Tests unitarios con Jest y e2e con Cypress en Angular

Continuamos este **tutorial de Angular Avanzado** con el tema controvertido del testing. Sí, ya sé que todos deberíamos hacerlo siempre, pero también sé que no es cierto. Así que vamos a ponerle remedio eliminando excusas y facilitando las pruebas automatizadas.

> Por si hace falta su defensa: Las pruebas automáticas de código son la principal técnica de reducción de bugs y garantizan el buen funcionamiento durante un refactoring. Bueno para el usuario bueno para el programador.

Los desarrollos que hoy en día hacemos con **Angular** suelen ser de tamaño medio o grande y con una esperanza de vida y mantenimiento que se mide en años. Así que cuantas más pruebas tengamos menos miedo tendremos a cambiar el código. Y la necesidad de cambio siempre estará ahí. Veremos como *Jest* y *Cypress* nos ayudan muchísimo en la tarea.

Partiendo del mono repositorio creado usando las herramientas de Nrwl.io/ aprovecharemos las herramientas que instala y configura dos productos de última generación que facilitan la tarea. Usaremos Jest para los test unitarios y Cypress para los de integración *End to End*. Vamos a empezar por el final.

> Código asociado a este tutorial en *GitHub*: AcademiaBinaria/angular-boss

1. Test de Integración con Cypress

```
As a: developer,
  I want: to test end to end my app
  so that: I can be sure of the functions
```

Las pruebas de integración son las que tienen un retorno de inversión más inmediato. Si esta es tu primera aproximación al testing, te recomiendo empezar por aquí. Se trata de definir un escenario y pre programar el comportamiento del usuario. El sistema lo ejecutará y podrás contrastar el resultado con algún valor esperado.

Son pruebas de caja negra que interactúan con el sistema en ejecución. Todo ello automatizado y con informes visuales interactivos o en ficheros de texto con el resultado de las pruebas.

1.1 Cypress

Cypress es el equivalente a Protractor, el producto propio de Angular. La ventaja principal es que no está atado a ningún framework y por tanto lo que hagas valdrá para probar cualquier web en cualquier tecnología.

Con cada aplicación generada se crea una hermana para sus pruebas *e2e*. Esa aplicación de pruebas está configurada y lista para compilar, servir y probar su aplicación objetivo. Con Nx tenemos todo instalado, configurado y listo para ejecutarse con comandos como los siguientes que yo coloco en el `packcage.json`:

```
{
  "e2e:shop": "ng e2e shop-e2e --watch",
  "e2e:warehouse": "ng e2e warehouse-e2e --watch"
}
```

Luego se pueden lanzar desde la terminal muy cómodamente.

```
yarn e2e:shop
yarn e2e:warehouse
```

1.2 Test e2e

```
GIVEN: the shop web app
  WHEN: user visits home page
    THEN: should display welcome message
    THEN: should display welcome message from the API
```

Tu trabajo como *tester* será definir las pruebas en la carpeta `/integration`. Por ejemplo para empezar nos ofrecen el fichero `app-spec.ts` en el que yo he especificado el comportamiento deseado por mi página. La sintaxis se entiende por si misma. He seguido el convenio **GIVEN, WHEN, THEN** para especificar pruebas que podrían considerarse casi de comportamiento o aceptación.

`apps\shop-e2e\src\integration\app.spec.ts`

```
import { getGreeting } from '../support/app.po';

describe('GIVEN: the shop web app', () => {
  beforeEach(() => cy.visit('/'));
  context('WHEN: user visits home page', () => {
    it('THEN: should display welcome message', () => {
      getGreeting().contains('Hello world');
    });
    // needs the api server to run
    // yarn start:api
    it('THEN: should display welcome message from the API', () => {
      getGreeting().contains('and Welcome to api!');
    });
  });
});
```

La parte más técnica y tediosa es la que accede al *DOM* y lo mejor es tener eso a parte. En la carpeta `/support` nos sugieren que creemos utilidades para tratar con el *DOM* y de esa forma mantener los test lo más cercanos posible a un lenguaje natural de negocio. En este caso uso una aproximación libre al BDD con gherkin para mantener el espíritu de sencillez de un tutorial sobre tecnología Angular, no sobre testing.

apps\shop-e2e\src\support\app.po.ts

```
export const getGreeting = () => cy.get('h1');
```

2. Test Unitarios con Jest

Las pruebas unitarias, muy asociadas al TDD, son las mejores amigas del desarrollador. Pero, también son las más engorrosas para empezar. Así que aquí veremos una introducción sencilla para que nadie deje de hacerlas.

2.1 Jest

Jest es un framework de testing para JavaScript muy sencillo y rápido. Puedes usarlo con cualquier otro framework. Para el caso de Angular ya viene preconfigurado si usas las extensiones Nx.

Mete los siguientes scripts en el `package.json` y así tendrás a mano siempre las pruebas. Te recomiendo que desarrolles con el test unitario lanzado; es la manera más rápida de probar el código que estés tocando. Idealmente incluso con el test antes del código.

```
{
  "test:shop": "ng test shop --watch --verbose",
  "test:warehouse": "ng test warehouse --watch --verbose",
  "test:api": "ng test api --watch --verbose"
}
```

```
yarn test:shop
yarn test:warehouse
yarn test:api
```

2.2 Tests unitarios

2.2.1 Componentes

```
GIVEN: an AppComponent declared in AppModule
  WHEN: the AppModule is compiled
    THEN: should create the component
    THEN: should have a property title with value 'shop'
    THEN: should render 'Hello world' in a H1 tag
```

En este caso queremos probar una librería de componentes. Y empezamos por el componente raíz. Esta es una prueba unitarias pero con un toque de integración parcial pues necesita de otros componentes para

ejecutarse. Cuanto más arriba en la jerarquía esté el componente mayor será su necesidad de integrar a otros. Pero no pasa nada, de esta forma te aseguras de que el módulo de la librería se importa y que sus componentes se exportan correctamente.

