

Angular: Pruebas Unitarias con Jasmine y Karma – Udemy 06-2023

En este curso se aprenderá a comprender como funcionan los test unitarios, diferenciar y comprender Jasmine y Karma, test de integración, test unitarios de buena calidad, que tipo de test aplicar en cada momento, subir cobertura de un proyecto y buenas prácticas.

Sección 2: Conceptos Generales

Test Unitario: Es el encargado de comprobar que un fragmento de código funciona correctamente. El conjunto de estas pruebas nos ayuda a garantizar el funcionamiento de la aplicación.

Las ventajas del mismo son:

- Asegurar que el código nuevo que introduzcamos no haga que deje de funcionar el resto.
- Asegurarnos que el software a subir a producción funcione correctamente.
- Son útiles si utilizamos integración continua en nuestro proyecto.
- Permite probar distintas partes del proyecto sin que otras estén completas aún.
- Nos permite darnos cuenta si estamos haciendo un código muy complejo, ya que este será más difícil de probar.
- **Unit Test:** Testean métodos dentro de las clases. Si un método necesita acceso a otra clase, se van a simular (mockear).

Integration Test: En estos test se hacen usos de otras clases. Pero si necesitan acceso a otra clase no se van a simular.

E2E (End to End): Son las pruebas que simulan al usuario al momento de utilizar la aplicación. Hacer click en un botón, ir a un registro y demás.

Cobertura/coverage: Mide el grado en que el código ha sido probado. Puede ser general del proyecto o de cada método. No va a ser una cobertura del 100%, se recomienda que sea, en lo posible, del 80%. Más que la cantidad, es más importante la calidad de los mismos.

Jasmine: Es un marco de prueba JS. Lo utiliza por defecto Angular y nos permite hacer diferentes tipos de pruebas.

Karma: Es una herramienta que nos permite ejecutar en un navegador pruebas de Jasmine desde la consola de comandos. Es la forma física de ejecutar los test.

Sección 4: Test Unitarios

Una vez descargado el proyecto para comenzar a trabajar, instalamos las dependencias.

En este ejemplo utiliza JSON server para simular un back. Existe un db.json que tiene nuestra base de datos desde donde consume la info (api simulada). Para ello lo instalamos:

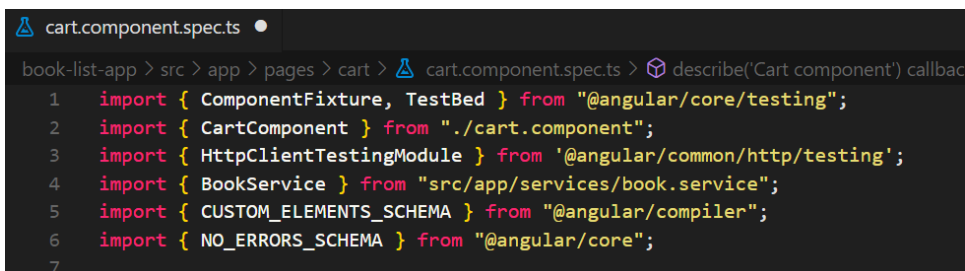
- Npm install -g json-server.
- Para inicializarlo: json-server --watch "nombre db.json".

Estructura de un fichero de pruebas (.spec)

En cada archivo .spec vamos a crear un **describe** que representa al componente a testear. Dentro del mismo veremos un **TestBed** que crea la estructura de configuración del unit test. Y finalmente veremos un **it**, donde cada it es un test unitario. Por lo general hay un describe que es un componente o servicio a probar y dentro del mismo todos los it que representan las pruebas de ese componente o servicio.

Configurar un TestBed

Al componente que deseo testear le creo un **archivo .spec.ts**. Dentro del mismo creo un describe, que el mismo está compuesto por el nombre del **describe** que quiera ponerle y una función call back que va a contener todo lo que voy a realizar.



```
1 import { ComponentFixture, TestBed } from "@angular/core/testing";
2 import { CartComponent } from "../cart.component";
3 import { HttpClientTestingModule } from '@angular/common/http/testing';
4 import { BookService } from "src/app/services/book.service";
5 import { CUSTOM_ELEMENTS_SCHEMA } from "@angular/compiler";
6 import { NO_ERRORS_SCHEMA } from "@angular/core";
7
```

```
describe('Cart component', () => {  
  //Se declara el componente a testear.  
  let component: CartComponent;  
  //Declaramos otra variable: Utilizaremos para traer, por ej, el servicio inyectado dentro del component, hacer que  
  //detecte cambios, etc.  
  let fixture: ComponentFixture<CartComponent>  
  
  //Luego creamos un fichero de configuración TestBed con un event (antes que ejecute un test, ejecuta el beforeEach):  
  beforeEach(() => {  
    TestBed.configureTestingModule({  
      //Siempre HttpClientTestingModule -> Cuando el componente hace una petición http (service), no la hace de  
      //manera real. Si utilizamos Angular Material en este componente debemos importar el modulo.  
      imports: [  
        HttpClientTestingModule  
      ],  
      //Componente que utilizamos en el test  
      declarations: [  
        CartComponent,  
      ],  
      //Servicios que utiliza el componente.  
      providers: [  
        BookService  
      ],  
      //Utilizarlas siempre.  
      schemas: [  
        CUSTOM_ELEMENTS_SCHEMA,  
        NO_ERRORS_SCHEMA  
      ],  
    }).compileComponents();  
  });  
});
```

```
//Luego creamos otro beforeEach para instanciar el componente inicializado arriba (component).  
beforeEach(() => {  
  fixture = TestBed.createComponent(CartComponent); //Lo extraemos del TestBed anterior de esta manera.  
  component = fixture.componentInstance; //Lo instanciamos.  
  fixture.detectChanges(); //El component entra en onInit haciendo lo que tiene que hacer en su metodo onInit.  
});
```

Todo esto dentro del describe.

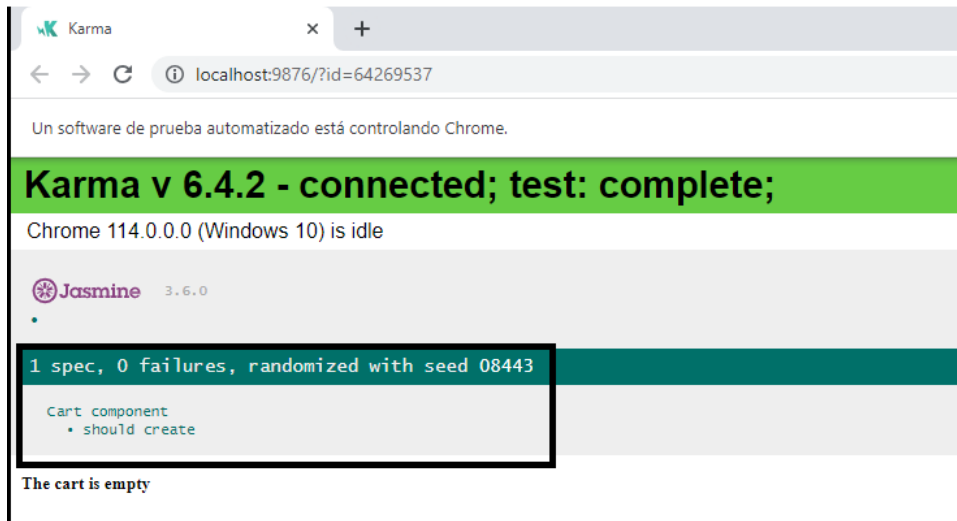
Crear el primer test

El primer test que haremos es el más sencillo, comprobar que el componente se ha creado.

El mismo se crea con `it` y lleva 2 parámetros, el nombre del test y una función call back con lo que deseamos testear.

```
it('should create', () => {  
  expect(component).toBeTruthy(); //Veremos si se ha instanciado correctamente.  
});
```

Para probarlo: **ng test**. Esto me va a abrir Karma en un browser mostrando el resultado de mis test:



Como vemos allí, está la cantidad de test que han pasado y el nombre del mismo y un punto verde arriba que representa que el test es correcto.

