TEST DRIVEN DEVELOPMENT

TEST, CODE, REFACTOR, REPEAT

HI, MY NAME IS DIMI.

Developer
Ordina Belgium
https://github.com/DimiDeKerf

TEST

WHAT IS A TEST?

Process insuring the quality, performance and reliability of something is in place before bringing it into widespread use.

TEST

- Requirements are satisfied
- Responds correctly to all input (happy and unhappy flows)
- Performance

TEST DRIVEN DEVELOPMENT

Software development process relying on a short development cycle. Requirements are translated in tests, then the software is improved to only pass the new tests.

RED

Write a test that fails, verifying a specific case of the software.

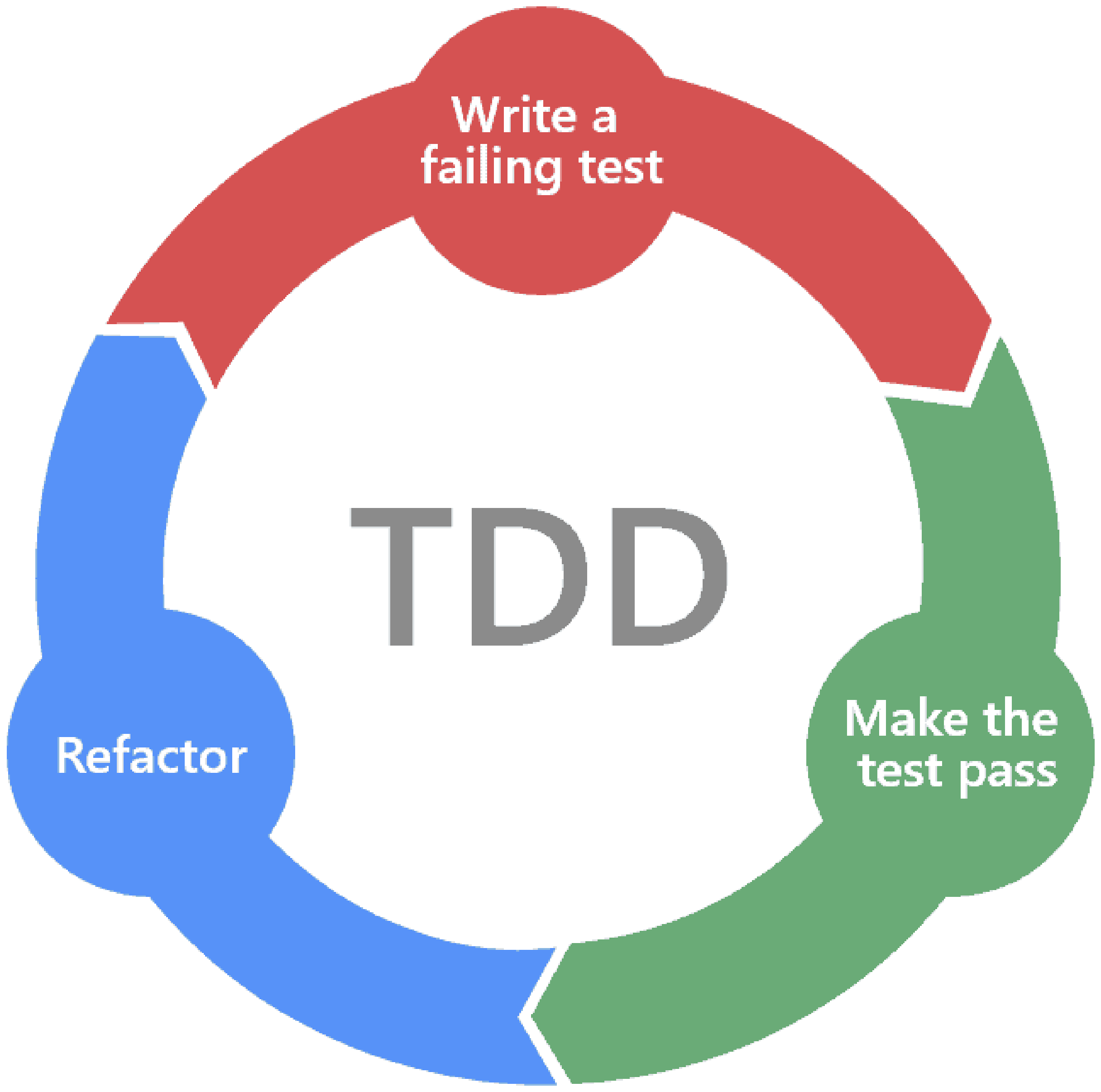
GREEN

Implement the minimal amount of code in order to only pass the test.

REFACTOR

Refactor code, while making sure all tests still pass.

TEST CYCLE

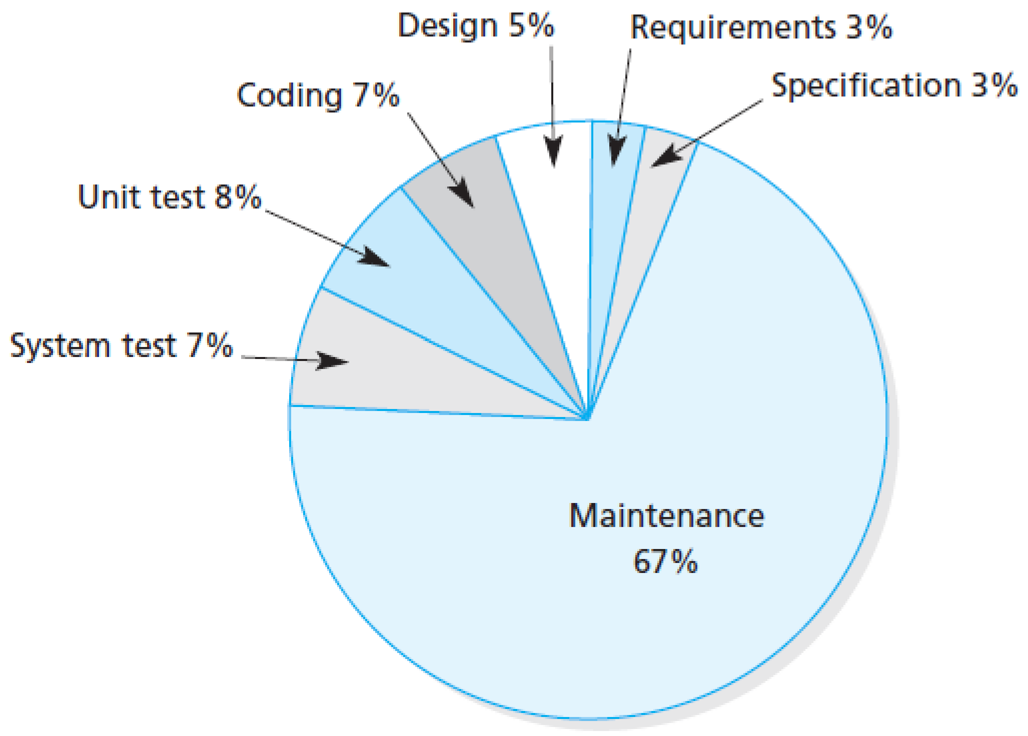


Red - Green - Refactor

TEST DRIVEN DEVELOPMENT

DEVELOPMENT COSTS

Around 65% of costs goes to maintenance



WHY MAINTENANCE IS SO EXPENSIVE?

- Code entropy
- Isolated ownership
- No validation on code

LEGACY CODE

Code inherited from somebody else or older version of software

LEGACY CODE

Usually no validation in place, making it hard to perform changes without breaking anything.

HOW CAN TDD HELP OUT?

