement.textContent).toContain('Te
st message');

    // Advance time
    tick(2999);
    fixture.detectChanges();
    expect(component.visible()).toBe(true); // Still
visible

    tick(1);
    fixture.detectChanges();
    expect(component.visible()).toBe(false); // Now
hidden
  }));
});
```

Solution 2: waitForAsync & whenStable - Wait for Real Async

`waitForAsync` (formerly `async`) creates an async test zone and waits for all async operations to complete.

Basic Example:

```
import { waitForAsync } from '@angular/core/testing';

it('should handle async operations', waitForAsync(() =>
{
  component.loadData(); // Starts async operation

  // Wait for ALL async operations to complete
  fixture.whenStable().then(() => {
    // Now everything is done
    fixture.detectChanges();
    expect(component.data).toBeDefined();
  });
}));
```

Modern Async/Await Syntax:

```
it('should handle async operations', async () => {
  component.loadData();

  await fixture.whenStable(); // Clean syntax
  fixture.detectChanges();

  expect(component.data).toBeDefined();
});
```

Testing Component with Async ngOnInit:

```
@Component({
  selector: 'app-user-list',
  standalone: true,
  template: `
    @for (user of users(); track user.id) {
      <div>{{ user.name }}</div>
    }
  `
})
export class UserListComponent implements OnInit {
  users = signal<User[]>([]);
  private userService = inject(UserService);

  ngOnInit(): void {
    this.userService.getUsers().subscribe(users => {
      this.users.set(users);
    });
  }
}

describe('UserListComponent', () => {
  it('should load users on init', async () => {
    const mockUsers = [{ id: 1, name: 'John' }];

mockUserService.getUsers.and.returnValue(of(mockUsers))
;

    // ngOnInit runs, starts async operation
    fixture.detectChanges();

    // Wait for observable to complete
    await fixture.whenStable();
    fixture.detectChanges();

    expect(component.users()).toEqual(mockUsers);
  });
});
```

Testing HTTP Requests with waitForAsync:

```
it('should fetch data from API', waitForAsync(() => {
  component.fetchData();

  fixture.whenStable().then(() => {
    const req = httpMock.expectOne('/api/data');
    req.flush({ data: 'test' });

    fixture.detectChanges();
    expect(component.data).toBeDefined();
  });
}));
```

---

fakeAsync vs waitForAsync - When to Use Which?

Use fakeAsync when:

- ✅ Testing timers (setTimeout, setInterval)
- ✅ Testing debounce/throttle
- ✅ You need precise time control
- ✅ Testing time-based logic
- ✅ RxJS operators with time (delay, debounce)

Use waitForAsync when:

- ✅ Testing promises
- ✅ Testing observables without time operators
- ✅ Component initialization with async data
- ✅ You don't need time control
- ✅ Complex async flows

Can't use fakeAsync for:

- ❌ XHR requests (use HttpClientTestingModule)
- ❌ Some third-party libraries
- ❌ WebSockets

Comparison Table:

```
// fakeAsync - Control time
it('test timers', fakeAsync(() => {
  setTimeout(() => { }, 1000);
  tick(1000); // Advance virtual clock
}));

// waitForAsync - Wait for completion
it('test promises', waitForAsync(() => {
  Promise.resolve();
  fixture.whenStable().then(() => {
    // Done when stable
  });
}));
```

Common Patterns & Pitfalls

❌ Forgetting tick() for microtasks:

```
it('broken promise test', fakeAsync(() => {
  let resolved = false;
  Promise.resolve().then(() => resolved = true);

  // Missing tick()!
  expect(resolved).toBe(true); // FAILS
}));

it('fixed promise test', fakeAsync(() => {
  let resolved = false;
  Promise.resolve().then(() => resolved = true);

  tick(); // Flush microtasks
  expect(resolved).toBe(true); // PASSES
}));
```

❌ Mixing fakeAsync with real HTTP:

```
// ❌ DON'T DO THIS
it('broken test', fakeAsync(() => {
  httpClient.get('/api/data').subscribe(); // Real HTTP
doesn't work!
  tick(1000);
}));

// ✅ DO THIS
it('fixed test', () => {
  httpClient.get('/api/data').subscribe();
  const req = httpMock.expectOne('/api/data');
  req.flush({ data: 'test' });
});
```

✅ **Best Practice - Component with Timer**:

```
it('should show loading spinner for at least 500ms',
fakeAsync(() => {
  component.loadData();
  fixture.detectChanges();

  // Spinner should appear immediately
  expect(component.isLoading()).toBe(true);

  // Even if data loads instantly, show spinner for
500ms
  tick(100);
  expect(component.isLoading()).toBe(true);

  tick(400); // Total 500ms
  expect(component.isLoading()).toBe(false);
}));
```

Complete Real-World Example:

```
@Component({
  selector: 'app-search',
  standalone: true,
  template: `
    <input (input)="onSearch($event)" />
    @if (isSearching()) {
      <div>Searching...</div>
    }
    @for (result of results(); track result.id) {
      <div>{{ result.name }}</div>
    }
  `
})
export class SearchComponent {
  private searchSubject = new Subject<string>();
  isSearching = signal(false);
  results = signal<SearchResult[]>([]);

  constructor() {
    this.searchSubject.pipe(
      debounceTime(300),
      tap(() => this.isSearching.set(true)),
      switchMap(term =>
this.searchService.search(term)),
      tap(() => this.isSearching.set(false))
    ).subscribe(results => this.results.set(results));
  }

  onSearch(event: Event): void {
    const term = (event.target as
HTMLInputElement).value;
    this.searchSubject.next(term);
  }
}

describe('SearchComponent - Complete Async Test', () =>
{
  it('should debounce and search', fakeAsync(() => {
    const mockResults = [{ id: 1, name: 'Result 1' }];

mockSearchService.search.and.returnValue(of(mockResults
));

    fixture.detectChanges();

    const input =
fixture.nativeElement.querySelector('input');

    // Type quickly (debounced)
    input.value = 'a';
    input.dispatchEvent(new Event('input'));
    tick(100);

    input.value = 'ab';
    input.dispatchEvent(new Event('input'));
    tick(100);

    input.value = 'abc';
    input.dispatchEvent(new Event('input'));

    // No search yet (debounced)

expect(mockSearchService.search).not.toHaveBeenCalled()
;

    // Wait for debounce
    tick(300);

    // Now search happens
```

```
expect(mockSearchService.search).toHaveBeenCalledWith('
abc');

    // Wait for observable
    tick();
    fixture.detectChanges();

    expect(component.results()).toEqual(mockResults);
  }));
});
```

Quick Reference:

```
// fakeAsync utilities
fakeAsync(() => { })        // Create fake async zone
tick(ms)                    // Advance virtual clock
flush()                     // Advance to end of all
timers
flushMicrotasks()          // Flush promises/microtasks
discardPeriodicTasks()     // Clear intervals

// waitForAsync utilities
waitForAsync(() => { })     // Create async test zone
fixture.whenStable()        // Promise that resolves
when stable
```

Exercise:

1. Test a countdown timer component (use fakeAsync + tick)
2. Test a component that loads data with a 2-second delay

3. Test debounced search input (type multiple times, only last search executes)
4. Test a notification that auto-dismisses after 5 seconds
5. Test a component that makes an HTTP call on init (use waitForAsync)

Week 3-4 Checkpoint

- [ ] Can configure TestBed correctly
- [ ] Understand component fixtures
- [ ] Can query DOM elements
- [ ] Know when to call detectChanges()
- [ ] Can test async code

Week 3-4 Project

Create and test a `TodoListComponent` with:

- Add todo functionality
- Remove todo functionality
- Toggle todo completion
- Filter todos (all/active/completed)

Phase 4: Testing Components (Week 5-6)

Learning Objectives

- Test isolated components

- Test integrated components
- Handle user interactions
- Test inputs and outputs
- Work with forms and validation

Topics to Study

4.1 Isolated Component Tests (Day 1-2)

Concept: Test component logic without rendering

```
// counter.component.ts
import { Component, signal } from '@angular/core';