Test a métodos con return

Método del componente:

```
public getTotalPrice(listCartBook: Book[]): number {  
  let totalPrice = 0;  
  listCartBook.forEach((book: Book) => {  
    totalPrice += book.amount * book.price;  
  });  
  return totalPrice;  
}
```

Spect.ts de prueba del componente:

```
8
9  const listBook: Book[] = [
10    {
11      name: '',
12      author: '',
13      isbn: '',
14      price: 15,
15      amount: 2
16    },
17    {
18      name: '',
19      author: '',
20      isbn: '',
21      price: 20,
22      amount: 2
23    },
24    {
25      name: '',
26      author: '',
27      isbn: '',
28      price: 8,
29      amount: 2
30    },
31  ]
32
33  describe('Cart component', () => {
```

```
//Con it creamos cada uno de los unit test.
it('should create', () => {
  expect(component).toBeTruthy(); //Veremos si se ha instanciado correctamente.
});

//Testear un metodo con return. Ceo un array de libros fuera del describe para probar y para tenerlo para
otros test.
it('totalPrice', () => {
  const totalPrice = component.getTotalPrice(listBook);
  expect(totalPrice).toBeGreaterThan(0); //El totalPrice es mayor a cero.
  expect(totalPrice).toBe(86); //Va a ser 86;
  expect(totalPrice).not.toBeNull(); //No sera null.
});
```

Si quiero conocer el coverage, cierro en la terminal el ng test (ctrl + c) y ejecuto **ng test --code-coverage**. Una vez que se ejecutan todos los test, directamente arrastro el index de la nueva carpeta coverage en el navegador y allí veremos el % de cobertura de los test:

File	Statements	Branches	Functions	Lines
app/pages/cart	57.14%	12/21	0%	11/20
app/services	20%	7/35	14.28%	6/34
environments	100%	1/1	100%	1/1

Test a métodos sin return -> uso de spyon.

En estos casos, tenemos que observar que el método a testear llame a los métodos que se encuentran dentro del mismo o que debe ejecutar. Para ello se utiliza unos **espías** que vienen con Jasmine que verifica si esos métodos se han llamado.

Unos de los métodos llaman al service (updateAmountBook(book)), por lo tanto, debemos ver como llamarlo.

¡Importante! -> Un unit test nunca debe llamar otro método.

Por esto, junto al espía vemos cómo evitar esto, ya que no se debe hacer.

IR A LA ÚLTIMA PÁGINA

Pero esto no lo vamos a utilizar (última imagen) sino que crearemos un espía:

Para ello se declara una const donde creamos el **spyOn("servicio", "método a espiar")**. Con esto sabremos si el método del service es llamado o no. Pero antes debemos llamar al método principal quien es el que llama al método del service.

```
//Testear un método sin return. Aquí se probará que el método llame al los métodos: this._bookService.
updateAmountBook(book) o this.getTotalPrice(this.listCartBook).
it('onInputNumberChange Increments Correctly', () => {
  const action = 'plus';
  const book = listBook[0];

  const spy1 = spyOn(service, 'updateAmountBook');

  component.onInputNumberChange(action, book); //Llamamos al método que llama al método del service.

  expect(spy1).toHaveBeenCalled(); //Testeamos que el método sea llamado correctamente.
});
```

Este test me arrojará un error porque el método al que llamó no solo llama al método del service, sino también al método que calcula el total price, pero si lo comento en el componente veremos que funciona. Debemos crear un espía al otro método que está llamando.

Pero bien, anteriormente dijimos que un **test unitario no debería llamar al método de un servicio**, en caso de que lo necesite hay que simular la llamada. **Con el espía verificamos que la llamada se haga, pero en realidad además de comprobar que se llame correctamente, hay que hacer que anule dicha llamada.**

```
//Testear un método sin return. Aquí se probará que el método llame al los métodos: this._bookService.  
updateAmountBook(book) o this.getTotalPrice(this.listCartBook).  
it('onInputNumberChange Increments Correctly', () => {  
  const action = 'plus';  
  const book = listBook[0];  
  
  const spy1 = spyOn(service, 'updateAmountBook').and.callFake(() => null); //Como solo quiero ver que llame  
  al servicio pero no quiero que realmente lo llame hago una falsa llamada que devuelva algo.  
  
  component.onInputNumberChange(action, book); //Llamamos al método que llama al método del service.  
  expect(spy1).toHaveBeenCalled(); //Testeamos que el método sea llamado correctamente.  
});
```

Aquí lo que hago es crear una falsa llamada, para que me devuelva algo en lugar de literalmente ejecutar el método del service.

Ahora nos queda hacer lo mismo con el otro método que llame el método principal:

```
//Testear un método sin return. Aquí se probará que el método llame al los métodos: this._bookService.  
updateAmountBook(book) o this.getTotalPrice(this.listCartBook).  
it('onInputNumberChange Increments Correctly', () => {  
  const action = 'plus';  
  const book = listBook[0];  
  
  const spy1 = spyOn(service, 'updateAmountBook').and.callFake(() => null); //Como solo quiero ver que llame  
  al servicio pero no quiero que realmente lo llame hago una falsa llamada que devuelva algo.  
  
  const spy2 = spyOn(component, 'getTotalPrice').and.callFake(() => null);  
  
  component.onInputNumberChange(action, book); //Llamamos al método que llama al método del service.  
  expect(spy1).toHaveBeenCalled(); //Testeamos que el método sea llamado correctamente.  
  expect(spy2).toHaveBeenCalled();  
});
```


Si refrescamos el coverage, veremos que la línea se cubrió, pero faltaría la opción de reducir en lugar de sumar (action = 'plus');

```
it('onInputNumberChange Decrements Correctly', () => {  
  const action = 'minus';  
  const book = listBook[0];  
  
  const spy1 = spyOn(service, 'updateAmountBook').and.callFake(() => null); // Como solo quiero ver que llame al servicio pero no quiero que realmente lo llame hago una falsa llamada que devuelva algo.  
  
  const spy2 = spyOn(component, 'getTotalPrice').and.callFake(() => null);  
  
  component.onInputNumberChange(action, book); // Llamamos al método que llama al método del service.  
  expect(spy1).toHaveBeenCalled(); // Testeamos que el método sea llamado correctamente.  
  expect(spy2).toHaveBeenCalled();  
});
```

Hasta el momento, solo estamos testeando que realmente si este haciendo una llamada a los métodos internos, pero ¿sabemos si realmente está incrementando o disminuyendo?

```
it('onInputNumberChange Increments Correctly', () => {  
  const action = 'plus';  
  const book = listBook[0];  
  
  const spy1 = spyOn(service, 'updateAmountBook').and.callFake(() => null); // Como solo quiero ver que llame al servicio pero no quiero que realmente lo llame hago una falsa llamada que devuelva algo.  
  
  const spy2 = spyOn(component, 'getTotalPrice').and.callFake(() => null);  
  
  // Verificamos si realmente se ha incrementado  
  expect(book.amount).toBe(2);  
  component.onInputNumberChange(action, book); // Llamamos al método que llama al método del service.  
  expect(book.amount).toBe(3);  
  
  expect(spy1).toHaveBeenCalled(); // Testeamos que el método sea llamado correctamente.  
  expect(spy2).toHaveBeenCalled();  
});  
  
it('onInputNumberChange Decrements Correctly', () => {  
  const action = 'minus';  
  const book = listBook[0];  
  
  const spy1 = spyOn(service, 'updateAmountBook').and.callFake(() => null); // Como solo quiero ver que llame al servicio pero no quiero que realmente lo llame hago una falsa llamada que devuelva algo.  
  
  const spy2 = spyOn(component, 'getTotalPrice').and.callFake(() => null);  
  
  expect(book.amount).toBe(3);  
  component.onInputNumberChange(action, book); // Llamamos al método que llama al método del service.  
  expect(book.amount).toBe(2);  
});
```

¡Importante! Los test no se lanzan de manera consecutiva, por lo que, al verificar el incremento y decremento, hay veces que no funciona. Para ello a cada método le podemos asignar un listBook diferente o crear un book para cada uno de los test.