BUSINESS BENEFITS

- Verify requirements by defined tests
- Test failures prevents features from not working anymore with new version
- Lower maintenance costs

DEVELOPER BENEFITS

- Design first mentality
- Avoid over-engineering
- Increase momentum
- Confidence to refactor
- Tests provide documentation for unit's API

CUSTOMER BENEFITS

- Focus on real needs of customer
- Fast improvement cycle

TESTING APPLICATIONS

DIFFERENT WAYS OF TESTING APPLICATIONS

- Unit testing
- Integration testing
- Acceptance testing

UNIT TESTS

Small, fast tests. Part of the development cycle.

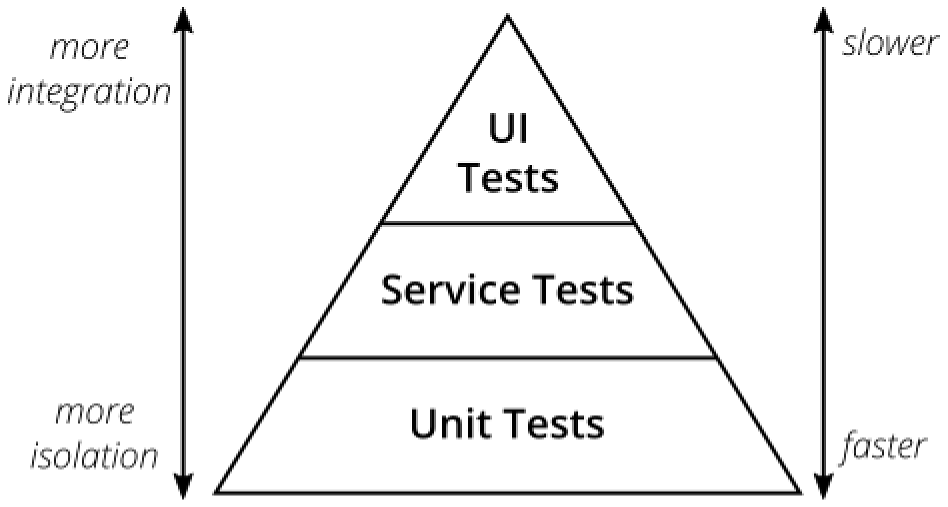
INTEGRATION TESTS

Verify behavior between different units. Slower than unit tests

ACCEPTANCE TESTS

Test from the user point of view. Expensive tests to run

TESTING PYRAMID



UNIT TESTS

- Tests a single unit, class
- No IO, network calls,...
- Mock dependencies

MOCKING

Remove dependencies to create a controlled environment for our unit.

Mocks will have a predefined behavior.