@Component({
  selector: 'app-counter',
  standalone: true,
  template: `
    <div>
      <span>{{ count() }}</span>
      <button (click)="increment()">+</button>
    </div>
  `
})
export class CounterComponent {
  count = signal(0);

  increment(): void {
    this.count.update(c => c + 1);
  }
  decrement(): void {
    this.count.update(c => c - 1);
  }

  reset(): void {
    this.count.set(0);
  }
}

// counter.component.spec.ts
describe('CounterComponent (Isolated)', () => {
  let component: CounterComponent;

  beforeEach(() => {
    // No TestBed needed!
    component = new CounterComponent();
  });

  it('should start at 0', () => {
    expect(component.count()).toBe(0);
  });

  it('should increment', () => {
    component.increment();
    expect(component.count()).toBe(1);
  });

  it('should decrement', () => {
    component.increment();
    component.increment();
    component.decrement();
    expect(component.count()).toBe(1);
  });

  it('should reset', () => {
    component.count.set(10);
    component.reset();
    expect(component.count()).toBe(0);
  });
});
```

When to use: Pure logic testing, no DOM interaction needed

Exercise: Write isolated tests for 3 components

4.2 Integrated Component Tests (Day 3-4)

Concept: Test component with its template

```
describe('CounterComponent (Integrated)', () => {
  let fixture: ComponentFixture<CounterComponent>;
  let component: CounterComponent;
  let compiled: HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [CounterComponent]
    });

    fixture =
TestBed.createComponent(CounterComponent);
    component = fixture.componentInstance;
    compiled = fixture.nativeElement;
    fixture.detectChanges();
  });

  it('should display count', () => {
    const span = compiled.querySelector('span');
    expect(span?.textContent).toBe('0');
  });

  it('should increment on button click', () => {
    const button = compiled.querySelector('button');

    button?.click();
    fixture.detectChanges();

    const span = compiled.querySelector('span');
    expect(span?.textContent).toBe('1');
  });

  it('should update view when count changes', () => {
    component.count.set(5);
    fixture.detectChanges();

    const span = compiled.querySelector('span');
    expect(span?.textContent).toBe('5');
  });
});
```

Exercise: Add DOM tests to your isolated tests

4.3 Testing Inputs & Outputs (Day 5)

Component with Inputs/Outputs:

```
import { Component, input, output } from
'@angular/core';

@Component({
  selector: 'app-user-card',
  standalone: true,
  template: `
    <div class="card">
      <h3>{{ name() }}</h3>
      <p>{{ email() }}</p>
      <button (click)="handleDelete()">Delete</button>
    </div>
  `
})
export class UserCardComponent {
  name = input.required<string>();
  email = input<string>('');
  deleted = output<void>();

  handleDelete(): void {
    this.deleted.emit();
  }
}

// Testing
describe('UserCardComponent', () => {
  let fixture: ComponentFixture<UserCardComponent>;
  let component: UserCardComponent;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [UserCardComponent]
    });

    fixture =
TestBed.createComponent(UserCardComponent);
    component = fixture.componentInstance;
  });

  it('should display name input', () => {
    fixture.componentRef.setInput('name', 'John Doe');
    fixture.detectChanges();

    const h3 =
fixture.nativeElement.querySelector('h3');
    expect(h3?.textContent).toBe('John Doe');
  });

  it('should emit deleted event', () => {
    let emitted = false;
    component.deleted.subscribe(() => {
      emitted = true;
    });

    component.handleDelete();

    expect(emitted).toBe(true);
  });

  it('should emit on button click', () => {
    let deleteCount = 0;
    component.deleted.subscribe(() => deleteCount++);
    fixture.componentRef.setInput('name', 'John');
    fixture.detectChanges();

    const button =
fixture.nativeElement.querySelector('button');
    button?.click();

    expect(deleteCount).toBe(1);
  });
});
```

Exercise: Create and test a component with 3 inputs and 2 outputs

4.4 Testing Forms (Day 6-7)

Reactive Forms:

```
import { Component, signal } from '@angular/core';
import { ReactiveFormsModule, FormBuilder, Validators }
from '@angular/forms';

@Component({
  selector: 'app-login-form',
  standalone: true,
  imports: [ReactiveFormsModule],
  template: `
    <form [formGroup]="loginForm"
(ngSubmit)="onSubmit()">
      <input formControlName="email" type="email" />
      <input formControlName="password" type="password"
/>
      <button
```

```
[disabled]="loginForm.invalid">Login</button>
    </form>
  `
})
export class LoginFormComponent {
  private fb = inject(FormBuilder);

  loginForm = this.fb.group({
    email: ['', [Validators.required,
Validators.email]],
    password: ['', [Validators.required,
Validators.minLength(8)]]
  });

  onSubmit(): void {
    if (this.loginForm.valid) {
      console.log(this.loginForm.value);
    }
  }
}

// Testing
describe('LoginFormComponent', () => {
  let fixture: ComponentFixture<LoginFormComponent>;
  let component: LoginFormComponent;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [LoginFormComponent]
    });

    fixture =
TestBed.createComponent(LoginFormComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create form with empty fields', () => {

expect(component.loginForm.get('email')?.value).toBe(''
);

expect(component.loginForm.get('password')?.value).toBe
('');
  });

  it('should be invalid when empty', () => {
    expect(component.loginForm.valid).toBe(false);
  });

  it('should validate email format', () => {
    const email = component.loginForm.get('email');

    email?.setValue('invalid');
    expect(email?.hasError('email')).toBe(true);

    email?.setValue('valid@email.com');
    expect(email?.hasError('email')).toBe(false);
  });

  it('should validate password length', () => {
    const password =
component.loginForm.get('password');
```

```
    password?.setValue('short');
    expect(password?.hasError('minlength')).toBe(true);

    password?.setValue('long-enough-password');

expect(password?.hasError('minlength')).toBe(false);
  });

  it('should be valid with correct input', () => {
    component.loginForm.setValue({
      email: 'test@example.com',
      password: 'password123'
    });

    expect(component.loginForm.valid).toBe(true);
  });

  it('should disable button when form invalid', () => {
    const button =
fixture.nativeElement.querySelector('button');

    expect(button?.disabled).toBe(true);

    component.loginForm.setValue({
      email: 'test@example.com',
      password: 'password123'
    });
    fixture.detectChanges();

    expect(button?.disabled).toBe(false);
  });
});
```

Exercise: Create and test a registration form with 5+ fields

4.5 Testing User Interactions (Day 8-9)

```
describe('User Interactions', () => {
  it('should handle click events', () => {
    const button =
fixture.debugElement.query(By.css('button'));

    spyOn(component, 'handleClick');

    button.triggerEventHandler('click', null);

    expect(component.handleClick).toHaveBeenCalled();
  });

  it('should handle input changes', () => {
    const input =
fixture.nativeElement.querySelector('input');

    input.value = 'test';
    input.dispatchEvent(new Event('input'));
    fixture.detectChanges();

    expect(component.searchTerm()).toBe('test');
  });

  it('should handle form submission', () => {
    const form = fixture.nativeElement.querySelector('form');

    spyOn(component, 'onSubmit');

    form.dispatchEvent(new Event('submit'));

    expect(component.onSubmit).toHaveBeenCalled();
  });

  it('should handle keyboard events', () => {
    const input = fixture.debugElement.query(By.css('input'));

    const event = new KeyboardEvent('keyup', { key: 'Enter' });
    input.nativeElement.dispatchEvent(event);

    expect(component.submitted).toBe(true);
  });
});
```

Exercise: Test a component with complex user interactions

Week 5-6 Checkpoint

- ☐ Can write isolated component tests
- ☐ Can write integrated component tests
- ☐ Can test inputs and outputs
- ☐ Can test reactive forms

- [ ] Can test user interactions

Week 5-6 Project

Create and fully test a `ContactFormComponent` with:

- Name, email, phone, message fields
- Validation for each field
- Submit functionality
- Error display
- Success message
- Form reset

Phase 5: Testing Services & Dependencies (Week 7)

Learning Objectives

- Test services with dependencies
- Mock HTTP requests
- Test RxJS observables
- Handle error scenarios
- Test state management

Topics to Study

5.1 Service Testing Basics (Day 1)

```
// user.service.ts
import { Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class UserService {
  private users: User[] = [];

  addUser(user: User): void {
    this.users.push(user);
  }

  getUsers(): User[] {
    return [...this.users];
  }

  getUserById(id: number): User | undefined {
    return this.users.find(u => u.id === id);
  }

  removeUser(id: number): void {
    this.users = this.users.filter(u => u.id !== id);
  }
}

