# Pruebas unitarias en Angular2

## Prueba de un componente

En primer lugar necesitamos importar todas las herramientas que usaremos:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By }              from '@angular/platform-browser';
import { DebugElement }    from '@angular/core';
```

Y después el componente a ser testeado:

```
import { StnSideMenuComponent } from "./components/sample.component";
```

El TestBed es la primera y más importante de las utilidades de prueba de Angular. Crea un módulo de prueba Angular - una clase @NgModule - que configura con el método configureTestingModule para producir el entorno del módulo para la clase que deseas probar.

Llama a configureTestingModule dentro de un BeforeEach para que, antes de ejecutar cada spec, TestBed pueda restablecerse a un estado base. El estado base incluye una configuración predeterminada del módulo de prueba que consiste en los declarables (componentes, directivas y tuberías) y los providers (algunos de ellos mockeados) que casi todo el mundo necesita.

Vamos a proceder con la prueba:

```
describe('component test suite', () => {

  let comp:    MySampleComponent;
  let fixture: ComponentFixture<MySampleComponent>;
  let de:      DebugElement;
  let el:      HTMLElement;

  // Proporciona nuestras implementaciones o datos 'mockeados' para el inyector de dependencia
  beforeEach(() => {

    TestBed.configureTestingModule({
      declarations: [StnSideMenuComponent], // declara el test component
    });

    fixture = TestBed.createComponent(StnSideMenuComponent);
    comp = fixture.componentInstance;

    // Consulta por título <h1> mediante el selector de elementos CSS
    de = fixture.debugElement.query(By.css('h1'));
    // Or queryAll
    el = de.nativeElement;

  });

});
```

El método createComponent devuelve una ComponentFixture, un identificador en el entorno de prueba que rodea al componente creado. El accesorio proporciona acceso a la instancia del componente en sí y al DebugElement que es un identificador en el elemento DOM del

componente. **También cierra la actual instancia de TestBed a una configuración adicional**.

> No vuelva a configurar el TestBed después de llamar a createComponent.

El método de consulta toma una función predicada y busca el árbol DOM entero del accesorio para el primer elemento que satisface el predicado. El resultado es un DebugElement diferente, uno asociado con el elemento DOM correspondiente.
La clase By es una clase para producir predicados.
Un predicado "A predicate" es una función que devuelve un booleano. Un predicado de consulta recibe un 'DebugElement` y devuelve 'true' si el elemento cumple los criterios de selección.
Por último, la instalación asigna el elemento DOM de la propiedad DebugElement nativeElement a él.
Las pruebas confirmarán que el contiene el texto del título esperado:

```
  it('should display original title', () => {
    fixture.detectChanges();
    expect(el.textContent).toContain(comp.title);
  });

  it('should display a different test title', () => {
    comp.title = 'Test Title';
    fixture.detectChanges();
    expect(el.textContent).toContain('Test Title');
  });
```

Estas pruebas solicitan el DebugElement para el elemento nativo HTML que satisfaga sus expectativas. Las pruebas completas finalmente se parecen a esto:

```
describe('component test suite', () => {

  let comp:    MySampleComponent;
  let fixture: ComponentFixture<MySampleComponent>;
  let de:      DebugElement;
  let el:      HTMLElement;

  // provide our implementations or mock-data to the dependency injector
  beforeEach(() => {

    TestBed.configureTestingModule({
      declarations: [StnSideMenuComponent], // declare the test component
    });

    fixture = TestBed.createComponent(StnSideMenuComponent);
    comp = fixture.componentInstance;

    // query for the title <h1> by CSS element selector
    de = fixture.debugElement.query(By.css('h1'));
    // Or queryAll
    el = de.nativeElement;

  });

  it('should display original title', () => {
     fixture.detectChanges();
     expect(el.textContent).toContain(comp.title);
  });

  it('should display a different test title', () => {
     comp.title = 'Test Title';
```

```
    fixture.detectChanges();
    expect(el.textContent).toContain('Test Title');
  });

});
```

> TestBed.createComponent() no activa la detección de cambios.

En el entorno de producción Angular detecta cambios automáticamente, en el entorno de pruebas tenemos que llamar explícitamente a `detectChanges()` para decir a Angular que algo ha cambiado por lo que debe activar el proceso de detección de cambios.

Hay algunos casos en los que un componente tiene una plantilla externa especificada por su parámetro `templateUrl`. Esos casos son un problema para la prueba porque el framework tiene que ir a buscar archivos externos que por naturaleza involucran métodos asíncronos. La prueba anterior no funcionará bajo estas circunstancias.

¿Entonces, qué podemos hacer? Bueno, existe la función async para ese propósito, vamos a proceder primero importándolo:

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

La configuración de la prueba para MySampleComponent debe dar tiempo al compilador de plantilla de Angular a leer los archivos. Empezando desde donde lo dejamos, la lógica sería dividir la configuración en dos llamadas beforeEach. El primero beforeEach maneja la compilación asíncrona, y el segundo procede como de costumbre:

```
// async beforeEach
beforeEach(async(() => {

  TestBed.configureTestingModule({
    declarations: [ MySampleComponent ], // declare the test component
  })
  .compileComponents();  // compile template and css

