Testeando Aplicaciones de Angular

Es importante tener en cuenta que Angular tiene una arquitectura basada en componentes. Cada componente es una unidad independiente y se puede probar individualmente. Esto permite realizar pruebas de forma aislada y garantizar la integridad de cada componente.

Contenido

1. ¿Cómo funcionan los tests en Angular? 1

3. Cómo testear un componente de Angular 3

3. 1. Primeros pasos 3

3. 2. Componentes con dependencias 5

3.3. Tests que involucran el DOM 6

Enlaces de interés 10

1. ¿Cómo funcionan los tests en Angular?

Angular utiliza el framework de testeo Jasmine y Karma para ejecutar los tests. Por defecto, las aplicaciones de Angular creadas mediante el CLI de Angular ya vienen listas para ejecutar tests. Esto se puede hacer con el comando **`ng test`**.

Los tests deben ir en archivos cuyo nombre termine en **`.spec.ts`** para que la herramienta de testeo los pueda reconocer como tales. Usualmente se tiene un archivo de test por cada componente que se desee testear, por ejemplo, para **`app.component.ts`** tenemos **`app.component.spec.ts`**.

Cuando corremos los tests, un navegador controlado por Karma es iniciado donde podremos ver los resultados. Más adelante veremos que también es posible ejecutar los tests sin tener que lanzar un navegador.