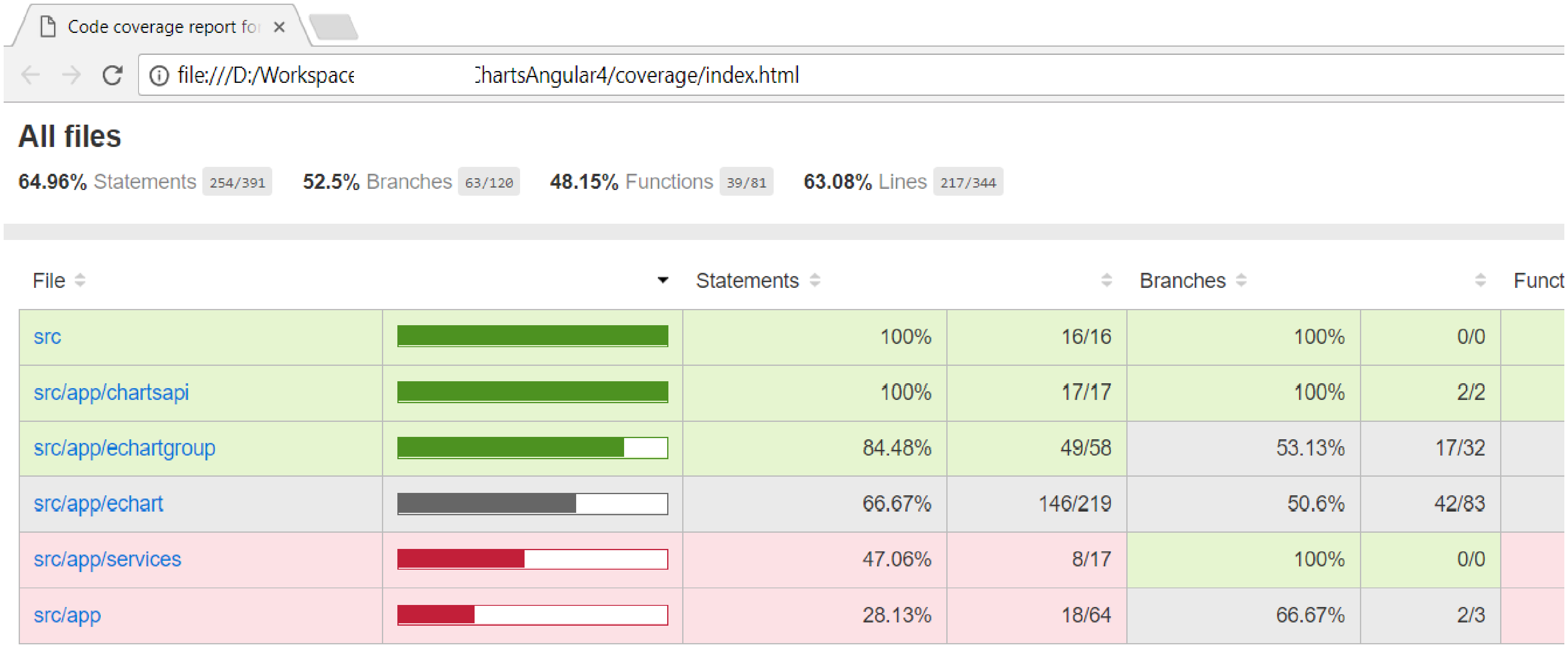
TEST API

Write unit tests for your public methods. The private methods should be hit by calling the public ones.

CODE COVERAGE

Metrics to determine the number of code lines that are validated by tests.

CODE COVERAGE



All files

64.96% Statements 254/391 **52.5%** Branches 63/120 **48.15%** Functions 39/81 **63.08%** Lines 217/344

File		Statements		Branches		Funct
src		100%	16/16	100%	0/0	
src/app/chartsapi		100%	17/17	100%	2/2	
src/app/echartgroup		84.48%	49/58	53.13%	17/32	
src/app/echart		66.67%	146/219	50.6%	42/83	
src/app/services		47.06%	8/17	100%	0/0	
src/app		28.13%	18/64	66.67%	2/3	

TESTING FRAMEWORKS

JASMINE

Behavior-driven development framework.

MOCHA

Alternative testing framework. Relative to JUnit.

KARMA

Test runner. Runs your tests in multiple browsers

JEST

Test framework developed by Facebook. Combination of both a testing framework and runner.

WRITING TESTS

TEST FILE

Test are usually found in *.spec.ts or *.test.ts files

TEST STRUCTURE

```
describe('My awesome test', () => {
    it('passes everytime', () => {
        expect(true).toBe(true);
    });
});
```

TEST SUITE

Collection of several related tests.

DESCRIBE

Describes high level summary of test suite. Describe blocks can be nested.

IT

Describes a test spec.

Usually has one expectation.

SETUP

```
describe('My awesome test', () => {
    beforeAll(() => {
        // Runs before test suit
    });

    beforeEach(() => {
        // Runs before every test
    });

    it('passes everytime', () => {
        expect(true).toBe(true);
    });
});
```

TEARDOWN

Cleanup code, so test can run independently of eachother.

```
describe('My awesome test', () => {
    afterEach(() => {
        // Runs after every test
    });

    afterAll(() => {
        // Runs after test suit
    });

    it('passes everytime', () => {
        expect(true).toBe(true);
    });
});
```

VERIFICATION

Verify if result matches expectation.