}));

// synchronous beforeEach
beforeEach(() => {
  fixture = TestBed.createComponent(MySampleComponent);

  comp = fixture.componentInstance; // MySampleComponent test instance

  // query for the title <h1> by CSS element selector
  de = fixture.debugElement.query(By.css('h1'));
  el = de.nativeElement;
});
```

Y SÍ, podríamos sólo usar async().then() en el primero pero por razones de legibilidad se recomiendab dividirlo en dos forEach. No te preocupes, el segundo se ejecutará sólo después de que la primera (la asíncrona) haya terminado.

Como dijimos antes el método TestBed.configureTestingModule devuelve la clase TestBed para poder encadenar las llamadas a otros métodos estáticos TestBed como el anterior compileComponents().

> El método compileComponents() compila asincrónicamente todos los componentes configurados en el módulo de prueba.

> Al llamar a compileComponents se cierra la instancia TestBed actual para una configuración adicional. Haz compileComponents el último paso antes de llamar `TestBed.createComponent para instanciar el componente-bajo-prueba.

Cuando compileComponents se completa, las plantillas externas y los archivos css han sido "inlined" y TestBed.createComponent puede crear nuevas instancias de MySampleComponent sincrónicamente.

## Prueba de un componente con una dependencia

Los componentes suelen tener dependencias de servicio. Supongamos que nuestro MySampleComponent depende de un UserService externo para recuperar información del usuario y el método vale la pena probar, por lo que en nuestro TestBed:

```
TestBed.configureTestingModule({
   declarations: [ MySampleComponent ],
// providers:    [ UserService ]  // NO! Don't provide the real service!
                                  // Provide a test-double instead
   providers:    [ {provide: UserService, useValue: userServiceStub } ]
});
```

Un componente bajo prueba no tiene que ser inyectado con servicios reales. De hecho, generalmente es mejor si son prueba dobles (stubs, fakes, spies, o mocks). El propósito de la especificación es probar el componente, no el servicio y los servicios reales pueden ser un problema.

Declarando a nuestro proveedor le decimos a TestBed que use nuestro código en lugar del UserService real dondequiera que se llame.

```
userServiceStub = {
  isLoggedIn: true,
  user: { name: 'Test User'}
};
```

Angular tiene un sistema de inyección jerárquico, por lo que tiene un inyector en todos los niveles. Si deseas obtener una referencia de lo que realmente está inyectado en el componente puedes conseguirlo del inyector del componente bajo prueba:

```
// UserService realmente inyectado en el componente
userService = fixture.debugElement.injector.get(UserService);
```

> El inyector componente es una propiedad de DebugElement del accesorio.

O directamente desde el inyector raíz del módulo:

```
// UserService desde el inyector raíz del módulo
userService = TestBed.get(UserService);
```

**Importante:** La instancia userService inyectada en el componente es un objeto completamente diferente al declarado en la prueba, en su lugar es un clon del userServiceStub proporcionado. Así que cambiar el código fuente una vez inyectado no afecta el comportamiento del servicio o estado en el componente.

```
 beforeEach(() => {

    // stub UserService for test purposes
```

```
    userServiceStub = {
      isLoggedIn: true,
      user: { name: 'Test User'}
    };

    TestBed.configureTestingModule({
       declarations: [ MySampleComponent ],
       providers:    [ {provide: UserService, useValue: userServiceStub } ]
    });

    fixture = TestBed.createComponent(MySampleComponent);
    comp    = fixture.componentInstance;

    // UserService from the root injector
    userService = TestBed.get(UserService);

    //  get the "welcome" element by CSS selector (e.g., by class name)
    de = fixture.debugElement.query(By.css('.welcome'));
    el = de.nativeElement;

});

it('should welcome the user', () => {

  fixture.detectChanges();
  const content = el.textContent;
  expect(content).toContain('Welcome', '"Welcome ..."');
  expect(content).toContain('Test User', 'expected name');

});

it('should welcome "Bubba"', () => {

  userService.user.name = 'Bubba'; // welcome message hasn't been shown yet
  fixture.detectChanges();
  expect(el.textContent).toContain('Bubba');

});
```

## La función de inyección

La función de inyección es una de las utilidades de prueba de Angular. Inyecta servicios en la función de prueba donde puedes alterarlos, espiarlos y manipularlos. Es el equivalente de la anterior TestBed.get (UserService) pero en el nivel de la función.

Utiliza el inyector TestBed actual y sólo puede devolver los servicios proporcionados en ese nivel. No devuelve servicios del componente providers.

La función de inyección tiene dos parámetros:

1. Una serie de fichas de inyección de dependencia de Angular.
2. Una función de prueba cuyos parámetros corresponden exactamente a cada elemento de la matriz de tokens de inyección.