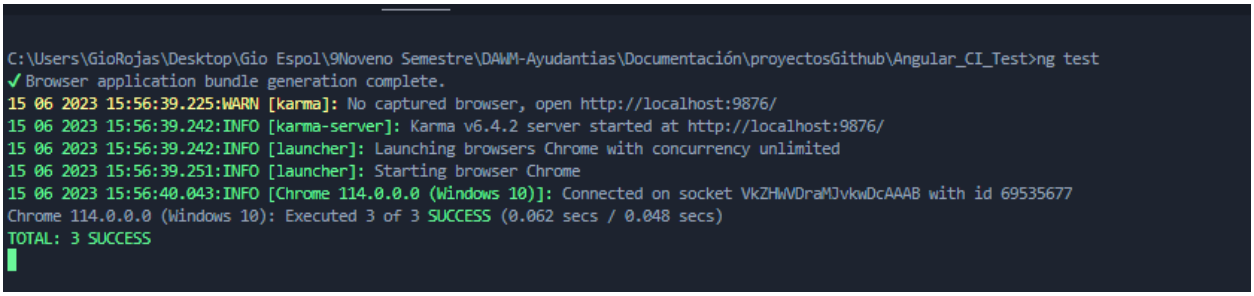
2. ng test

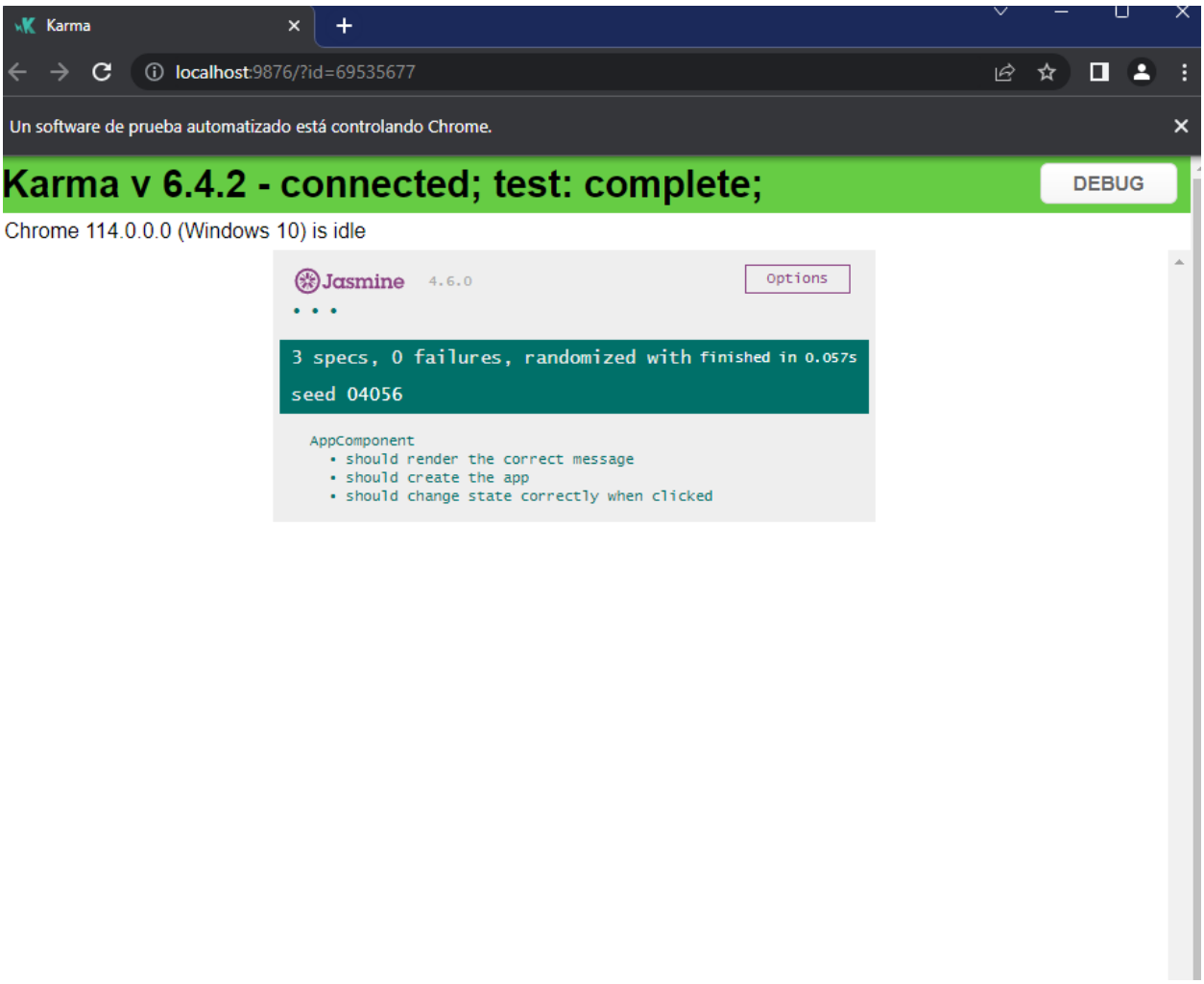
Cuando ejecutas ng test, Karma se encarga de:

- Compilar tu código fuente y los archivos de prueba
- Abrir un navegador (el predeterminado del usuario) en modo de ejecución de pruebas
- Ejecutar las pruebas definidas en los archivos **`.spec.ts`**

Elaborado por: Geovanny Rojas Lindao

- Este navegador se queda escuchando los archivos de tu proyecto y vuelve a ejecutar pruebas automáticamente cuando detecta modificaciones.
- Mostrar los resultados de las pruebas en la terminal del navegador.





Para terminar los testing se debe culminar el proceso de ng test, dentro de la terminal.

Se recomienda revisar la documentación

Elaborado por: Geovanny Rojas Lindao

3. Cómo testear un componente de Angular

3. 1. Primeros pasos

Suponiendo que tenemos el siguiente componente:

```
@Component({
  selector: 'lightswitch-comp',
  template: `
      <button type="button" (click)="clicked()">Click me!</button>
      <span>{{message}}</span>`
})
export class LightswitchComponent {
  isOn = false;
  clicked() { this.isOn = !this.isOn; }
  get message() { return `The light is ${this.isOn ? 'On' : 'Off'}`; }
}
```

Podemos testear su comportamiento verificando que al hacer clic retorna el valor esperado. A menudo es suficiente con testear los componentes de esta forma sin tener que involucrar al DOM. Por ejemplo, un test para este componente podría verse de la siguiente forma:

```
describe('LightswitchComp', () => {
  it('#clicked() should toggle #isOn', () => {
    const comp = new LightswitchComponent();
    expect(comp.isOn).withContext('off at first').toBe(false);
    comp.clicked();
    expect(comp.isOn).withContext('on after click').toBe(true);
   });
});
```

Elaborado por: Geovanny Rojas Lindao

Como podemos ver, en este test ejecutamos un método en el componente que se ejecutaría en respuesta a algún evento de la aplicación y comprobamos que el estado del componente haya cambiado de manera apropiada.