Test a métodos privados

Un método privado nunca es llamado directamente por el usuario, sino que llama a un método público y esta llama al privado. Por lo tanto, se testea el método público y a través de él se testea el método privado. Si desde un método público no podemos testear un método privado, es porque el código no está bien pensado ya que no está accediendo al método privado.

```
public onClearBooks(): void {
  if (this.listCartBook && this.listCartBook.length > 0) {
    this._clearListCartBook();
  } else {
    console.log("No books available");
  }
}

private _clearListCartBook() {
  this.listCartBook = [];
  this._bookService.removeBooksFromCart();
}
```

En este método privado podemos testear que realmente el carrito se vacíe dando como resultado un length de 0 y testeando la excepción si el carrito que deseo limpiar ya está vacío (**OJO, falta espiar el llamado al service**):

```
//Probando método privado a través del método público que lo llama.
it('onClearBooks works correctly', () => {
  component.listCartBook = listBook;
  component.onClearBooks();
  expect(component.listCartBook.length).toBe(0);
  expect(component.listCartBook.length === 0).toBeTrue();
});

//Probando método privado a través del método público que lo llama. Excepción.
it('onClearBooks works correctly', () => {
  component.listCartBook = [];
  component.onClearBooks();
  expect(component.listCartBook.length).toBe(0);
  expect(component.listCartBook.length === 0).toBeTrue();
});
```

Pero lo que podemos testear también es que se esté llamando de manera correcta al método privado a través de un espía de este y del service:

```
//Probando método privado a través del método público que lo llama.
it('onClearBooks works correctly', () => {

  //Para espiar un método privado hay que castear con any al método privado. En este caso, para que no me falle el
  //test, hay que llamarlo y a su vez espiarlo, sino no se ejecuta y no limpia.
  const spy1 = spyOn((component as any), '_clearListCartBook').and.callThrough();
  //Espia para el método del service
  const spy2 = spyOn(service, 'removeBooksFromCart').and.callFake(() => null);
  component.listCartBook = listBook;
  component.onClearBooks();

  expect(component.listCartBook.length).toBe(0);
  expect(component.listCartBook.length === 0).toBeTrue();
  expect(spy1).toHaveBeenCalled(); //Vemos si es llamado correctamente.
  expect(spy2).toHaveBeenCalled();
});
```

OnInit -> Observar si llamo al service y espiar

En caso de que el ngOnInit este llamando algún método del service, por regla general de los unit test debemos espiarlo, ya que no debemos estar llamando a otros métodos desde el unit test:

```
    }).compileComponents();
  });

  //Luego creamos otro beforeEach para instanciar el componente inicializado arriba (component).
  beforeEach(() => {
    fixture = TestBed.createComponent(CartComponent); //Lo extraemos del TestBed anterior de esta manera.
    service = fixture.debugElement.injector.get(BookService); //Instancio el service a utilizar.
    component = fixture.componentInstance; //Lo instanciamos.
    fixture.detectChanges(); //El component entra en ngOnInit haciendo lo que tiene que hacer en su metodo ngOnInit.
    //Espio el llamado del ngOnInit al service dandole como rta al falso llamado el listado de libros que busca.
    spyOn(service, 'getBooksFromCart').and.callFake(() => listBook);
  });

  //Con it creamos cada uno de los unit test.
  it('should create', () => {
```

Test a suscripciones --> Subscribe – Observable

En el método que tenemos estamos haciendo una suscripción a un método de un service que me devuelve un array de libros.

```
public getBooks(): void {  
  this.bookService.getBooks().pipe(take(1)).subscribe((resp: Book[]) => {  
    this.listBook = resp;  
  });  
}
```

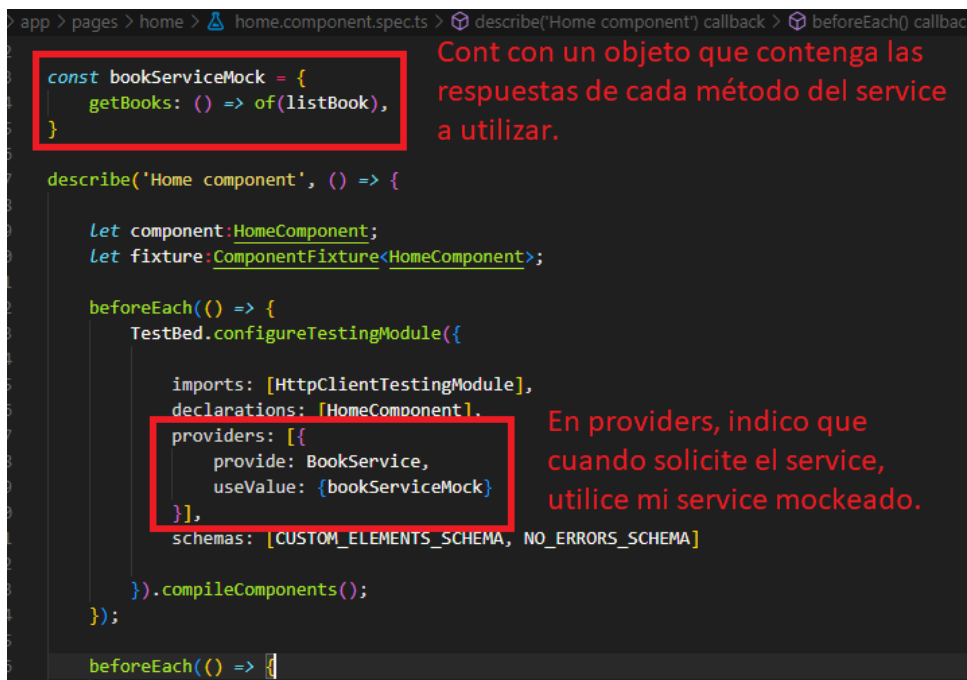
En dicho caso debemos espiar al service haciendo que nos devuelva un valor que queremos, pero el problema radica en que el `getBooks()` no devuelve un array, sino que **devuelve el resultado de una suscripción, es un observable**.

```
it('getBook get books from the subscription', () => {  
  //Como utiliza el servicio tenemos que traerlo:  
  const bookService = fixture.debugElement.injector.get(BookService);  
  //const listBook:Book[] = [];  
  //Luego creamos un espia de dicho service devolviendo un observable.  
  const spy1 = spyOn(bookService, 'getBooks').and.returnValue( of(listBook) );  
  
  component.getBooks();  
  
  expect(spy1).toHaveBeenCalled(); //Ahora con esto ya esta entrando a la suscripción.  
  expect(component.listBook.length).toBe(3); //Como pasamos un array de books, el length debería ser 3.  
});
```

Mock de un servicio

Si tenemos un método que utiliza varios métodos de un servicio, en lugar de hacer `returnValues` con cada uno de ellos tal cual hicimos en el punto anterior (Test a suscripciones) podemos **crear un servicio mock** que simule cada uno de ellos.

Para ello creamos un servicio con cada método que utilizamos del mismo y lo que yo quiero que me retorne:



¡Atención! UseValue, el bookServiceMock no va entre llaves.

Como estamos mockeando el service, ya no necesitamos utilizar el espía:

```
it('getBook get books from the subscription', () => {
  //Como utiliza el servicio tenemos que traerlo:
  fixture.debugElement.injector.get(BookService);

  component.getBooks();

  expect(component.listBook.length).toBe(3); //Como pasamos un array de books, el length debería ser 3.
});
```

Podemos verificar en el coverage que no se testeó el uso de `getBooks()` presente en el service.

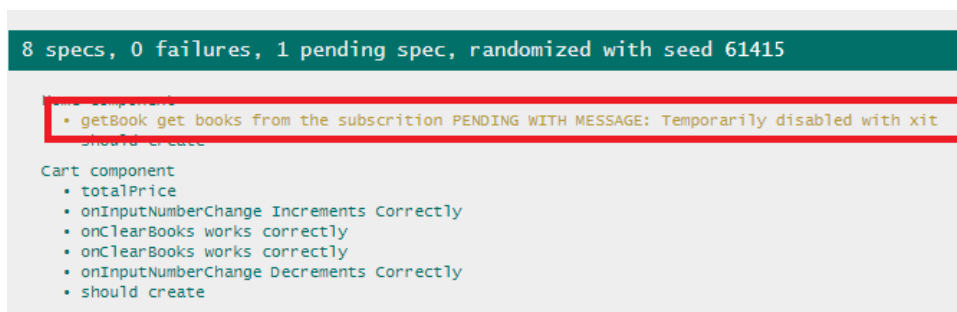
Xit, Fit, XDescribe, FDescribe

Cuando habla de **FIT** hace que solo se hagan los test que tenga una **F**. Al test que en el IT le agrego una F, se va a ejecutar solamente ese test.