// user.service.spec.ts
describe('UserService', () => {
  let service: UserService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(UserService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });

  it('should add user', () => {
    const user = { id: 1, name: 'John' };

    service.addUser(user);

    expect(service.getUsers()).toContain(user);
  });

  it('should get user by id', () => {
    const user = { id: 1, name: 'John' };
    service.addUser(user);

    const found = service.getUserById(1);

    expect(found).toEqual(user);
  });

  it('should remove user', () => {
    const user = { id: 1, name: 'John' };
    service.addUser(user);

    service.removeUser(1);

    expect(service.getUsers()).not.toContain(user);
  });
});
```

Exercise: Write tests for a simple service

5.2 Testing HTTP Services (Day 2-3)

```
import { HttpClient } from '@angular/common/http';
import { Injectable, inject } from '@angular/core';
import { Observable } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class ApiService {
  private http = inject(HttpClient);
  private apiUrl = 'https://api.example.com';

  getUsers(): Observable<User[]> {
    return this.http.get<User[]>
(`${this.apiUrl}/users`);
  }

  getUserById(id: number): Observable<User> {
    return this.http.get<User>
(`${this.apiUrl}/users/${id}`);
  }

  createUser(user: Partial<User>): Observable<User> {
    return this.http.post<User>(`${this.apiUrl}/users`,
user);
  }
  updateUser(id: number, user: Partial<User>):
Observable<User> {
    return this.http.put<User>
(`${this.apiUrl}/users/${id}`, user);
  }

  deleteUser(id: number): Observable<void> {
    return this.http.delete<void>
(`${this.apiUrl}/users/${id}`);
  }
}

// Testing with HttpClientTestingModule
import { HttpClientTestingModule, HttpTestingController
} from '@angular/common/http/testing';

describe('ApiService', () => {
  let service: ApiService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule]
    });

    service = TestBed.inject(ApiService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  afterEach(() => {
    // Verify no outstanding requests
    httpMock.verify();
  });

  it('should fetch users', () => {
    const mockUsers = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Jane' }
    ];

    service.getUsers().subscribe(users => {
      expect(users).toEqual(mockUsers);
      expect(users.length).toBe(2);
    });

    const req =
httpMock.expectOne('https://api.example.com/users');
    expect(req.request.method).toBe('GET');
    req.flush(mockUsers);
  });

  it('should fetch user by id', () => {
    const mockUser = { id: 1, name: 'John' };

    service.getUserById(1).subscribe(user => {
      expect(user).toEqual(mockUser);
    });

    const req =
httpMock.expectOne('https://api.example.com/users/1');
    expect(req.request.method).toBe('GET');
    req.flush(mockUser);
  });
  it('should create user', () => {
    const newUser = { name: 'John' };
    const createdUser = { id: 1, name: 'John' };

    service.createUser(newUser).subscribe(user => {
      expect(user).toEqual(createdUser);
    });

    const req =
httpMock.expectOne('https://api.example.com/users');
    expect(req.request.method).toBe('POST');
    expect(req.request.body).toEqual(newUser);
    req.flush(createdUser);
  });

  it('should handle HTTP errors', () => {
    const errorMessage = 'User not found';

    service.getUserById(999).subscribe({
      next: () => fail('should have failed'),
      error: (error) => {
        expect(error.status).toBe(404);
        expect(error.statusText).toBe('Not Found');
      }
    });

    const req =
httpMock.expectOne('https://api.example.com/users/999')
;
    req.flush(errorMessage, { status: 404, statusText:
'Not Found' });
  });
  it('should handle network errors', () => {
    service.getUsers().subscribe({
      next: () => fail('should have failed'),
      error: (error) => {
        expect(error.error.type).toBe('error');
      }
    });

    const req =
httpMock.expectOne('https://api.example.com/users');
    req.error(new ProgressEvent('error'));
  });
});
```

Exercise: Create and test an HTTP service with full CRUD operations

5.3 Testing RxJS Observables (Day 4-5)

```
import { Injectable, inject, signal } from
'@angular/core';
import { Observable, Subject, BehaviorSubject,
debounceTime, map, filter } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class SearchService {
  private searchSubject = new Subject<string>();

  search$ = this.searchSubject.asObservable().pipe(
    debounceTime(300),
    filter(term => term.length >= 3),
    map(term => term.toLowerCase())
  );

  search(term: string): void {
    this.searchSubject.next(term);
  }
}

// Testing observables
describe('SearchService', () => {
  let service: SearchService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(SearchService);
  });

  it('should debounce search terms', fakeAsync(() => {
    const results: string[] = [];

    service.search$.subscribe(term =>
results.push(term));

    service.search('test1');
    tick(100);
    service.search('test2');
    tick(100);
    service.search('test3');
    tick(300);

    expect(results).toEqual(['test3']);
  }));
  it('should filter short terms', fakeAsync(() => {
    const results: string[] = [];

    service.search$.subscribe(term =>
results.push(term));

    service.search('ab');
    tick(300);
    service.search('abc');
    tick(300);
    service.search('abcd');
    tick(300);

    expect(results).toEqual(['abc', 'abcd']);
  }));

  it('should convert to lowercase', fakeAsync(() => {
    const results: string[] = [];

    service.search$.subscribe(term =>
results.push(term));

    service.search('TEST');
    tick(300);

    expect(results).toEqual(['test']);
  }));
});
```

Testing with TestScheduler:

```
import { TestScheduler } from 'rxjs/testing';

describe('SearchService with TestScheduler', () => {
  let scheduler: TestScheduler;

  beforeEach(() => {
    scheduler = new TestScheduler((actual, expected) =>
{
      expect(actual).toEqual(expected);
    });
  });

  it('should debounce and filter', () => {
    scheduler.run(({ cold, expectObservable }) => {
      const input$ = cold('a-b-c-d|', {
        a: 'te',
        b: 'tes',
        c: 'test',
        d: 'testi'
      });

      const expected = '300ms b 0ms c 0ms d|';
      const values = { b: 'tes', c: 'test', d: 'testi'
};

      const output$ = input$.pipe(
        debounceTime(300),
        filter(term => term.length >= 3)
      );

      expectObservable(output$).toBe(expected, values);
    });
  });
});
```

Exercise: Test a service with complex observable chains

5.4 Mocking Dependencies (Day 6)

```
@Injectable({ providedIn: 'root' })
export class AuthService {
  private http = inject(HttpClient);
  private storage = inject(StorageService);

  login(email: string, password: string):
Observable<User> {
    return this.http.post<User>('/api/login', { email,
password }).pipe(
      tap(user => this.storage.set('user', user))
    );
  }
}

// Testing with mocked dependencies
describe('AuthService', () => {
  let service: AuthService;
  let httpMock: HttpTestingController;
  let mockStorage: jasmine.SpyObj<StorageService>;

  beforeEach(() => {
    mockStorage =
jasmine.createSpyObj('StorageService', ['get', 'set',
'remove']);

    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [
        { provide: StorageService, useValue:
mockStorage }
      ]
    });

    service = TestBed.inject(AuthService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  afterEach(() => {
    httpMock.verify();
  });

  it('should login and store user', () => {
    const mockUser = { id: 1, email: 'test@example.com'
};

    service.login('test@example.com',
'password').subscribe(user => {
      expect(user).toEqual(mockUser);

expect(mockStorage.set).toHaveBeenCalledWith('user',
mockUser);
    });

