ANGULAR TESTING

TESTING - INTRODUCTION

For testing an application in general, you need 2 functionalities:

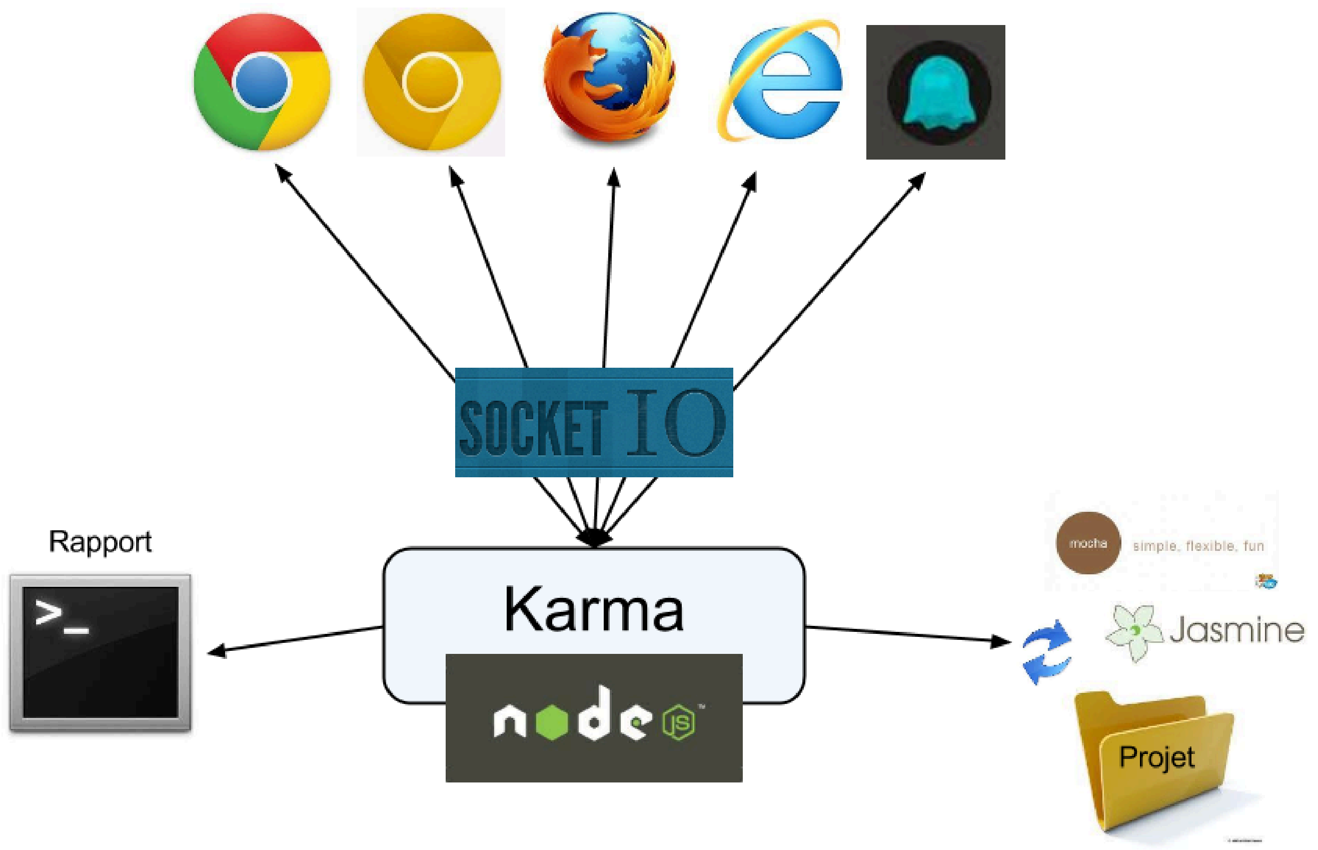
- A test runner that identifies and runs the files containing the tests
- An assertion library that verifies the expected behavior

Out of the box, Angular uses `Karma` as test runner and `Jasmine` as assertion library.

By default, test files are identified by the pattern: `*.spec.ts`.

TESTING - KARMA

- Karma is a tool that automates the execution of tests



TESTING - JASMINE

- Organize your tests using `describe` and `it` functions
- Follow the 3 steps pattern in each test: `Given`, `When`, `Then`
- Identify the thing being tested using `expect`
- Use matchers to verify the expected behavior: `toBe`, `toBeTruthy`, `toContain`, ...

```
describe('boolean variable', () => {
  let value?: boolean;

  it('should be inverted when using "!" operator', () => {
    // Given
    value = true;

    // When
    value = !value;

    // Then
    expect(value).toBe(false); // equivalent to `expect(value).toBeFalse();`
  });
});
```

TESTING - JASMINE | HOOKS

- Use hooks to setup and teardown your tests using:
  - `beforeEach`, `afterEach`, `beforeAll`, `afterAll`

```
describe('boolean variable', () => {
  let value?: boolean;

  beforeEach(() => {
    // Given
    value = true;
  });

  it('should be inverted when using "!" operator', () => {
    // When
    value = !value;

    // Then
    expect(value).not.toBeTrue(); // <-- notice the usage of `.not`
  });
});
```

TESTING - JASMINE | SPIES

- Use a spy to watch how a method is been used during the test
- Create a spy: `jasmine.createSpy` or `spyOn`
- Spy matchers: `toHaveBeenCalled`, `toHaveBeenCalledWith`, `and.returnValue`, …

```
// Given
class Counter {
  count = 0;

  increment() { this.count += 1; this.log('increment'); }

  log(message: string) { console.log('Counter:', message); }
}
const count = new Counter();
const logSpy = spyOn(count, 'log'); // <-- Spying on the `log` method

// When
count.increment();

// Then
expect(logSpy).toHaveBeenCalledWith('increment');
```