Si a un describe le agrego una F, ejecutará todos los test de dicho describe y no los demás (**FDESCRIBE**).

Cuando a un test o a un describe le agrego una X, ejecutará todos los test menos los que tengan la X.

Esto es útil para que, en caso de que tengamos test con fallos por ej. por culpa de que el código está mal, lo podemos comentar, pero esto hará que podamos olvidarnos de dicho test, en cambio con la X lo vamos a ver que existe, pero como pendiente.



```
8 specs, 0 failures, 1 pending spec, randomized with seed 61415

* getBook get books from the subscription PENDING WITH MESSAGE: Temporarily disabled with xit
  should create

Cart component
  • totalPrice
  • onInputNumberChange Increments Correctly
  • onClearBooks works correctly
  • onClearBooks works correctly
  • onInputNumberChange Decrements Correctly
  • should create
```

BeforeEach, BeforeAll, AfterEach, AfterAll

El BeforeAll() es una función que se va a ejecutar al principio de todo y nunca más.

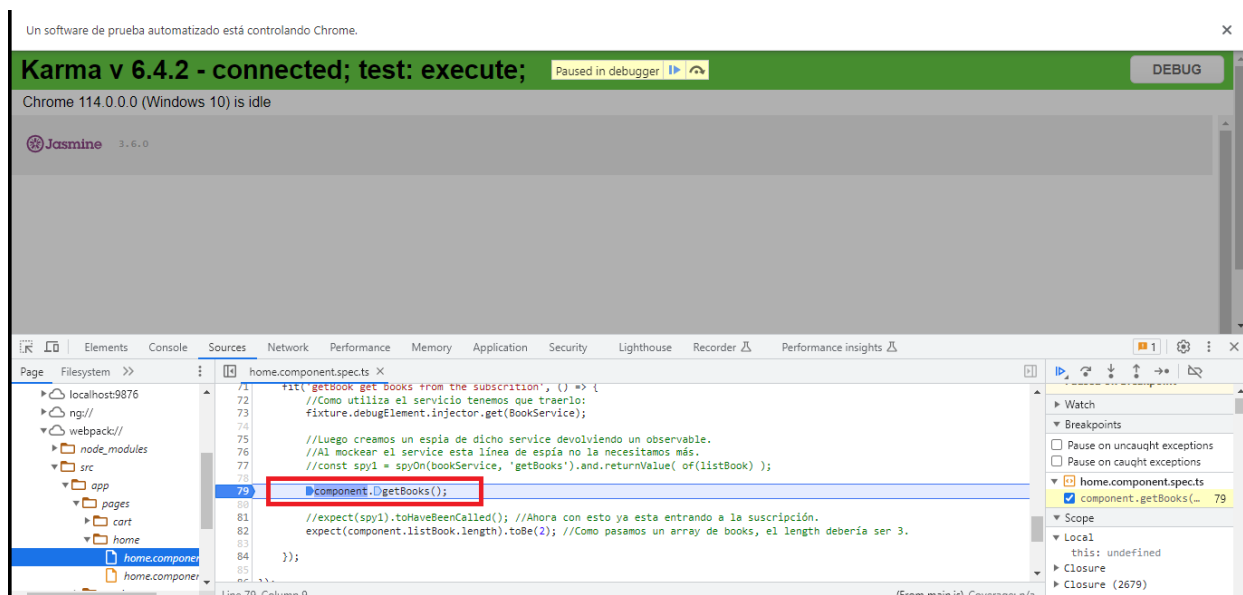
El BeforeEach() se ejecuta antes de cada test.

El AfterEach() es lo mismo, pero salta después de cada test (lo utilizaremos para probar el service).

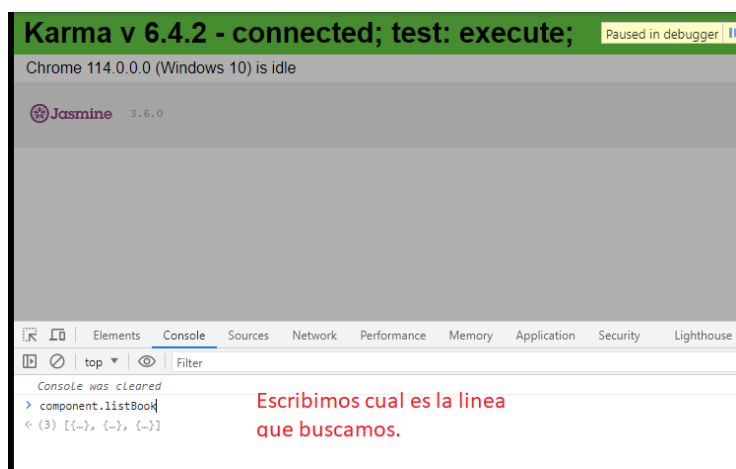
El AfterAll() saltara después de que terminen todos los test.

Depurar en Karma

Cuando tenemos algún error en un test y queremos depurarlo, podemos hacerlo en karma. Para ello vamos a inspeccionar en el navegador donde tenemos Karma abierto con el resultado de los test, vamos a source, context, webPack, y ya tenemos nuestra aplicación así que entramos a src, app, vamos al componente donde tenemos el spec y ponemos un punto de interrupción donde queremos saber que nos está trayendo el llamado:



Recargamos y nos vamos a la consola para ver cuál es el valor que nos está mostrando:



Y allí podremos darnos cuenta del error que tenemos.

Test a un pipe

Si nosotros agregamos un Pipe, por ej. Para cortar la descripción en 15 caracteres, el test va a fallar porque no reconoce al pipe personalizado que hemos creado. Para ello debemos mockearlo para que no arroje error en el test del home que queremos hacer:

```
<div class="card-body" >
  <p class="card-text alignCenter" style="margin-bottom: 2rem;" {{book.description | reduceText:15 }}</p>
  <!-- <p class="card-text alignCenter" style="margin-bottom: 2rem;" >{{book.description | reduceText:15}}</p>
  <div class="divPrice">
    <span>{{book.price}}€</span>
  </div>
  <div class="alignCenter divAddCart" (click)="bookService.addBookToCart(book)">
    <i class="fas fa-cart-plus addCart"></i>
  </div>
</div>
```

```
home.component.spec.ts  reduce-text.pipe.ts X  home.component.html
src > app > pages > reduce-text > reduce-text.pipe.ts > ...
1  import { Pipe, PipeTransform } from '@angular/core';
2
3  @Pipe({
4    name: 'reduceText'
5  })
6  export class ReduceTextPipe implements PipeTransform {
7
8    transform(value: string, ...args: number[]): string {
9      return value.substring(0, args[0]);
10     }
11
12   }
13
```

```
//Mockeo del pipe, como no me interesa testear el pipe, devuelvo un string vacío para que pase mi test del componente.
@Pipe({name: 'reduceText'})
class ReduceTextMock implements PipeTransform {
  transform():string {
    return '';
  }
}

describe('Home component', () => {

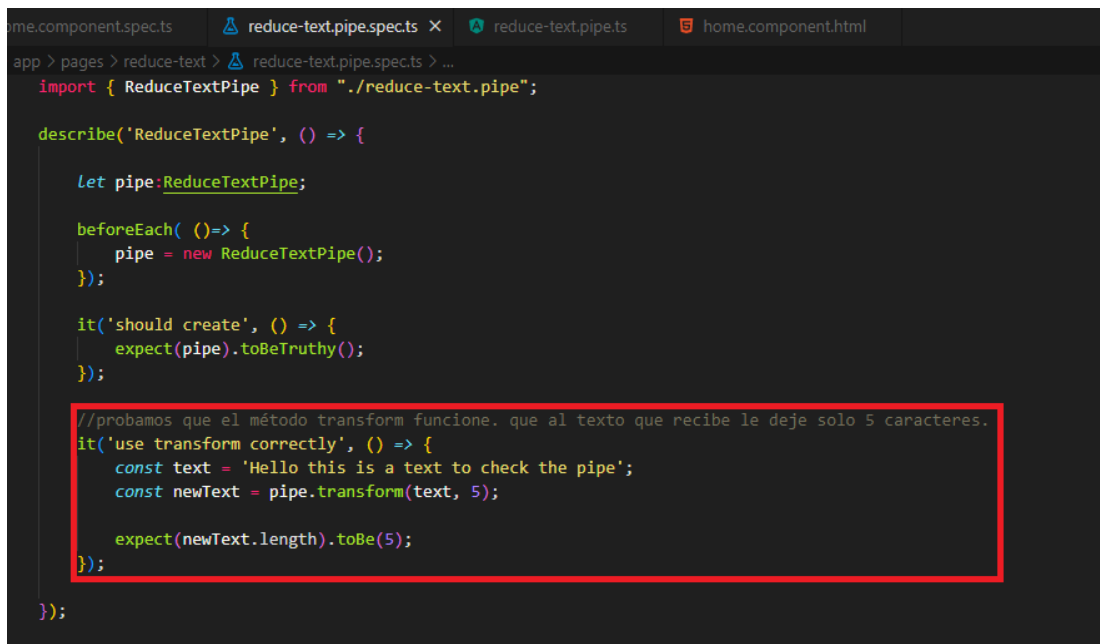
  let component: HomeComponent;
  let fixture: ComponentFixture<HomeComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      declarations: [HomeComponent, ReduceTextMock],
      providers: [