    const req = httpMock.expectOne('/api/login');
    req.flush(mockUser);
  });
});
```

Exercise: Test a service with 3+ dependencies

Week 7 Checkpoint

- ☐ Can test services in isolation
- ☐ Can mock HTTP requests
- ☐ Can test observables
- ☐ Can handle async service tests
- ☐ Can mock service dependencies

Week 7 Project

Create and test a `TodoService` that:

- Fetches todos from an API
- Caches todos locally
- Filters todos by status
- Searches todos
- Handles loading/error states

Phase 6: Advanced Testing Patterns (Week 8-9)

Learning Objectives

- Test pipes and directives
- Test Angular signals
- Test routing and guards
- Test component communication
- Performance testing

Topics to Study

6.1 Testing Pipes (Day 1)

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'truncate',
  standalone: true
})
export class TruncatePipe implements PipeTransform {
  transform(value: string, limit: number = 50,
ellipsis: string = '...'): string {
    if (!value) return '';
    if (value.length <= limit) return value;
    return value.substring(0, limit) + ellipsis;
  }
}

// Testing pipes
describe('TruncatePipe', () => {
  let pipe: TruncatePipe;

  beforeEach(() => {
    pipe = new TruncatePipe();
  });

  it('should create', () => {
    expect(pipe).toBeTruthy();
  });

  it('should return empty string for null/undefined', () => {
    expect(pipe.transform('')).toBe('');
  });

  it('should not truncate short strings', () => {
    const text = 'Short text';
    expect(pipe.transform(text, 20)).toBe(text);
  });

  it('should truncate long strings', () => {
    const text = 'This is a very long text that should be truncated';
    const result = pipe.transform(text, 20);

    expect(result).toBe('This is a very long ...');
    expect(result.length).toBe(23);
  });

  it('should use custom ellipsis', () => {
    const text = 'Long text';
    const result = pipe.transform(text, 4, '---');

    expect(result).toBe('Long---');
  });
```

```
});

// Testing pipe in component
@Component({
  selector: 'app-text-display',
  standalone: true,
  imports: [TruncatePipe],
  template: `<p>{{ text | truncate:10 }}</p>`
})
export class TextDisplayComponent {
  text = 'This is a very long text';
}

describe('TextDisplayComponent with Pipe', () => {
  let fixture: ComponentFixture<TextDisplayComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [TextDisplayComponent]
    });

    fixture =
TestBed.createComponent(TextDisplayComponent);
    fixture.detectChanges();
  });

  it('should truncate text in template', () => {
    const p = fixture.nativeElement.querySelector('p');
    expect(p.textContent).toBe('This is a ...');
  });
});
```

Exercise: Create and test 3 custom pipes

6.2 Testing Directives (Day 2)

```
import { Directive, ElementRef, HostListener, input }
from '@angular/core';

@Directive({
  selector: '[appHighlight]',
  standalone: true
})
export class HighlightDirective {
  color = input<string>('yellow');

  private el = inject(ElementRef);

  @HostListener('mouseenter')
  onMouseEnter(): void {
    this.highlight(this.color());
  }

  @HostListener('mouseleave')
  onMouseLeave(): void {
    this.highlight('');
  }

  private highlight(color: string): void {
    this.el.nativeElement.style.backgroundColor =
color;
  }
}
```

```
// Testing directive
@Component({
  selector: 'app-test',
  standalone: true,
  imports: [HighlightDirective],
  template: '<div appHighlight
[color]="testColor">Test</div>'
})
class TestComponent {
  testColor = 'blue';
}

describe('HighlightDirective', () => {
  let fixture: ComponentFixture<TestComponent>;
  let div: HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [TestComponent]
    });

    fixture = TestBed.createComponent(TestComponent);
    fixture.detectChanges();
    div = fixture.nativeElement.querySelector('div');
  });

  it('should highlight on mouse enter', () => {
    div.dispatchEvent(new Event('mouseenter'));
    fixture.detectChanges();

    expect(div.style.backgroundColor).toBe('blue');
  });

  it('should remove highlight on mouse leave', () => {
    div.dispatchEvent(new Event('mouseenter'));
    div.dispatchEvent(new Event('mouseleave'));
    fixture.detectChanges();

    expect(div.style.backgroundColor).toBe('');
  });
});
```

Exercise: Create and test a structural directive

6.3 Testing Signals (Day 3-4)

```
import { Component, signal, computed, effect } from
'@angular/core';

@Component({
  selector: 'app-cart',
  standalone: true,
  template: `
    <div>
      <p>Items: {{ itemCount() }}</p>
      <p>Total: {{ total() }}</p>
      <p>Tax: {{ tax() }}</p>
      <p>Grand Total: {{ grandTotal() }}</p>
    </div>
  `
})
export class CartComponent {
  items = signal<CartItem[]>([]);
  taxRate = signal(0.1);

  itemCount = computed(() => this.items().length);

  total = computed(() =>
    this.items().reduce((sum, item) => sum + item.price
* item.quantity, 0)
  );

  tax = computed(() => this.total() * this.taxRate());

  grandTotal = computed(() => this.total() +
this.tax());

  addItem(item: CartItem): void {
    this.items.update(items => [...items, item]);
  }

  removeItem(id: number): void {
    this.items.update(items => items.filter(item =>
item.id !== id));
  }

  clear(): void {
    this.items.set([]);
  }
}

// Testing signals
describe('CartComponent', () => {
  let component: CartComponent;
  beforeEach(() => {
    component = new CartComponent();
  });

  it('should start with empty cart', () => {
    expect(component.itemCount()).toBe(0);
    expect(component.total()).toBe(0);
  });

  it('should add items', () => {
    component.addItem({ id: 1, name: 'Item 1', price:
10, quantity: 2 });

    expect(component.itemCount()).toBe(1);
    expect(component.total()).toBe(20);
  });

  it('should calculate tax', () => {
    component.addItem({ id: 1, name: 'Item 1', price:
100, quantity: 1 });

    expect(component.tax()).toBe(10);
    expect(component.grandTotal()).toBe(110);
  });

  it('should update computed values when signals
change', () => {
    component.addItem({ id: 1, name: 'Item 1', price:
100, quantity: 1 });
    expect(component.total()).toBe(100);
    component.addItem({ id: 2, name: 'Item 2', price:
50, quantity: 1 });
    expect(component.total()).toBe(150);
  });

  it('should remove items', () => {
    component.addItem({ id: 1, name: 'Item 1', price:
100, quantity: 1 });
    component.addItem({ id: 2, name: 'Item 2', price:
50, quantity: 1 });

    component.removeItem(1);

    expect(component.itemCount()).toBe(1);
    expect(component.total()).toBe(50);
  });

  it('should clear cart', () => {
    component.addItem({ id: 1, name: 'Item 1', price:
100, quantity: 1 });

    component.clear();

    expect(component.itemCount()).toBe(0);
    expect(component.total()).toBe(0);
  });
});
```

Testing effects:

```
it('should run effect on signal change', () => {
  let effectRuns = 0;

  TestBed.runInInjectionContext(() => {
    const count = signal(0);

    effect(() => {
      count(); // Read signal
      effectRuns++;
    });

    expect(effectRuns).toBe(1); // Initial run

    count.set(1);
    expect(effectRuns).toBe(2);
  });
});
```

Exercise: Test a component using signals for state management

6.4 Testing Routing (Day 5-6)

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { inject } from '@angular/core';

@Component({
  selector: 'app-navigation',
  standalone: true,
  template: `
    <button (click)="goToHome()">Home</button>
    <button
(click)="goToProfile(userId)">Profile</button>
  `
})
export class NavigationComponent {
  private router = inject(Router);
  userId = 123;

  goToHome(): void {
    this.router.navigate(['/home']);
  }

  goToProfile(id: number): void {
    this.router.navigate(['/profile', id]);
  }
}

// Testing routing
import { RouterTestingModule } from
'@angular/router/testing';
import { Location } from '@angular/common';

describe('NavigationComponent', () => {
  let fixture: ComponentFixture<NavigationComponent>;
  let component: NavigationComponent;
  let router: Router;
  let location: Location;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [
        NavigationComponent,
        RouterTestingModule.withRoutes([
          { path: 'home', component: DummyComponent },
          { path: 'profile/:id', component: 
DummyComponent }
        ])
      ]
    });

    fixture = 
TestBed.createComponent(NavigationComponent);
    component = fixture.componentInstance;
    router = TestBed.inject(Router);
    location = TestBed.inject(Location);

    fixture.detectChanges();
  });

  it('should navigate to home', fakeAsync(() => {
    component.goToHome();
    tick();

    expect(location.path()).toBe('/home');
  }));

  it('should navigate to profile with id', fakeAsync(() 
=> {
    component.goToProfile(123);
    tick();

    expect(location.path()).toBe('/profile/123');
  }));
  it('should spy on router navigate', () => {
    spyOn(router, 'navigate');

    component.goToHome();


expect(router.navigate).toHaveBeenCalledWith(['/home'])
;
  });
});
```

Testing Route Guards:

```
import { CanActivateFn } from '@angular/router';
import { inject } from '@angular/core';

export const authGuard: CanActivateFn = (route, state)
=> {
  const authService = inject(AuthService);

  if (authService.isAuthenticated()) {
    return true;
  }

  return inject(Router).createUrlTree(['/login']);
};

// Testing guard
describe('authGuard', () => {
  let guard: CanActivateFn;
  let mockAuthService: jasmine.SpyObj<AuthService>;

  let router: Router;

  beforeEach(() => {
    mockAuthService =
jasmine.createSpyObj('AuthService',
['isAuthenticated']);

    TestBed.configureTestingModule({
      imports: [RouterTestingModule],
      providers: [
        { provide: AuthService, useValue:
mockAuthService }
      ]
    });

    guard = authGuard;
    router = TestBed.inject(Router);
  });

  it('should allow access when authenticated', () => {

mockAuthService.isAuthenticated.and.returnValue(true);

    const result = TestBed.runInInjectionContext(() =>
      guard({} as any, {} as any)
    );

    expect(result).toBe(true);
  });

  it('should redirect to login when not authenticated',
() => {
```

```
mockAuthService.isAuthenticated.and.returnValue(false);

    const result = TestBed.runInInjectionContext(() =>
      guard({} as any, {} as any)
    );

    expect(result.toString()).toContain('/login');
  });
});
```

Exercise: Test a component with complex routing logic

6.5 Testing Component Communication (Day 7)

Parent-Child Communication:

```
// Parent component
@Component({
  selector: 'app-parent',
  standalone: true,
  imports: [ChildComponent],
  template: `
    <app-child
      [data]="parentData"
      (dataChanged)="handleDataChange($event)">
    </app-child>
  `
})
export class ParentComponent {
  parentData = signal('initial');
  receivedData = signal('');
  handleDataChange(data: string): void {
    this.receivedData.set(data);
  }
}

// Testing parent-child communication
describe('Parent-Child Communication', () => {
  let fixture: ComponentFixture<ParentComponent>;
  let parent: ParentComponent;
  let childDebug: DebugElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [ParentComponent, ChildComponent]
    });

    fixture = TestBed.createComponent(ParentComponent);
    parent = fixture.componentInstance;
    childDebug =
fixture.debugElement.query(By.directive(ChildComponent)
);
    fixture.detectChanges();
  });

  it('should pass data to child', () => {
    parent.parentData.set('test data');
    fixture.detectChanges();

    const child = childDebug.componentInstance;
    expect(child.data()).toBe('test data');
  });
  it('should receive data from child', () => {
    const child = childDebug.componentInstance;

    child.dataChanged.emit('changed data');

    expect(parent.receivedData()).toBe('changed data');
  });
});
```

Service-based Communication:

```
@Injectable({ providedIn: 'root' })
export class MessageService {
  private messageSubject = new Subject<string>();
  message$ = this.messageSubject.asObservable();

  sendMessage(message: string): void {
    this.messageSubject.next(message);
  }
}

describe('Service-based Communication', () => {
  let service: MessageService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(MessageService);
  });

  it('should broadcast messages', () => {
    const messages: string[] = [];

    service.message$.subscribe(msg =>
messages.push(msg));

    service.sendMessage('test1');
    service.sendMessage('test2');

    expect(messages).toEqual(['test1', 'test2']);
  });
});
```

Exercise: Test complex component interactions

6.6 Advanced TestBed Techniques (Day 8-9)

Going Beyond the Basics

Now that you're comfortable with TestBed fundamentals, let's explore advanced techniques for handling complex testing scenarios.

---

1. TestBed.overrideComponent() - Modify Components for Testing

Sometimes you need to test a component but change its configuration just for tests.

Use Cases:

- Replace a component's template
- Change component providers
- Modify component metadata

```
@Component({
  selector: 'app-user-dashboard',
  standalone: true,
  template: `
    <app-heavy-chart [data]="chartData"></app-heavy-
chart>
    <app-user-list [users]="users"></app-user-list>
  `,
  providers: [ExpensiveService]
})
export class UserDashboardComponent {
  chartData = signal([]);
  users = signal([]);
}

describe('UserDashboardComponent', () => {
  it('should override template for simpler testing', ()
=> {
    TestBed.configureTestingModule({
      imports: [UserDashboardComponent]
    });

    // Override the template - remove heavy components
    TestBed.overrideComponent(UserDashboardComponent, {
      set: {
        template: `<div>Simplified Dashboard</div>`
      }
    });

    const fixture =
TestBed.createComponent(UserDashboardComponent);
```

```
    fixture.detectChanges();


expect(fixture.nativeElement.textContent).toContain('Si
mplified Dashboard');
  });

  it('should override component providers', () => {
    const mockService =
jasmine.createSpyObj('ExpensiveService', ['getData']);

    TestBed.configureTestingModule({
      imports: [UserDashboardComponent]
    });

    // Replace component's provider
    TestBed.overrideComponent(UserDashboardComponent, {
      set: {
        providers: [
          { provide: ExpensiveService, useValue:
mockService }
        ]
      }
    });

    const fixture =
TestBed.createComponent(UserDashboardComponent);
    // Component now uses mockService instead of real
ExpensiveService
  });
});
```

Override Options:

```
TestBed.overrideComponent(MyComponent, {
  set: {
    template: '<div>New template</div>',
    templateUrl: './new-template.html',
    styleUrls: ['./new-styles.css'],
    providers: [/* new providers */],
    changeDetection: ChangeDetectionStrategy.Default
  },
  add: {
    // Add providers without removing existing ones
    providers: [AdditionalService]
  },
  remove: {
    // Remove specific providers
    providers: [ServiceToRemove]
  }
});
```

---

2. TestBed.overrideProvider() - Replace Service Implementations

More convenient than provider configuration for swapping services globally.

```
describe('App with Overridden Providers', () => {
  let mockAuthService: jasmine.SpyObj<AuthService>;

  beforeEach(() => {
    mockAuthService =
jasmine.createSpyObj('AuthService', ['login',
'logout']);
```

```
    TestBed.configureTestingModule({
      imports: [AppComponent]
    });

    // Override any AuthService provider in the entire testing module
    TestBed.overrideProvider(AuthService, {
      useValue: mockAuthService
    });
  });

  it('should use mocked auth service', () => {
    mockAuthService.login.and.returnValue(of({ success: true }));

    const fixture = TestBed.createComponent(AppComponent);
    // All components using AuthService get the mock
  });
});
```

---

3. TestBed.overrideModule() - Modify Module Configuration

When testing with NgModules (non-standalone components):

```
@NgModule({
  declarations: [MyComponent],
  imports: [ExpensiveModule],
  providers: [RealService]
})
```

```
export class MyModule {}

describe('MyComponent with Module Override', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [MyModule]
    });

    // Override module configuration
    TestBed.overrideModule(MyModule, {
      remove: {
        imports: [ExpensiveModule],
        providers: [RealService]
      },
      add: {
        imports: [MockModule],
        providers: [MockService]
      }
    });
  });
});
```

---

4. TestBed.overrideDirective/Pipe() - Replace Directives/Pipes

```
@Directive({
  selector: '[appTooltip]',
  standalone: true
})
export class TooltipDirective {
  // Complex tooltip logic with third-party library
}
```

```
describe('Component with Overridden Directive', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [MyComponent, TooltipDirective]
    });

    // Replace with simpler version for testing
    TestBed.overrideDirective(TooltipDirective, {
      set: {
        // Simplified directive behavior
      }
    });
  });
});
```

---

5. TestBed.resetTestingModule() - Clean Slate

Reset TestBed between test files or when you need a fresh start.