```
    it('should evaluate a service', inject([Router], (router: Router) => {

      const spy = spyOn(router, 'navigateByUrl');

      heroClick(); // trigger click on first inner <div class="hero">

      // args passed to router.navigateByUrl()
      const navArgs = spy.calls.first().args[0];

      // expecting to navigate to id of the component's first hero
```

```
      const id = comp.heroes[0].id;
      expect(navArgs).toBe('/heroes/' + id,
        'should nav to HeroDetail for first hero');
    }));
```

## Prueba de un componente con un servicio asíncrono real

Cuando desees probar componentes que esperan respuestas asíncronas de un servicio, puedes utilizar un Spy:

```
beforeEach(() => {
    TestBed.configureTestingModule({
       declarations: [ TwainComponent ],
       providers:    [ TwainService ],
    });

    fixture = TestBed.createComponent(TwainComponent);
    comp    = fixture.componentInstance;

    // TwainService actually injected into the component
    twainService = fixture.debugElement.injector.get(TwainService);

    // Setup spy on the `getQuote` method
    spy = spyOn(twainService, 'getQuote')
          .and.returnValue(Promise.resolve(testQuote));

    // Get the Twain quote element by CSS selector (e.g., by class name)
    de = fixture.debugElement.query(By.css('.twain'));
    el = de.nativeElement;

  });

  it('should not show quote before OnInit', () => {
    expect(el.textContent).toBe('', 'nothing displayed');
    expect(spy.calls.any()).toBe(false, 'getQuote not yet called');
  });
```

El spy está diseñado de tal manera que cualquier llamada a getQuote recibe una promesa inmediatamente resuelta con una cita de la prueba. El spy pasa por alto el método real getQuote y por lo tanto no se pondrá en contacto con el servidor.

## The asynchronous it

Incluso cuando en el caso anterior el servicio responde con una promesa resuelta de inmediato, podemos simplemente comprobar si la función ha sido llamada pero no para el valor que se ha resuelto. Para ese escenario necesitamos esperar a que el valor esté disponible cuando el motor javascript ha hecho la vuelta completa.
Podemos usar la misma función 'async()' desde arriba en la implementación 'it':

```
it('should show quote after getQuote promise (async)', async(() => {
  fixture.detectChanges();

  fixture.whenStable().then(() => { // wait for async getQuote
    fixture.detectChanges();        // update view with quote
    expect(el.textContent).toBe(testQuote);
  });
}));
```

Esta prueba no tiene acceso directo a la promesa devuelta por la llamada a twainService.getQuote porque está enterrada dentro de TwainComponent.ngOnInit y, por lo tanto, inaccesible a una prueba que sólo sondea la superficie API del componente.

El método ComponentFixture.whenStable () devuelve su propia promesa que se resuelve cuando se completa la promesa getQuote. De hecho, la promesa whenStable se resuelve cuando todas las actividades asincrónicas pendientes dentro de esta prueba completan ... la definición de "estable".

> También existe el 'fakeAsync()' que es una variante de async pero proporciona más de una lectura de tipo lineal (no cubierto aquí).

Otra forma es utilizar el tradicional callback done() de la función 'it':

```
it('should show quote after getQuote promise (done)', done => {
  fixture.detectChanges();

  // get the spy promise and wait for it to resolve
  spy.calls.mostRecent().returnValue.then(() => {
    fixture.detectChanges(); // update view with quote
    expect(el.textContent).toBe(testQuote);
    done();
  });
});
```

Incluso cuando el callback done() generalmente se desaconseja podría cubrir casos en los que la implementación async o fakeAsync no es suficiente.

## Prueba de un componente de entrada y salida

## Pruebas de extremo a extremo (e2e) con Protractor

Pruebas de extremo a extremo es como probar desde el punto de vista del usuario. Mediante la simulación del comportamiento del usuario podemos probar que un flujo de aplicación desde el principio hasta el final se comporta como se esperaba.

El propósito de realizar pruebas de extremo a extremo es identificar las dependencias del sistema y asegurar que la integridad de los datos se mantenga entre varios componentes del sistema. Dado que el propósito no es probar componentes aislados del sistema, tomando el punto de vista del usuario podemos probar de forma integral las cosas que trabajan juntas como el acceso a la base de datos, la red, la comunicación con los otros sistemas y aplicaciones, etc.

Por lo tanto, en primer lugar necesitamos una herramienta capaz de comportarse como un usuario, pero desde una perspectiva programable, y que es Selenium Webdriver. Selenium nos permite automatizar el navegador.

Angular utiliza específicamente Protractor. Protractor se situa en la parte superior de Selenium, es un framework dedicado a pruebas end-to-end de aplicaciones AngularJS. Permite realizar pruebas en un amplio rango de navegadores. Dado que está hecho a medida para AngularJS, puede insertarse en el ciclo de vida de la aplicación. Y así saber cuando una operación asincrónica particular ha terminado, liberándote así de la necesidad de forzar explícitamente al navegador a dormir o a esperar a un acontecimiento dado.

Ejemplo de prueba:

```
describe("sample test suite", () => {
  it("should display proper welcome message", () => {
    browser.get("http://my-site.com")

    const welcomeMessageLocator = by.id("welcome-message");
    const welcomeMessageElementFinder = element(welcomeMessageLocator);

    expect(welcomeMessageElementFinder.getText()).toBe("Welcome Angular 2 Developer!");
  });
});
``
```