```


Como vemos, en el spec del componente donde implementamos el pipe, debemos mockearlo y agregarlo en los declarations.

Ahora, testaremos el Pipe en sí para testear que realmente el pipe se crea y el método transform funciona correctamente:



```
import { ReduceTextPipe } from "../reduce-text.pipe";

describe('ReduceTextPipe', () => {

  let pipe: ReduceTextPipe;

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

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

  //probamos que el método transform funcione. que al texto que recibe le deje solo 5 caracteres.
  it('use transform correctly', () => {
    const text = 'Hello this is a text to check the pipe';
    const newText = pipe.transform(text, 5);

    expect(newText.length).toBe(5);
  });

});
```

Test a un servicio (con peticiones a una API)

Las peticiones a una API siempre deben estar en el servicio, nunca en el componente.

Para poder **testear peticiones, debemos mockearlas** ya que no deben ser peticiones reales.

```

book.service.spec.ts
src > app > services > book.service.spec.ts > describe('BookService') callback
5
6 describe('BookService', () => {
7
8   let service: BookService;
9   let httpMock: HttpTestingController; //Usaremos para hacer peticiones mockeadas, no reales.
10
11   beforeEach(() => {
12     TestBed.configureTestingModule({
13       imports: [HttpClientTestingModule],
14       providers: [BookService],
15       schemas: [CUSTOM_ELEMENTS_SCHEMA, NO_ERRORS_SCHEMA]
16     });
17   });
18
19   beforeEach(() => {
20     service = TestBed.inject(BookService);
21     httpMock = TestBed.inject(HttpTestingController);
22   });
23
24   //Solo se utiliza en servicios que hacen peticiones a apis.
25   afterEach(() => {
26     httpMock.verify(); //Verifica que no haya peticiones pendientes entre cada test. Para no lanzar el siguiente test
27     //si hay peticiones pendientes.
28   });
29

```

Para testear el método del service:

```

public getBooks(): Observable<Book[]> {
  const url: string = environment.API_REST_URL + `/book`;
  return this._httpClient.get<Book[]>(url);
}

```

Vamos a mockear el service y tenemos que simular que la petición se hace y que la misma retorna un observable (en este caso de listBook). Al devolver un observable, nos suscribimos al método cuya respuesta debemos obtener lo que le enviamos: listBook.

```

it('getBooks return a list of books and does a get method', () => {
  service.getBooks().subscribe((resp: Book[]) => {
    expect(resp).toEqual(listBook);
  });

  //Definimos la ruta a la cual se hará la petición.
  const req = httpMock.expectOne(environment.API_REST_URL + `/book`);
  //Se espera que el pedido de dicha URL sea un GET.
  expect(req.request.method).toBe('GET');
  //Como estamos mockeando la respuesta de la api debemos indicarle lo que esperamos:
  req.flush(listBook);
});

```

Finalizando test book service – Parte 1

El resto de los métodos que tenemos en el service, juegan con el carrito y el **localStorage**. Para probar esto hay que ver la manera de **simular** un localStorage.

```
public getBooksFromCart(): Book[] {  
  let listBook: Book[] = JSON.parse(localStorage.getItem('listCartBook'));  
  if (listBook === null) {  
    listBook = [];  
  }  
  return listBook;  
}
```

En este método, se devuelve un array de books que los obtiene del localStorage. En este caso, como obtiene un JSON, le hace un parse para pasarlo a objeto de JS.

Para evitar el acceso al localStorage hay que **crear un espía** para no acceder de verdad al mismo, solo debemos simularlo.

```
describe('BookService', () => {  
  let service: BookService;  
  let httpMock: HttpTestingController; //Usaremos para hacer peticiones mockeadas, no reales.  
  let storage = {};  
  
  beforeEach(() => {  
    TestBed.configureTestingModule({  
      imports: [HttpClientTestingModule],  
      providers: [BookService],  
      schemas: [CUSTOM_ELEMENTS_SCHEMA, NO_ERRORS_SCHEMA]  
    });  
  });  
  
  beforeEach(() => {  
    service = TestBed.inject(BookService);  
    httpMock = TestBed.inject(HttpTestingController);  
  
    //creamos los espías del localStorage. Como al llamar al localStorage llama a una key('listCartBook') retornamos el storage  
    spyOn(localStorage, 'getItem').and.callFake((key:string)=>{  
      return storage[key] ? storage[key] : null;  
    });  
  });  
});
```

```
it('getBooksFromCart return empty when localStorage es empty', () => {  
  const listBooks = service.getBooksFromCart();  
  expect(listBooks.length).toBe(0);  
});
```

Finalizando test book service – Parte 2

```
public addBookToCart(book: Book) {  
  let listBook: Book[] = JSON.parse(localStorage.getItem('listCartBook'));  
  if (listBook === null) { // Create a list with the book  
    book.amount = 1;  
    listBook = [ book ];  
  } else {  
    const index = listBook.findIndex((item: Book) => {  
      return book.id === item.id;  
    });  
    if (index !== -1) { // Update the quantity in the existing book  
      listBook[index].amount++;  
    } else {  
      book.amount = 1;  
      listBook.push(book);  
    }  
  }  
  localStorage.setItem('listCartBook', JSON.stringify(listBook));  
  this._toastSuccess(book);  
}
```

Como vemos tenemos un setItem en el método, por lo que vamos a tener que crear un espía para evitar que este seteando el localStorage:

```
});  
  
beforeEach( () => {  
  service = TestBed.inject(BookService);  
  httpMock = TestBed.inject(HttpTestingController);  
  
  storage = {}; //Seteamos el storage con cada test, para evitar problemas con los demas test.  
  //creamos los espías del localStorage. Como al llamar al localStorage llama a una key('listCartBook') retornamos el storage  
  spyOn(localStorage, 'getItem').and.callFake((key:string)=>{  
    return storage[key] ? storage[key] : null;  
  });  
  
  spyOn(localStorage, 'setItem').and.callFake((key:string, value:string) => {  
    return storage[key] = value; //setea la key con el value.  
  });  
});  
  
//Solo se utiliza en servicios que hacen peticiones a apis.
```

Este espía simula el setItem, que lo que hace es tomar la key y setearle el value con la nueva lista de books.