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

expect(service.getName()).toBe('John');

expect({}).toBeDefined();

expect(null).toBeNull();
```

MOCK DEPENDENCIES

```
class Calculator {
    sum(first, second) {
        return first + second;
    }
}

class MyComponent {
    constructor(private calculator: Calculator) {}

    sum(first, second) {
        return this.calculator.sum(first, second);
    }
}
```

MOCK DEPENDENCIES

```
describe('MyComponent', () => {
    let myComponent;
    let calculatorMock;

    beforeEach(() => {
        calculatorMock = {
            sum: () => 4
        };
        myComponent = new MyComponent(calculatorMock);
    });

    it('sums up 2 and 2', () => {
        expect(myComponent.sum(2, 2)).toBe(4);
    });
});
```

MOCK DEPENDENCIES

Most test frameworks come with mocking functionality.

```
describe('MyComponent', () => {
    let myComponent;
    let calculatorMock;

    beforeEach(() => {
        calculatorMock = jasmine.createSpyObj('Calculator', ['sum']);
        myComponent = new MyComponent(calculatorMock);
    });
});
```

SPY

Stub a function and spy if function has been called during test.

SPY

```
describe('MyComponent', () => {
    let myComponent;
    let calculatorMock;

    beforeEach(() => {
        calculatorMock = jasmine.createSpyObj('Calculator', ['sum']);
        myComponent = new MyComponent(calculatorMock);
    });

    it('calls the sum function', () => {
        spyOn(calculatorMock, 'sum');

        myComponent.sum(2, 2);

        expect(calculatorMock.sum).toHaveBeenCalled();
        expect(calculatorMock.sum).toHaveBeenCalledWith(2, 2); // Verify parameters
    });
});
```

SPY

Configure spies to return predefined value.

```
describe('MyComponent', () => {
    let myComponent;
    let calculatorMock;

    beforeEach(() => {
        calculatorMock = jasmine.createSpyObj('Calculator', ['sum']);
        myComponent = new MyComponent(calculatorMock);
    });

    it('sums up 2 and 2', () => {
        spyOn(calculatorMock, 'sum').and.returnValue(5);

        expect(myComponent.sum(2, 2)).toBe(5);
    });
});
```

NEST DESCRIBE BLOCKS

Scope functions in describe block

```
describe('MyComponent', () => {
    let myComponent;
    let calculatorMock;

    beforeEach(() => {
        calculatorMock = {
            sum: () => 4
        };
        myComponent = new MyComponent(calculatorMock);
    });

    describe('.sum', () => {
        it('sums up 2 and 2', () => {
            expect(myComponent.sum(2, 2)).toBe(4);
        });
    });
});
```

TEST FILE NEXT TO IMPLEMENTATION

- Easy to find
- Test reveal API of implementation
- Reminder to update test when implementation changes

EXERCISES

DEPENDENCY INJECTION

WHAT IS DEPENDENCY INJECTION?

Have an object supply the dependencies of another object by injecting them.

WITHOUT DEPENDENCY INJECTION

Without Dependency Injection, we cannot replace the calculator in our test.

```
class MyComponent {
    private calculator: Calculator;

    constructor() {
        this.calculator = new Calculator();
    }

    sum(first, second) {
        return this.calculator.sum(first, second);
    }
}
```

WITH DEPENDENCY INJECTION

With Dependency Injection, we can replace the dependency in our test.