TESTING - ANGULAR ENVIRONMENT

- Angular provides a powerful testing environment called `TestBed`
- Angular testing configuration is reset for every test (executed in `beforeEach`)

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

describe('my feature', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({ /* Test setup */ });
  });

  it('should work', () =>  /* ... */ });

  it('should work too', () =>  /* ... */ });
});
```

TESTING - COMPONENTS

- Components combine an HTML template and a TypeScript class
- You should test that they work together as intended
- `TestBed` helps you create the component's host element in the browser DOM
- The `fixture` gives you access to the component instance and its host element
- In the tests you must `detectChanges` manually verifying that the DOM state is correct

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { AppComponent } from './app.component';

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

let fixture = TestBed.createComponent(AppComponent);

let component = fixture.componentInstance;
let hostElement = fixture.nativeElement;

fixture.detectChanges();
```

TESTING - COMPONENTS | STRATEGIES

Class testing:

- Pros: Easy to setup, Easy to write, Most usual way to write unit tests
- Cons: Does not make sure your component behave the way it should

DOM testing:

- Pros: Make sure your component behave exactly the way it should
- Cons: Harder to setup, Harder to write

✅ Overall, DOM testing is more robust, but require more work to setup.

TESTING - EXAMPLE 1

- A simple counter component

```
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'app-counter',
  template: '<button (click)="increment()">{{ count }}</button>'
})
export class CounterComponent {
  @Input() count = 0;
  @Output() countChange = new EventEmitter<number>();

  protected increment() {
    this.count += 1;
    this.countChange.emit(this.count);
  }
}
```

TESTING - EXAMPLE 1

- Test setup

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { CounterComponent } from './counter.component';

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

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

    fixture = TestBed.createComponent(CounterComponent);

    fixture.detectChanges(); // <-- The template state needs to be initialized manually
  });
});
```

TESTING - EXAMPLE 1

- Actual Tests (1/2)

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

it('should display 0', () => {
  // Getting element using `debugElement`
  const button = fixture.debugElement.query(By.css('button')).nativeElement;

  expect((button as HTMLButtonElement).textContent).toContain(0);
});

it('should increment the count when clicking', () => {
  // Getting element using `nativeElement`
  const button = (fixture.nativeElement as HTMLElement).querySelector('button');

  button?.click(); // <-- The class state get automatically updated
  expect(fixture.componentInstance.count).toBe(1); // <-- Class testing

  fixture.detectChanges(); // <-- The template state update needs to be triggered manually
  expect(button?.textContent).toContain(1); // <-- DOM testing
});
```

TESTING - EXAMPLE 1

- Actual Tests (2/2)

```
it('should emit output with the current count when clicking', () => {
  const emitSpy = spyOn(fixture.componentInstance.countChange, 'emit');

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

  expect(emitSpy).toHaveBeenCalledWith(1);
});
```

TESTING - EXAMPLE 2

- Component with dependency
- We're going to explore two different approaches to test this use case

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

@Component({
  selector: 'app-number-parity',
  template: `
    <app-counter [(count)]="count" />

    <span>{{ count % 2 ? 'is odd' : 'is even' }}</span>
  `,
})
export class NumberParityComponent {
  count = 0;
}
```

TESTING - EXAMPLE 2 | FIRST APPROACH

- (1/2) Test setup with explicit dependency declaration

```typescript
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { CounterComponent } from '../counter/counter.component';
import { NumberParityComponent } from './number-parity.component';

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

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [NumberParityComponent, CounterComponent] // <-- Dependency declared!
    });
    fixture = TestBed.createComponent(NumberParityComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
});
```

TESTING - EXAMPLE 2 | FIRST APPROACH

- (2/2) Actual Tests accessing the dependency (the child component instance)

```
it('should bind count to the child component', () => {
  const counterComponent: CounterComponent =
    fixture.debugElement.query(By.directive(CounterComponent)).componentInstance;

  // Accessing the child component properties
  expect(counterComponent.count).toBe(component.count);
});

it('should be "odd" when child component emits', () => {
  const counterComponent: CounterComponent =
    fixture.debugElement.query(By.directive(CounterComponent)).componentInstance;

  // Accessing the child component methods
  counterComponent.countChange.emit(1);
  fixture.detectChanges();

  const span = (fixture.nativeElement as HTMLElement).querySelector('span');
  expect(span?.textContent).toContain('odd');
});
```

TESTING - EXAMPLE 2 | SECOND APPROACH

- (1/2) Test setup allowing unknown HTML elements

```
import { CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { CounterComponent } from '../counter/counter.component';
import { NumberParityComponent } from './number-parity.component';

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

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [NumberParityComponent], // <-- `CounterComponent` not declared...
      schemas: [CUSTOM_ELEMENTS_SCHEMA], // <-- ...but unknown HTML elements are allowed
    });
    fixture = TestBed.createComponent(NumberParityComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
});
```

TESTING - EXAMPLE 2 | SECOND APPROACH

- (2/2) Actual Tests using:
  - `debugElement.properties` and `debugElement.triggerEventHandler`

```
it('should bind count to CounterComponent', () => {
  const debugElement = fixture.debugElement.query(By.css('app-counter'));

  // Accessing bindings on the child element
  expect(debugElement.properties['count']).toBe(component.count);
});

it('should be "odd" when counter emits', () => {
  const debugElement = fixture.debugElement.query(By.css('app-counter'));

  // Triggering events on the child element
  debugElement.triggerEventHandler('countChange', 1);
  fixture.detectChanges();

  const span = (fixture.nativeElement as HTMLElement).querySelector('span');
  expect(span?.textContent).toContain('odd');
});
```

THANK YOU