Ahora, hecho esto debemos intentar agregar un book a listBook cuando llamamos el método correspondiente. Antes debemos crear otro espía para simular la llamada al _toastSuccess(book) y así testear correctamente el método privado.

```
it('addBookToCart add a book successfully when the list does not exist in the localStorage', () => {
  //Creamos la variable que utiliza el swal con su método fire para que no tenga error.
  const toast = {
    fire: () => null
  } as any;
  //creamos el espia retornando el resultado del metodo fire sobre el toast.
  const spy1 = spyOn(swal, 'mixin').and.callFake( () => {
    return toast;
  });

  let listBook = service.getBooksFromCart();
  expect(listBook.length).toBe(0);

  service.addBookToCart(book);

  listBook = service.getBooksFromCart();
  service.addBookToCart(book);
  expect(listBook.length).toBe(1);
  expect(spy1).toHaveBeenCalled();
});
```

Finalizando test book service – Parte 3

```
public removeBooksFromCart(): void {
  localStorage.setItem('listCartBook', null);
}
```

Lo que hace este método es eliminar la lista de libros del localStorage.

```
it('removeBooksFromCart removes the list from the localStorage', () => {
  //Agregamos un libro al carrito para tener uno que eliminar.
  service.addBookToCart(book);
  let listBook = service.getBooksFromCart();
  expect(listBook.length).toBe(1);

  service.removeBooksFromCart();
  listBook = service.getBooksFromCart();
  expect(listBook.length).toBe(0);
});
```

Alternativas para instanciar un componente o servicio

La manera en la que instanciamos un componente no es la única, también se puede hacer:

```

    },
    //Servicios que utiliza el componente.
    providers: [
      BookService,
      CartComponent
    ],
    //Utilizarlas siempre.
    schemas: [
      CUSTOM_ELEMENTS_SCHEMA,
      NO_ERRORS_SCHEMA
    ],
  }).compileComponents();
});

//Luego creamos otro beforeEach para instanciar el componente inicializado arriba (component).
beforeEach( () => { ...
});

//Con it creamos cada uno de los unit test...

fit('should create', inject([CartComponent], (testComponent: CartComponent) => {
  expect(testComponent).toBeTruthy();
}));

```

El componente va en providers

Lo ideal sería hacer la instancia de los componentes como hicimos anteriormente.

Test a rutas con routertestingmodule – parte 1

Para poder testear rutas, primero debemos reemplazar en nuestro HTML los [RouterLink] por eventos (click) que llamen a un método definido en el ts que, a través de router, permita navegar por mi home:

```

src > app > nav > nav.component.html > ...
Go to component
1 <nav class="navbar navbar-expand-lg navbar-dark bg-primary">
2   <a class="navbar-brand" (click)="navTo('home')" Book Shop</a>
3   <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarText" aria-controls="navbarText"
4     <span class="navbar-toggler-icon"></span>
5   </button>
6   <div class="collapse navbar-collapse" id="navbarText">
7     <ul class="navbar-nav mr-auto">
8       <li class="nav-item">
9         <a class="nav-link" (click)="navTo('home')" Home</a>
10      </li>
11    </ul>
12    <span class="navbar-text">
13      <i class="fas fa-shopping-cart fa-2x" (click)="navTo('cart')"></i>
14    </span>
15  </div>
16 </nav>

```

```
9   export class NavComponent implements OnInit {
10
11     constructor(
12       private router: Router
13     ) { }
14
15     ngOnInit(): void {
16     }
17
18     public navTo(path:string):void {
19       this.router.navigate(['/${path}']);
20     }
21
22   }
```

Creamos el archivo spec.ts de prueba y lo configuramos:

```
3   import { ComponentFixture, TestBed } from '@angular/core/testing';
4   import { RouterTestingModule } from '@angular/router/testing';
5
6   //Creamos una clase falsa para simular los componentes a los que cada path o ruta debe ir
7   class ComponentTestRoute {}
8
9   describe('Nav component', () => {
10
11     let component: NavComponent;
12     let fixture: ComponentFixture<NavComponent>;
13
14     beforeEach(() => {
15       TestBed.configureTestingModule({
16         //Importamos el modulo para testear rutas
17         imports: [
18           RouterTestingModule.withRoutes([ //Indicamos los path que vamos a testear y a que componente se dirige.
19             { path: 'home', component: ComponentTestRoute },
20             { path: 'cart', component: ComponentTestRoute }
21           ])
22         ],
23         declarations: [NavComponent],
24         schemas: [CUSTOM_ELEMENTS_SCHEMA, NO_ERRORS_SCHEMA]
25       }).compileComponents();
26     });
```

```
    beforeEach(() => {
      fixture = TestBed.createComponent(NavComponent);
      component = fixture.componentInstance;
      fixture.detectChanges();
    });

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

Test a rutas con routertestingmodule – parte 2

Para testear una ruta, primero debemos tener el router, que ya viene con RouterTestingModule y tenemos que crear un espía para ver si llamamos el navigate correctamente.

```
it('should navigate', () => {  
  const router = TestBed.inject(Router);  
  const spy1 = spyOn(router, 'navigate');  
  
  component.navTo('home');  
  //verificamos que el navigate va a /home.  
  expect(spy1).toHaveBeenCalledWith(['/home']);  
  
  component.navTo('cart');  
  //verificamos que el navigate va a /cart.  
  expect(spy1).toHaveBeenCalledWith(['/cart']);  
});
```

Sección 5: Angular Material - Dialog

Adaptando test a cart component

Cuando implementamos los módulos necesarios (MatDialogModule) y los providers necesarios (MatDialog) y vemos que nos arroja errores que son difíciles de interpretar, debemos mockear el MatDialog, de dicha forma al llamarlo y está vacío vamos a entender el error:

- Antes de mockear:


```
describe('Cart component', () => {  
  let component: CartComponent;  
  let fixture: ComponentFixture<CartComponent>;  
  let service: BookService;  
  
  beforeEach( () => {  
    TestBed.configureTestingModule({  
      imports: [  
        HttpClientTestingModule,  
        MatDialogModule  
      ],  
      declarations: [  
        CartComponent  
      ],  
      providers: [  
        BookService,  
        MatDialog  
      ],  
      schemas: [CUSTOM_ELEMENTS_SCHEMA, NO_ERRORS_SCHEMA]  
    }).compileComponents();  
  });  
});
```

Jasmine 3.6.0 Options

17 specs, 1 failure, randomized with seed 85566 finished in 0.07s

Spec List | Failures

Cart component > onClearBooks works correctly

Error: Unexpected synthetic listener @dialogContainer.start found. Please make sure that:

- Either 'BrowserAnimationsModule' or 'NoopAnimationsModule' are imported in your application.
- There is corresponding configuration for the animation named '@dialogContainer.start' defined in the 'animations' field of the '@Component' decorator (see <https://angular.io/api/core/Component#animations>).

Error: Unexpected synthetic listener @dialogContainer.start found. Please make sure that:

- Either 'BrowserAnimationsModule' or 'NoopAnimationsModule' are imported in your application.
- There is corresponding configuration for the animation named '@dialogContainer.start' defined in the 'animations' field of the '@Component' decorator (see <https://angular.io/api/core/Component#animations>).

at checkNoSyntheticProp (http://localhost:9876/_karma_webpack_/webpack:/node_modules/@angular/platform-browser/fesm2015/platform-browser.mjs:664:15)

at DefaultDomRenderer2.listen (http://localhost:9876/_karma_webpack_/webpack:/node_modules/@angular/platform-browser/fesm2015/platform-browser.mjs:654:24)

at listenerInternal (http://localhost:9876/_karma_webpack_/webpack:/node_modules/@angular/core/fesm2015/core.mjs:14933:1)

- Después de mockear MatDialog:

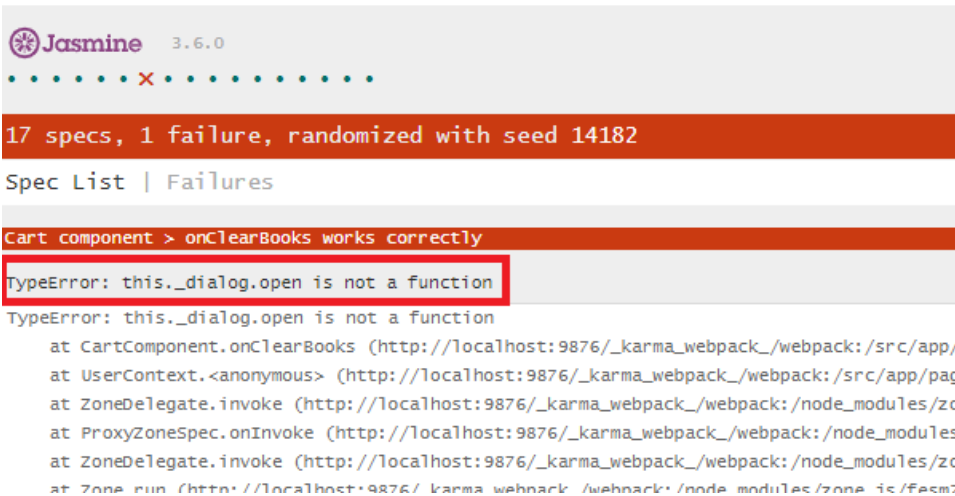
```
let fixture: ComponentFixture<CartComponent>;  
let service: BookService;  
  