```
class MyComponent {
    private calculator: Calculator;

    constructor(calculator: Calculator) {
        this.calculator = calculator;
    }

    sum(first, second) {
        return this.calculator.sum(first, second);
    }
}
```

ASYNCHRONOUS TESTING

ASYNCHRONOUS TESTING

Code often runs asynchronously, let our test framework know when code completes.

ASYNCHRONOUS SERVICE

```
class MyService {
    getProducts(): Promise<string[]> {
        return new Promise(resolve => {
            setTimeout(() => resolve(['Product 1', 'Product 2']), 2000)
        });
    }
}
```

ASYNCHRONOUS TESTING

Test won't wait for promise to be resolved

CALLBACKS

Call done argument inside the callback.

```
describe('MyService', () => {
    it('returns products', (done) => {
        const myService = new MyService();
        myService.getProducts().then(products => {
            expect(products.length).toBe(2);
            done();
        });
    });
});
```

PROMISE

Return a Promise.

```
describe('MyService', () => {
    it('returns products', () => {
        const myService = new MyService();
        return myService.getProducts().then(products => {
            expect(products.length).toBe(2);
        });
    });
});
```

ASYNC/AWAIT

Await for Promise to resolve.

```
describe('MyService', () => {
    it('returns products', async () => {
        const myService = new MyService();
        const products = await myService.getProducts();
        expect(products.length).toBe(2);
    });
});
```

OBSERVABLE

```
class MyService {
    getProducts(): Observable<string[]> {
        return of(['Product 1', 'Product 2']).pipe(
            delay(2000)
        );
    }
}
```

OBSERVABLE

Subscribe to an observable.

```
describe('MyService', () => {
    it('returns products', () => {
        const myService = new MyService();
        myService.getProducts().subscribe(products => {
            expect(products.length).toBe(2);
            done();
        });
    });
});
```

OBSERVABLE

Convert to a Promise.

```
describe('MyService', () => {
    it('returns products', async () => {
        const myService = new MyService();
        const products = await myService.getProducts().toPromise();
        expect(products.length).toBe(2);
    });
});
```

BEST PRACTICES

PRODUCTION CODE QUALITY

Maintain the same quality for test code as for production code. Keep it readable and maintainable.

TEST BOTH HAPPY AND UNHAPPY FLOWS

Cover negative cases as well, making sure the application can handle these cases.

TEST BOTH HAPPY AND UNHAPPY FLOWS

Cover negative cases as well, making sure the application can handle these cases.

CODE COVERAGE

Aim to write meaningful tests, not to have 100% coverage.

Independent test

Execution order of tests should not matter. Test should run independently.

UNIT TESTS DON'T COVER EVEYTHING.

Don't only rely on unit tests to prevent bugs. Make sure to have other tests and verifications in place.

TESTING IN ANGULAR

TESTING IN ANGULAR

Angular uses Jasmine and Karma by default.

Other testing frameworks can be configured.

TESTING IN ANGULAR

```
ng test
```

Launches Karma in watch mode, retriggering tests on code changes.

TESTBED

Testing utility of Angular that initializes test environment and manages dependencies.

SERVICES WITHOUT TESTBED

```
describe('ProductService', () => {
    let service;
    let httpClientMock;

    beforeEach(() => {
        httpClientMock = jasmine.createSpyObj('HttpClient', ['get']);
        service = new ProductService();
    });

    describe('.getProducts', () => {
        const expectedProducts: Product[] = [
            {id: '1', name: 'Product1', description: '', productCode: 'P1'},
            {id: '2', name: 'Product2', description: '', productCode: 'P2'},
        ];

        beforeEach(() => {
            httpClientMock.get.and.returnValue(of(expectedProducts))
        });

        it('returns products', async () => {
```

SERVICES WITH TESTBED

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

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

        service = TestBed.inject(ProductService);
        httpMock = injector.get(HttpTestingController);
    });

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

    describe('.getProducts', () => {
        const expectedProducts: Product[] = [
```

COMPONENTS

Testing component can be divided in class and DOM testing.

COMPONENTS

Testing component class is usually sufficient. Use TestBed to validate template.

THANKS FOR WATCHING!

and remember… Red, green, refactor!