```
describe('Test Suite 1', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [Component1]
    });
  });
});

describe('Test Suite 2', () => {
  beforeAll(() => {
    // Clear all previous TestBed configuration
    TestBed.resetTestingModule();
  });

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [Component2]
    });
  });
});
```

When to use:

- Between major test file sections
- After complex overrides
- When experiencing test pollution

Warning: Usually not needed - TestBed resets automatically between test files.

---

6. Multi-Providers and Provider Trees

Testing Components with Multiple Providers:

```
// Shared service at root level
@Injectable({ providedIn: 'root' })
export class GlobalConfigService {
  config = { theme: 'light' };
}

// Component-level service
@Component({
  selector: 'app-settings',
  standalone: true,
  providers: [SettingsService]  // Component-level
provider
})
export class SettingsComponent {
  constructor(
    private settings: SettingsService,
    private config: GlobalConfigService
  ) {}
}

describe('Multi-Provider Testing', () => {
  it('should mock root-level provider', () => {
    const mockConfig = { config: { theme: 'dark' } };

    TestBed.configureTestingModule({
      imports: [SettingsComponent],
      providers: [
        // Override root-level service
        { provide: GlobalConfigService, useValue:
mockConfig }
      ]
    });

    const fixture =
TestBed.createComponent(SettingsComponent);
    const component = fixture.componentInstance;

    // Component gets the mock
    expect(component.config.config.theme).toBe('dark');
  });

  it('should override component-level provider', () =>
{
    const mockSettings =
jasmine.createSpyObj('SettingsService', ['save']);

    TestBed.configureTestingModule({
      imports: [SettingsComponent]
    });

    // Override component's own provider
    TestBed.overrideComponent(SettingsComponent, {
      set: {
        providers: [
          { provide: SettingsService, useValue:
mockSettings }
        ]
      }
    });

    const fixture =
TestBed.createComponent(SettingsComponent);
    // Component uses mockSettings instead of real
SettingsService
  });
});
```

---

7. Testing with ViewChild and ContentChild

```
@Component({
  selector: 'app-parent',
  standalone: true,
  imports: [ChildComponent],
  template: '<app-child #child></app-child>'
})
export class ParentComponent {
  @ViewChild('child') childComponent!: ChildComponent;

  ngAfterViewInit() {
    this.childComponent.doSomething();
  }
}

describe('ParentComponent with ViewChild', () => {
  it('should access child component', () => {
    TestBed.configureTestingModule({
      imports: [ParentComponent, ChildComponent]
    });

    const fixture =
TestBed.createComponent(ParentComponent);

    // ViewChild not available yet

expect(fixture.componentInstance.childComponent).toBeUn
defined();

    // Trigger change detection (calls ngAfterViewInit)
    fixture.detectChanges();

    // Now ViewChild is available

expect(fixture.componentInstance.childComponent).toBeDe
fined();
expect(fixture.componentInstance.childComponent).toBeIn
stanceOf(ChildComponent);
  });
});
```

---

8. Testing with NO_ERRORS_SCHEMA - Skip Unknown Elements

When you want to test a component without declaring all its child components:

```
@Component({
  selector: 'app-complex',
  standalone: true,
  template: `
    <app-header></app-header>
    <app-sidebar></app-sidebar>
    <app-content></app-content>
    <app-footer></app-footer>
  `
})
export class ComplexComponent {
  title = 'Complex App';
}

describe('ComplexComponent (Shallow)', () => {
  it('should test without child components', () => {
    TestBed.configureTestingModule({
      imports: [ComplexComponent],
      schemas: [NO_ERRORS_SCHEMA]  // Ignore unknown
elements
    });
    const fixture =
TestBed.createComponent(ComplexComponent);
    fixture.detectChanges();

    // Component created without errors
    // Child components are not rendered (empty tags)

expect(fixture.componentInstance.title).toBe('Complex
App');
  });
});
```

When to use NO_ERRORS_SCHEMA:

- ✅ Shallow component tests (test component in isolation)
- ✅ When child components are complex and not relevant to test
- ✅ Speeding up tests
- ❌ Don't use for integration tests (defeats the purpose)
- ❌ Can hide real template errors

---

9. Using CUSTOM_ELEMENTS_SCHEMA - Web Components

For Angular applications using Web Components:

```
import { CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';

describe('Component with Web Components', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [MyComponent],
      schemas: [CUSTOM_ELEMENTS_SCHEMA]  // Allow
custom elements
    });
  });

  it('should work with web components', () => {
    const fixture =
TestBed.createComponent(MyComponent);
    // <my-web-component> won't throw errors
  });
});
```

---

10. Advanced Provider Patterns

useFactory with Dependencies:

```
describe('Service with Factory Provider', () => {
  it('should inject dependencies into factory', () => {
    TestBed.configureTestingModule({
      providers: [
        ConfigService,
        {
          provide: ApiService,
          useFactory: (config: ConfigService) => {
            return new ApiService(config.apiUrl);
          },
          deps: [ConfigService]  // Factory
dependencies
        }
      ]
    });

    const apiService = TestBed.inject(ApiService);
    expect(apiService).toBeDefined();
  });
});
```

useClass for Inheritance Testing:

```
class MockApiService extends ApiService {
  override getData() {
    return of({ mock: true });
  }
}

describe('Component with Service Inheritance', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [MyComponent],
      providers: [
        { provide: ApiService, useClass: MockApiService
}
      ]
    });
  });
});
```

useExisting for Aliasing:

```
describe('Service Aliasing', () => {
  it('should alias service', () => {
    TestBed.configureTestingModule({
      providers: [
        UserService,
        // Alias: Inject UserService when someone asks
for AdminService
        { provide: AdminService, useExisting:
UserService }
      ]
    });

    const userService = TestBed.inject(UserService);
    const adminService = TestBed.inject(AdminService);

    // Same instance!
    expect(userService).toBe(adminService);
  });
});
```

---

11. Testing Platform-Specific Code

Override Platform ID:

```
import { PLATFORM_ID } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';

describe('Platform-Specific Component', () => {
  it('should test browser-specific code', () => {
    TestBed.configureTestingModule({
      imports: [MyComponent],
      providers: [
        { provide: PLATFORM_ID, useValue: 'browser' }
      ]
    });

    const fixture =
TestBed.createComponent(MyComponent);
    // Component thinks it's running in browser
  });

  it('should test server-specific code', () => {
    TestBed.configureTestingModule({
      imports: [MyComponent],
      providers: [
        { provide: PLATFORM_ID, useValue: 'server' }
      ]
    });

    const fixture =
TestBed.createComponent(MyComponent);
    // Component thinks it's running on server (SSR)
  });
});
```

---

12. Debugging TestBed Issues

Get Current TestBed Configuration:

```
it('should debug TestBed', () => {
  TestBed.configureTestingModule({
    imports: [MyComponent],
    providers: [MyService]
  });
  // Get the testing module
  const testingModule = TestBed.inject(TestBed as any);

  // Inspect injector
  const injector = TestBed.inject(Injector);
  console.log('Has MyService?', injector.get(MyService,
null) !== null);
});
```

Common TestBed Errors and Solutions:

```
// ❌ Error: "Can't resolve all parameters"
// Solution: Make sure all dependencies are provided

// ❌ Error: "No provider for Service!"
// Solution: Add to providers array
TestBed.configureTestingModule({
  providers: [MissingService]
});

// ❌ Error: "Component not found"
// Solution: Add to imports (standalone) or
declarations (NgModule)

// ❌ Error: "Template parse errors"
// Solution: Import required modules (FormsModule,
etc.) or use NO_ERRORS_SCHEMA
```

---

13. Performance: Reusing TestBed Configuration

For faster tests, reuse TestBed configuration across tests:

```
describe('Test Suite with Shared Config', () => {
  // Configure once for all tests
  beforeAll(() => {
    TestBed.configureTestingModule({
      imports: [CommonModule, SharedModule],
      providers: [SharedService]
    });
  });

  // Don't call TestBed.configureTestingModule again
  // Create components directly
  it('test 1', () => {
    const fixture =
TestBed.createComponent(Component1);
  });

  it('test 2', () => {
    const fixture =
TestBed.createComponent(Component2);
  });
});
```

⚠️ **Warning**: Only do this when tests truly share the same configuration. Otherwise, tests can affect each other.

---

14. Complete Real-World Example