beforeEach( () => {  
  TestBed.configureTestingModule({  
    imports: [  
      HttpClientTestingModule  
    ],  
    declarations: [  
      CartComponent  
    ],  
    providers: [  
      BookService,  
      {  
        provide: MatDialog,  
        useValue: {}  
      }  
    ],  
    schemas: [CUSTOM_ELEMENTS_SCHEMA, NO_ERRORS_SCHEMA]  
  }).compileComponents();  
});
```

Quitamos el import del module de matDialog

Cuando llame a MatDialog, utiliza un valor vacío.

Karma v 6.4.2 - connected; test: complete;

Chrome 114.0.0.0 (Windows 10) is idle



Jasmine 3.6.0

17 specs, 1 failure, randomized with seed 14182

Spec List | Failures

Cart component > onClearBooks works correctly

TypeError: this._dialog.open is not a function

TypeError: this._dialog.open is not a function

at CartComponent.onClearBooks (http://localhost:9876/_karma_webpack_/webpack:/src/app/...
at UserContext.<anonymous> (http://localhost:9876/_karma_webpack_/webpack:/src/app/pag...
at ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zon...
at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zon...
at ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zon...
at Zone.run (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:100:30)

Nos está faltando el método Open por lo que en el mock vamos a tener que agregarlo. Si observamos, el open se almacena en un dialogRef que a su vez está utilizando un afterClosed() que lo que hace es suscribirse a un observable:

```
public onClearBooks(): void {  
  if (this.listCartBook?.length > 0) {  
    const dialogRef = this.dialog.open(ConfirmDialogComponent, {  
      maxWidth: '400px',  
      data: {  
        title: '¿Estás seguro?',  
        message: 'Desea eliminar todos los productos del carrito?',  
      }  
    });  
  
    dialogRef.afterClosed().subscribe(dialogResult: boolean) => {  
      if (dialogResult) {  
        this._clearListCartBook();  
      }  
    });  
  } else {  
    console.log("No books available");  
  }  
}
```

El subscribe lo que hace es devolver un booleano por lo que el mock del MatDialog será:

```
const matDialogMock = {  
  open() {  
    return {  
      afterClosed: () => of(true)  
    }  
  },  
}
```

Aquí tenemos el método `open`, que a su vez retorna un método `afterClosed` que dicho método se suscribe a un observable que retorna un booleano. **Es un mockeo a un service.**

```
beforeEach( () => {  
  TestBed.configureTestingModule({  
    imports: [  
      HttpClientTestingModule  
    ],  
    declarations: [  
      CartComponent  
    ],  
    providers: [  
      BookService,  
      {  
        provide: MatDialog,  
        useValue: matDialogMock  
      }  
    ],  
    schemas: [CUSTOM_ELEMENTS_SCHEMA, NO_ERRORS_SCHEMA]  
  }).compileComponents();  
});
```

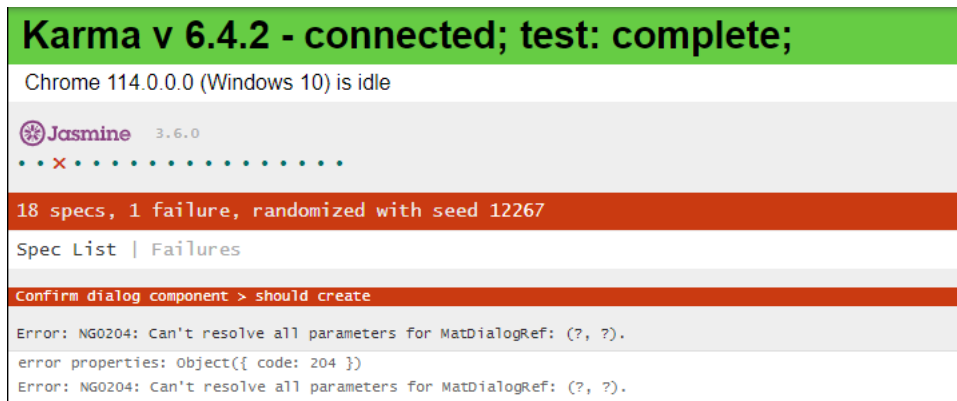
Y con esto ya solucionamos el inconveniente que nos arrojaba el `MatDialog`.

Test a confirmation dialog component

```
export class ConfirmDialogComponent {  
  
  public title: string;  
  public message: string;  
  
  constructor(  
    public dialogRef: MatDialogRef<ConfirmDialogComponent>,  
    @Inject(MAT_DIALOG_DATA) public data: DialogData  
  ) {}  
  
  public onConfirm(): void {  
    this.dialogRef.close(true);  
  }  
  
  public onDismiss(): void {  
    this.dialogRef.close(false);  
  }  
  
}
```

Nos vamos a encontrar con un problema similar al punto anterior:

```
confirmation-dialog.component.ts  confirmation-dialog.component.spec.ts X  
src > app > dialogs > confirmation-dialog > confirmation-dialog.component.spec.ts > ...  
6  
7 describe('Confirm dialog component', () => {  
8  
9   let component: ConfirmDialogComponent;  
10  let fixture: ComponentFixture<ConfirmDialogComponent>;  
11  
12  beforeEach(() => {  
13    TestBed.configureTestingModule({  
14      imports: [],  
15      declarations: [ConfirmDialogComponent],  
16      providers: [MatDialogRef, MAT_DIALOG_DATA],  
17      schemas: [NO_ERRORS_SCHEMA, CUSTOM_ELEMENTS_SCHEMA]  
18    }).compileComponents();  
19  });  
20  
21  beforeEach(() => {  
22    fixture = TestBed.createComponent(ConfirmDialogComponent);  
23    component = fixture.componentInstance;  
24    fixture.detectChanges();  
25  });  
26  
27  it('should create', () => {  
28    expect(component).toBeTruthy();  
29  });  
30  
31 });
```



Para solucionarlo creamos un mock del MAT_DIALOG_DATA y MatDialogRef:

```

7  const matDialogRefMock = {
8    close: () => null
9  }; //No me interesa que devuelva nada, solo ver que este llamando a la libreria.
10
11  describe('Confirm dialog component', () => {
12
13    let component: ConfirmDialogComponent;
14    let fixture: ComponentFixture<ConfirmDialogComponent>;
15
16    beforeEach(() => {
17      TestBed.configureTestingModule({
18        imports: [],
19        declarations: [ConfirmDialogComponent],
20        providers: [
21          //MatDialogRef,
22          //MAT_DIALOG_DATA,
23          {
24            provide: MAT_DIALOG_DATA,
25            useValue: {} //Como no nos interesa recibir una data (info que muestra en dialog), lo dejamos vacío.
26          },
27          {
28            provide: MatDialogRef,
29            useValue: matDialogRefMock
30          }
31        ],

```

Con esto se soluciona el test básico. **Cuando proveemos algo que da problemas**, lo mockeamos como solución más sencilla, en especial cuando no nos está brindando nada por lo que queremos testear.

Ahora vamos a testear los 2 métodos que utilizan el método close:

```

it('onConfirm send true value', () => {
  //Debemos crear un espia para llamar al servicio (MatDialogRef)
  const service = TestBed.inject(MatDialogRef); //Inyectamos el servicio que necesitamos.
  const spy = spyOn(service, 'close');

  component.onConfirm();
  expect(spy).toHaveBeenCalled(); //Verificamos la llamada al pasarle true.
});

it('onDismiss send false value', () => {
  //Debemos crear un espia para llamar al servicio (MatDialogRef)
  const service = TestBed.inject(MatDialogRef); //Inyectamos el servicio que necesitamos.
  const spy = spyOn(service, 'close');

  component.onDismiss();
  expect(spy).toHaveBeenCalled(); //Verificamos la llamada al pasarle true.
});

```

Sección 6: Ampliación

Introducción a los Test de Integración

En este caso, lo que queremos ver es que en nuestro HTML no esté disponible la etiqueta H5 cuando el listado de libros está lleno:

```

16 <div style="margin-top: 5px; margin-left: 1rem;">
17   x {{book.price}}€ = {{Math.round((book.amount * book.price) * 100) / 100}}€
18 </div>
19 </div>
20
21 <hr>
22
23 <h5 style="margin-bottom: 1rem;">Total price {{totalPrice}}€</h5>
24
25 <button type="button" class="btn btn-primary" style="margin-right: 10px;">Buy now</button>
26 <button (click)="onClearBooks()" type="button" class="btn btn-danger">Delete cart</button>
27
28 </ng-container>
29
30 <ng-container *ngIf='!listCartBook || listCartBook.length === 0'>
31   <h5 id="titleCartEmpty">The cart is empty</h5>
32 </ng-container>
33

```

```

//TEST INTEGRACIÓN
it('The title "The cart is empty" is not displayed when there is a cart', () => {
  component.listCartBook = listBook; //El listCartBook le ponemos la lista de libros creada.
  fixture.detectChanges(); //Detecta los cambios (lista actualizada)

  //Capturamos el elemento del html que queremos testear para ver si esta disponible o no. En este caso buscamos el
  //elemento que tiene id titleCartEmpty. Si queremos todos los h5 es queryAll.
  const debugElement: DebugElement = fixture.debugElement.query(By.css('#titleCartEmpty'));
  expect(debugElement).toBeFalsy(); //como la lista tiene elementos no debería existir.
});

```

Aquí se llenó la lista con el array de libros creado para los test, indicamos a angular que detecte los cambios (llene la lista) tomamos el elemento del HTML que queremos testear y le indicamos que esperamos que no exista, ya que la lista tiene elementos.

En el caso inverso:

```
it('The title "The cart is empty" is displayed when the list is empty', () => {
  component.listCartBook = [];
  fixture.detectChanges();

  const debugElement: DebugElement = fixture.debugElement.query(By.css('#titleCartEmpty'));
  expect(debugElement).toBeTruthy();
  if(debugElement){
    const element: HTMLElement = debugElement.nativeElement; //Extraemos el HTML texto que tenemos
    expect(element.innerHTML).toContain("The cart is empty");
  }
});
```

Automatizar test antes de generar build

Antes de generar el build, con un comando podemos hacer que pasen todos los test y, en caso de que pasen correctamente, ahí hace el build. En caso opuesto no.

Para eso, en el package.json creamos un comando propio indicando que queremos lanzar los test, una vez finalizados se cierran y lancen el build:

```
Depurar
"scripts": {
  "ng": "ng",
  "start": "ng serve",
  "build": "ng build",
  "test": "ng test",
  "test-coverage": "ng test --code-coverage",
  "tbuild": "ng test --no-watch && ng build",
  "lint": "ng lint",
  "e2e": "ng e2e",
  "mock": "json-server db.json"
},
```

Test a un formulario

Para testear un form se comienza como siempre, pero importando los módulos de formularios que estamos utilizando:

```
5
6  fdescribe('Form component', () => {
7
8    let component: FormComponent;
9    let fixture: ComponentFixture<FormComponent>;
10
11    beforeEach(() => {
12
13      TestBed.configureTestingModule({
14        imports: [
15          FormsModule,
16          ReactiveFormsModule
17        ],
18        declarations: [FormComponent],
19        schemas: [CUSTOM_ELEMENTS_SCHEMA, NO_ERRORS_SCHEMA],
20      }).compileComponents();
21    });
22
23    beforeEach(() => {
24
25      fixture = TestBed.createComponent(FormComponent);
26      component = fixture.componentInstance;
27      fixture.detectChanges();
28    });
29
30    it('should create', () => {
31      expect(component).toBeTruthy();
32    });
33  });
```

Algunos de los test que podemos implementar en el form:

- Name:


```
it('name field is required', () => {
  //obtenermos el campo nombre
  const nameField = component.form.get('name');
  //seteamos el valor al campo
  nameField.setValue('');
  //verificamos que el campo sea valido o no
  expect(nameField.invalid).toBeTrue();
});

it('name field has error with more than 5 chacarcters', () => {
  //obtenermos el campo nombre
  const nameField = component.form.get('name');
  //seteamos el valor al campo
  nameField.setValue('test name');
  //verificamos que el campo sea valido o no
  expect(nameField.valid).toBeFalse();
});

it('name field has error with less than 5 chacarcters', () => {
  //obtenermos el campo nombre
  const nameField = component.form.get('name');
  //seteamos el valor al campo
  nameField.setValue('Jack');
  //verificamos que el campo sea valido o no
  expect(nameField.valid).toBeTrue();
});
```

- Email:

```
it('email field is required', () => {
  //obtenermos el campo email
  const nameField = component.form.get('email');
  //seteamos el valor al campo
  nameField.setValue('');
  //verificamos que el campo sea valido o no
  expect(nameField.valid).toBeFalse();
});

it('email must be invalid', () => {
  //obtenermos el campo email
  const nameField = component.form.get('email');
  //seteamos el valor al campo
  nameField.setValue('test@');
  //verificamos que el campo sea valido o no
  expect(nameField.valid).toBeFalse();
});

it('email must be valid', () => {
  //obtenermos el campo email
  const nameField = component.form.get('email');
  //seteamos el valor al campo
  nameField.setValue('test@gmail.com');
  //verificamos que el campo sea valido o no
  expect(nameField.valid).toBeTrue();
});
```

- Formulario válido:

```
it('form is valid', () => {  
  const nameField = component.form.get('name');  
  const emailField = component.form.get('email');  
  
  nameField.setValue('Jack');  
  emailField.setValue('test@gmail.com');  
  
  expect(component.form.valid).toBeTrue();  
});
```

```
public onInputNumberChange(action: string, book: Book): void {
  const amount = action === 'plus' ? book.amount + 1 : book.amount - 1;
  book.amount = Number(amount);
  this.listCartBook = this._bookService.updateAmountBook(book);
  this.totalPrice = this.getTotalPrice(this.listCartBook);
}
```

```
},
]

describe('Cart component', () => {
  //Se declara el componente a testear.
  let component: CartComponent;
  //Declaramos otra variable: Utilizaremos para traer, por ej, el servicio inyectado dentro del component,
  //hacer que detecte cambios, etc.
  let fixture: ComponentFixture<CartComponent>;
  let service: BookService; //Variable global con el service.

  //Luego creamos un fichero de configuración TestBed con un event (antes que ejecute un test, ejecuta el
```

```
});
}).compileComponents();
});

//Luego creamos otro beforeEach para instanciar el componente inicializado arriba (component).
beforeEach(() => {
  fixture = TestBed.createComponent(CartComponent); //Lo extraemos del TestBed anterior de esta manera.
  service = fixture.debugElement.injector.get(BookService); //Instancio el service a utilizar.
  component = fixture.componentInstance; //Lo instanciamos.
  fixture.detectChanges(); //El component entra en onInit haciendo lo que tiene que hacer en su metodo
  onInit.
});

//Con it creamos cada uno de los unit test.
it('should create', () => {
```

```
//Testear un método sin return. Aquí se probará que el método llame al los métodos: this._bookService.
updateAmountBook(book) o this.getTotalPrice(this.listCartBook).
it('onInputNumberChange Increments Correctly', () => {
  const action = 'plus';
  const book = listBook[0];
  //Llamamos al servicio, el mismo es llamado por este método.
  const service = fixture.debugElement.injector.get(BookService); //Lo puedo declarar como global en el
  describe su lo utilizo en otros it.

  service |
  addBookToCa... (method) BookService.addBookToCart(book: ...
  getBooks
  getBooksFromCart
  removeBooksFromCart
  updateAmountBook
});
```

VOLVER A LA PÁGINA 7