Al grano, vamos a la aplicación *shop* para comprobar que su componente `AppComponent` funciona y (de paso) que se renderiza también el componente `ab-ui-greetings` incrustando los saludos.

`apps\shop\src\app\app.component.spec.ts`

```
import { UiModule } from '@a-boss/ui';
import { async, TestBed } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

describe('GIVEN: an AppComponent declared in AppModule', () => {
  describe('WHEN: the AppModule is compiled', () => {
    beforeEach(async(() => {
      TestBed.configureTestingModule({
        imports: [RouterTestingModule, UiModule],
        declarations: [AppComponent]
      }).compileComponents();
    }));

    it('THEN: should create the component', () => {
      const fixture = TestBed.createComponent(AppComponent);
      const app = fixture.debugElement.componentInstance;
      expect(app).toBeTruthy();
    });

    it(`THEN: should have a property title with value 'shop'`, () => {
      const fixture = TestBed.createComponent(AppComponent);
      const app = fixture.debugElement.componentInstance;
      expect(app.title).toEqual('shop');
    });

    it(`THEN: should render 'Hello world' in a H1 tag`, () => {
      const fixture = TestBed.createComponent(AppComponent);
      fixture.detectChanges();
      const compiled = fixture.debugElement.nativeElement;
      expect(compiled.querySelector('h1').textContent).toContain('Hello world');
    });
  });
});
```

2.2.2 Services

```
GIVEN: a GreetingsService
  WHEN: the DataModule is compiled
    THEN: should be created
```

```
    THEN: should return an observable when call 'getGrettings()'
    THEN: should return 'Welcome to api!' when call 'getGrettings()'
```

La prueba de servicios es más sencilla que la de componentes pues no hay que tratar con la renderización del HTML; sólo funcionalidad en una clase TypeScript. Pero, siempre hay un pero, muchos de estos servicios tratarán con llamadas asíncronas. Afortunadamente está todo pensado y se resuelve con dos conceptos: la función `async()` y inyección de réplicas (*mocks*) de las dependencias.

`libs\shared\data\src\lib\greetings\greetings.service.spec.ts`

```
// importar réplicas para testing de las dependencias del servicio
import {
  HttpClientTestingModule,
  HttpTestingController
} from '@angular/common/http/testing';
import { async, TestBed } from '@angular/core/testing';
import { Observable } from 'rxjs';
import { GreetingsService } from './greetings.service';

describe('GIVEN: a GreetingsService', () => {
  describe('WHEN: the DataModule is compiled', () => {
    beforeEach(() => {
      TestBed.configureTestingModule({
        imports: [HttpClientTestingModule]
      });
    });

    it('THEN: should be created', () => {
      const service: GreetingsService = TestBed.get(GreetingsService);
      expect(service).toBeTruthy();
    });

    it(`THEN: should return an observable when call 'getGrettings()'`, () => {
      const service: GreetingsService = TestBed.get(GreetingsService);
      const greetings$: Observable<any> = service.getGrettings$();
      expect(greetings$).toBeInstanceOf(Observable);
    });

    // Ojo al async para ejectuar las llamadas asíncronas
    it(`THEN: should return 'Welcome to api!' when call 'getGrettings()'`,
async(() => {
      const service: GreetingsService = TestBed.get(GreetingsService);
      service
        .getGrettings$()
        .subscribe(result =>
          expect(result).toEqual({ message: 'Welcome to api!' })
        );
      // mock del backend para no depender del servidor
      const httpMock = TestBed.get(HttpTestingController);
      // esperar a que se llame a esta ruta
      const req = httpMock.expectOne('http://localhost:3333/api');
```

```
        req.flush({ message: 'Welcome to api!' }); // responder con esto
        httpMock.verify(); // comprobar que no hay más llmadas
      }));
    });
  });
```

A partir de aquí es siempre igual. Defines un respuesta esperada, le das una entrada conocida y si algo no cuadra, entonces el código no pasa la prueba y tienes una oportunidad para mejorarlo.

En la prueba de servicios es fundamental que uses réplicas de sus dependencias. Es la forma de garantizar que pruebas únicamente el servicio, sin depender de nada. Esto era mucho más complejo con los componentes, pero un mandamiento con los servicios.

Resumen

En definitiva, los grandes desarrollos demandados por bancos, multinacionales o administración pública requieren soluciones fiables y mantenibles. Y eso pasa inexcusablemente por hacer testing. **Angular** facilita las pruebas unitarias y de integración; especialmente con las herramientas *Jest* y *Cypress* ya configuradas por **NxDevTools**.

Con este tutorial de formación avanzada en Angular te preparas para poder afrontar retos de tamaño industrial. Continúa aprendiendo a mejorar el rendimiento usando la detección del cambio en Angular.

> Aprender, programar, disfrutar, repetir. -- *Saludos, Alberto Basalo*