```
@Component({
  selector: 'app-dashboard',
  standalone: true,
  imports: [CommonModule, ChartModule, GridModule],
  template: `
    <div class="dashboard">
      <app-chart [data]="chartData()"></app-chart>
      <app-grid [items]="items()"></app-grid>
      <app-notifications></app-notifications>
    </div>
  `,
  providers: [DashboardService]
})
export class DashboardComponent implements OnInit {
  chartData = signal<any[]>([]);
  items = signal<any[]>([]);

  private dashboardService = inject(DashboardService);
  private analyticsService = inject(AnalyticsService);

  ngOnInit() {
    this.loadData();
    this.analyticsService.trackPageView('dashboard');
  }

  loadData() {
    this.dashboardService.getData().subscribe(data => {
      this.chartData.set(data.chart);
      this.items.set(data.items);
    });
  }
}

describe('DashboardComponent - Advanced Testing', () =>
{
  let component: DashboardComponent;
  let fixture: ComponentFixture<DashboardComponent>;
  let mockDashboardService:
jasmine.SpyObj<DashboardService>;
  let mockAnalyticsService:
jasmine.SpyObj<AnalyticsService>;

  beforeEach(() => {
    mockDashboardService =
jasmine.createSpyObj('DashboardService', ['getData']);
    mockAnalyticsService =
jasmine.createSpyObj('AnalyticsService',
['trackPageView']);

    mockDashboardService.getData.and.returnValue(of({
      chart: [1, 2, 3],
      items: ['a', 'b', 'c']
    }));

    TestBed.configureTestingModule({
      imports: [DashboardComponent],
      schemas: [NO_ERRORS_SCHEMA]  // Ignore child
components
    });

    // Override component's provider
    TestBed.overrideComponent(DashboardComponent, {
      set: {
        providers: [
          { provide: DashboardService, useValue:
mockDashboardService }
```

```
        ]
      }
    });

    // Override root-level provider
    TestBed.overrideProvider(AnalyticsService, {
      useValue: mockAnalyticsService
    });

    fixture =
TestBed.createComponent(DashboardComponent);
    component = fixture.componentInstance;
  });

  it('should load data on init', () => {
    fixture.detectChanges();  // Triggers ngOnInit


expect(mockDashboardService.getData).toHaveBeenCalled()
;
    expect(component.chartData()).toEqual([1, 2, 3]);
    expect(component.items()).toEqual(['a', 'b', 'c']);
  });

  it('should track analytics', () => {
    fixture.detectChanges();


expect(mockAnalyticsService.trackPageView).toHaveBeenCa
lledWith('dashboard');
```

```
  });
});
```

Key Takeaways:

✅ **Use overrides** for modifying components/providers without changing source code
✅ **NO_ERRORS_SCHEMA** for shallow testing (test component in isolation)
✅ **Multi-provider scenarios** require understanding provider hierarchy
✅ **Factory providers** give you full control over service creation
✅ **resetTestingModule()** rarely needed (TestBed resets automatically)
✅ **Performance optimization** with shared TestBed configuration (use cautiously)

Exercise:

1. Create a component with a complex template and test it with NO_ERRORS_SCHEMA
2. Override a component's template to simplify testing
3. Test a component with both root-level and component-level providers
4. Use overrideProvider to replace a service in an entire test suite
5. Create a factory provider that depends on two other services

---

Week 8-9 Checkpoint

- ☐ Can test pipes
- ☐ Can test directives
- ☐ Can test signals

- [ ] Can test routing
- [ ] Can test component communication
- [ ] Can use advanced TestBed techniques

Week 8-9 Project

Create a master-detail view with:

- List component showing items
- Detail component showing selected item
- Service for communication
- Routing between views
- Full test coverage

Phase 7: Real-World Application (Week 10-12)

Learning Objectives

- Apply all learned concepts
- Write tests for existing code
- Practice TDD workflow
- Achieve high code coverage
- Optimize test performance

Topics to Study

7.1 Test-Driven Development (Day 1-3)

TDD Workflow:

```
1. Write a failing test (RED)
2. Write minimal code to pass (GREEN)
3. Refactor while keeping tests green (REFACTOR)
```

Example TDD Session:

```
// Step 1: Write failing test
describe('TodoService', () => {
  it('should add todo', () => {
    const service = new TodoService();

    service.addTodo({ title: 'Test', completed: false
});

    expect(service.getTodos().length).toBe(1);
  });
});

// Step 2: Implement minimal code
export class TodoService {
  private todos: Todo[] = [];

  addTodo(todo: Todo): void {
    this.todos.push(todo);
  }

  getTodos(): Todo[] {
    return this.todos;
  }
}

// Step 3: Refactor
export class TodoService {
  private todos: Todo[] = [];

  addTodo(todo: Todo): void {
    this.todos = [...this.todos, todo]; // Immutable
  }

  getTodos(): ReadonlyArray<Todo> {
    return this.todos;
  }
}
```

Exercise: Build a feature using strict TDD

7.2 Testing Legacy Code (Day 4-5)

Strategy for Testing Existing Code:

1. **Identify critical paths**
2. **Add characterization tests** (document current behavior)
3. **Refactor for testability**
4. **Add comprehensive tests**

Example:

```
// Legacy code (hard to test)
export class LegacyService {
  getData() {
    const http = new HttpClient();
    const result = http.get('api/data');
    const processed = this.process(result);
    localStorage.setItem('data', processed);
    return processed;
  }
}

// Refactored for testing
export class ImprovedService {
  constructor(
    private http: HttpClient,
    private storage: Storage
  ) {}

  getData(): Observable<Data> {
    return this.http.get<Data>('api/data').pipe(
      map(data => this.process(data)),
      tap(data => this.storage.setItem('data', data))
    );
  }

  private process(data: Data): Data {
    // Extracted for testing
    return data;
  }
}
```

Exercise: Add tests to an untested component

7.3 Code Coverage Strategies (Day 6-7)

Running Coverage:

```
# Generate coverage report
ng test --code-coverage --watch=false

# Coverage with threshold
ng test --code-coverage --watch=false \
  --codeCoverageExclude='**/*.spec.ts'
```

Setting Coverage Thresholds:

```
// karma.conf.js
coverageReporter: {
  check: {
    global: {
      statements: 80,
      branches: 80,
      functions: 80,
      lines: 80
    }
  }
}
```

Coverage Best Practices:

- Focus on critical business logic (90%+)
- Don't obsess over 100% coverage

- Exclude trivial code from coverage
- Use coverage to find untested paths
- Balance coverage with test quality

Exercise: Achieve 80%+ coverage on a module

7.4 Test Performance Optimization (Day 8)

Strategies:

1. **Use isolated tests when possible**

```
// Faster - no TestBed
it('should calculate', () => {
  const service = new CalculatorService();
  expect(service.add(2, 2)).toBe(4);
});

// Slower - full TestBed setup
it('should calculate', () => {
  TestBed.configureTestingModule({});
  const service = TestBed.inject(CalculatorService);
  expect(service.add(2, 2)).toBe(4);
});
```

2. **Share TestBed configuration**

```
// Shared module for tests
const testingModule = {
  imports: [CommonModule, HttpClientTestingModule],
  providers: [CommonService]
};

// Reuse in multiple test files
beforeEach(() => {
  TestBed.configureTestingModule(testingModule);
});
```

3. **Use NO_ERRORS_SCHEMA for shallow tests**

```
TestBed.configureTestingModule({
  imports: [MyComponent],
  schemas: [NO_ERRORS_SCHEMA] // Ignore unknown
elements
});
```

4. **Parallel test execution** (Karma)

```
// karma.conf.js
browsers: ['ChromeHeadless'],
concurrency: Infinity
```

Exercise: Optimize slow test suite

7.5 Best Practices & Patterns (Day 9-10)

Test Naming:

```
// Good
it('should display error message when login fails', ()
=> {});
it('should disable submit button when form is invalid',
() => {});

// Bad
it('test1', () => {});
it('works', () => {});
```

Test Organization:

```
describe('UserComponent', () => {
  describe('Initialization', () => {
    it('should create component', () => {});
    it('should load user data', () => {});
  });

  describe('User Actions', () => {
    it('should save changes', () => {});
    it('should cancel editing', () => {});
  });

  describe('Validation', () => {
    it('should validate email format', () => {});
    it('should validate required fields', () => {});
  });
});
```

DRY in Tests:

```
// Helper functions
function createUser(overrides?: Partial<User>): User {
  return {
    id: 1,
    name: 'John',
    email: 'john@example.com',
    ...overrides
  };
}

// Reusable setup
function setupComponent(config?:
Partial<ComponentConfig>) {
  TestBed.configureTestingModule({
    imports: [MyComponent],
    ...config
  });
  return TestBed.createComponent(MyComponent);
}
```

Avoid Test Interdependence:

```
// Bad - tests depend on each other
describe('Counter', () => {
  let counter = 0;

  it('should start at 0', () => {
    expect(counter).toBe(0);
  });

  it('should increment', () => {
    counter++; // Changes state for next test!
    expect(counter).toBe(1);
  });
});
```

```
// Good - independent tests
describe('Counter', () => {
  let counter: number;

  beforeEach(() => {
    counter = 0; // Fresh state for each test
  });

  it('should start at 0', () => {
    expect(counter).toBe(0);
  });

  it('should increment', () => {
    counter++;
    expect(counter).toBe(1);
  });
});
```

Week 10-12 Final Project

Build a complete Todo application with full test coverage:

Features:

- Create, read, update, delete todos
- Filter by status (all/active/completed)
- Search todos
- Persist to API
- Loading states

- Error handling
- Routing (list/detail views)

Requirements:

- 80%+ code coverage
- All components tested
- All services tested
- Integration tests for key workflows
- Proper test organization
- Use TDD for new features

Deliverables:

- Fully tested application
- Coverage report
- Testing documentation
- Lessons learned write-up

Resources & References

Official Documentation

- Angular Testing Guide
- Jasmine Documentation
- Karma Documentation

Books

- **"Angular Development with TypeScript"** by Yakov Fain
- **"Testing Angular Applications"** by Jesse Palmer
- **"The Art of Unit Testing"** by Roy Osherove

Online Courses

- Angular Testing Masterclass (Angular University)
- Testing Angular Applications (Pluralsight)
- Unit Testing for TypeScript & JavaScript (Udemy)

Articles & Blogs

- Angular Blog Testing Articles
- Testing best practices on Medium
- Dev.to Angular testing tutorials

Tools

- Karma - Test runner
- Jasmine - Testing framework
- Angular Testing Library
- Wallaby.js - Live test runner

Video Tutorials

- YouTube: Angular Testing Tutorials

- FreeCodeCamp: Angular Testing Course
- Academind: Angular Testing Videos

Practice Projects

Beginner Level

1. **Calculator App**
    - Basic arithmetic operations
    - History of calculations
    - Clear functionality
2. **Todo List**
    - Add/remove todos
    - Mark as complete
    - Filter todos
3. **Weather Display**
    - Fetch weather data
    - Display current weather
    - Format temperature

Intermediate Level

4. **Blog Platform**

- Create/edit/delete posts
- Comment system
- User authentication
- Search functionality

5. **E-commerce Cart**
   - Add/remove items
   - Quantity management
   - Price calculations
   - Discount codes
6. **Dashboard**
   - Multiple widgets
   - Data visualization
   - User preferences
   - Responsive layout

Advanced Level

7. **Task Management System**
   - Projects and tasks
   - Drag and drop
   - Real-time updates
   - Complex filtering
8. **Social Media Feed**

- Infinite scroll
- Like/comment
- Real-time notifications
- Image uploads

9. **Analytics Platform**
    - Complex data processing
    - Charts and graphs
    - Export functionality
    - User roles and permissions

Common Pitfalls & Solutions

Pitfall 1: Not Calling detectChanges()

```
// Problem
it('should update view', () => {
  component.value = 10;
  // View not updated!
  expect(compiled.textContent).toContain('10'); //
FAILS
});

// Solution
it('should update view', () => {
  component.value = 10;
  fixture.detectChanges(); // Trigger change detection
  expect(compiled.textContent).toContain('10'); //
PASSES
});
```

Pitfall 2: Testing Implementation Details

```
// Bad - tests internal implementation
it('should call private method', () => {
  spyOn(component as any, 'privateMethod');
  component.publicMethod();

expect(component['privateMethod']).toHaveBeenCalled();
});

// Good - tests public behavior
it('should process data correctly', () => {
  const result = component.publicMethod(input);
  expect(result).toEqual(expectedOutput);
});
```

Pitfall 3: Not Mocking Dependencies

```
// Bad - uses real HTTP
it('should fetch data', () => {
  service.getData().subscribe(/* ... */); // Real API
call!
});

// Good - mocks HTTP
it('should fetch data', () => {
  service.getData().subscribe(/* ... */);
  const req = httpMock.expectOne('/api/data');
  req.flush(mockData);
});
```

Pitfall 4: Async Test Timing Issues

```
// Problem
it('should load data', () => {
  component.loadData();
  expect(component.data).toBeDefined(); // FAILS -
async not complete
});

// Solution
it('should load data', fakeAsync(() => {
  component.loadData();
  tick(); // Wait for async operations
  expect(component.data).toBeDefined(); // PASSES
}));
```

Pitfall 5: Shared State Between Tests

```
// Problem
describe('Service', () => {
  const service = new Service(); // Shared instance!

  it('test 1', () => {
    service.value = 10;
  });
  it('test 2', () => {
    expect(service.value).toBe(0); // FAILS - still 10
from test 1
  });
});

// Solution
describe('Service', () => {
  let service: Service;

  beforeEach(() => {
    service = new Service(); // Fresh instance per test
  });

  it('test 1', () => {
    service.value = 10;
  });

  it('test 2', () => {
    expect(service.value).toBe(0); // PASSES - new
instance
  });
});
```

Pitfall 6: Brittle Selectors

```
// Bad - fragile DOM queries
const element =
fixture.nativeElement.querySelector('div >
span.class1.class2');
```

```
// Good - stable queries
const element =
fixture.nativeElement.querySelector('[data-
testid="user-name"]');
```

Pitfall 7: Over-Mocking

```
// Bad - mocking everything
const mockService = {
  method1: () => 'mock',
  method2: () => 'mock',
  method3: () => 'mock',
  // Loses real behavior
};

// Good - mock only what's needed
const service = new RealService(mockDependency);
```

Progress Tracking

Week 1: ◻ Testing Fundamentals

- ◻ Understand testing importance
- ◻ Know testing pyramid
- ◻ Use AAA pattern
- ◻ Generate coverage reports

Week 2: ☐ Jasmine Basics

- ☐ Write describe/it suites
- ☐ Use 10+ matchers
- ☐ Setup/teardown correctly
- ☐ Create spies and mocks

Week 3-4: ☐ Angular Testing Utilities

- ☐ Configure TestBed
- ☐ Use component fixtures
- ☐ Query DOM elements
- ☐ Handle async tests

Week 5-6: ☐ Testing Components

- ☐ Isolated tests
- ☐ Integrated tests
- ☐ Test inputs/outputs
- ☐ Test forms
- ☐ Test interactions

Week 7: ☐ Testing Services

- ☐ Basic service tests
- ☐ Mock HTTP requests
- ☐ Test observables

- ☐ Mock dependencies

Week 8-9: ☐ Advanced Patterns

- ☐ Test pipes
- ☐ Test directives
- ☐ Test signals
- ☐ Test routing
- ☐ Test communication

Week 10-12: ☐ Real-World Application

- ☐ Practice TDD
- ☐ Test legacy code
- ☐ Achieve coverage goals
- ☐ Optimize performance
- ☐ Complete final project

Certification & Next Steps

After Completing This Plan

You should be able to:

- ✅ Write comprehensive unit tests for any Angular application
- ✅ Use Jasmine effectively

- ✅ Apply TDD methodology
- ✅ Achieve and maintain high code coverage
- ✅ Debug failing tests efficiently
- ✅ Mentor others in testing practices

Next Learning Steps

1. **Integration Testing**
    - Test multiple components together
    - Test feature modules
    - Test complex workflows
2. **E2E Testing**
    - Learn Cypress or Playwright
    - Test user journeys
    - Visual regression testing
3. **Performance Testing**
    - Load testing
    - Stress testing
    - Profiling test suites
4. **Advanced Topics**
    - Snapshot testing
    - Visual testing
    - Mutation testing

- Contract testing

5. **CI/CD Integration**

    - Automate test runs
    - Set up test pipelines
    - Configure test reporting
    - Implement quality gates

Conclusion

Unit testing is a critical skill for Angular developers. This 12-week plan provides a structured path from fundamentals to advanced testing techniques. Remember:

- **Practice consistently** - Testing is a skill that improves with practice
- **Write tests first** - Try TDD for new features
- **Refactor fearlessly** - Good tests enable confident refactoring
- **Share knowledge** - Help others learn testing
- **Stay curious** - Keep learning new testing patterns and tools

Happy testing! 🧪✨

Last Updated: November 4, 2025

Version: 1.0