3. 2. Componentes con dependencias

Para testear componentes con dependencias podemos utilizar el TestBed de Angular. Cuando usamos el **TestBed**, lo que usualmente se hace es crear mocks de las dependencias que necesitamos para instanciar a nuestro componente, como en el siguiente ejemplo:

```
let comp: WelcomeComponent
let userService: UserService
beforeEach(() => {
  TestBed.configureTestingModule({
      providers: [
      WelcomeComponent,
      { provide: UserService, useClass: MockUserService }
      ]
  });
  comp = TestBed.inject(WelcomeComponent);
  userService = TestBed.inject(UserService);
});
```

Aquí podemos ver cómo configuramos el **TestBed** para poder inyectar el componente **WelcomeComponent**, que depende de **UserService**. Para ejecutar tests de está manera es importante estar familiarizados con el concepto de mocks y stubs. Básicamente un mock o un stub son clases que se crean de manera ad-hoc para simular el comportamiento del sujeto real. Esto es útil ya que nos permite testear los componentes con dependencias de manera aislada del resto del sistema.

```
it('should welcome logged in user after Angular calls ngOnInit', () => {
  comp.ngOnInit();
  expect(comp.welcome).toContain(userService.user.name);
});
```

NOTA: En los tests, se deben llamar explícitamente los métodos del ciclo de vida de los componentes, ya que Angular no hace esto por nosotros durante la ejecución de los tests.

3.3. Tests que involucran el DOM

Para testear que nuestros componentes se estén renderizando y respondiendo a eventos de manera apropiada, es necesario involucrar al DOM en nuestros tests. Podemos hacer uso de características adicionales de **`TestBed`** para lograr esto. Por ejemplo:

```
describe('BannerComponent (minimal)', () => {
  it('should create', () => {
      TestBed.configureTestingModule({declarations: [BannerComponent]});
      const fixture = TestBed.createComponent(BannerComponent);
      const component = fixture.componentInstance;
      expect(component).toBeDefined();
  });
});
```

En este ejemplo usamos el **`TestBed`** para crear una fixture (ComponentFixture) y poder acceder a nuestro componente a través de ella.

NOTA: Después de llamar a **`createComponent`** no se puede volver a invocar métodos de configuración en la **`TestBed`**. De hacerlo, una excepción será arrojada.
Podemos acceder al HTML del componente a través del objeto fixture:

```
it('should have <p> with "banner works!"', () => {
  const bannerElement: HTMLElement = fixture.nativeElement;
  const p = bannerElement.querySelector('p')!;
  expect(p.textContent).toEqual('banner works!');
});
```

En este caso utilizamos el método **`querySelector`** de la clase **`HTMLElement`** para acceder a la etiqueta de párrafo en la template de nuestro componente.

Segundo Ejemplo

Si queremos verificar que una tabla de nuestro componente se esté generando correctamente podemos hacer lo siguiente.

Supongamos que tienes un componente llamado TableComponent que genera una tabla a partir de una matriz de datos:

```html
<!-- table.component.html -->
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Email</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of data">
      <td>{{ item.name }}</td>
      <td>{{ item.email }}</td>
    </tr>
  </tbody>
</table>
```

Usamos la directiva *ngFor para recorrer la variable “data” dentro de nuestro componente y la variable de inicialización “item” para referencias el objeto actual.
Ahora para testear el correcto renderizado de la tabla en nuestro archivo .specs.ts escribimos lo siguiente.

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { TableComponent } from './table.component';
import { render } from '@testing-library/angular';

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

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [TableComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(TableComponent);
    component = fixture.componentInstance;
  });

  it('should render the table with the correct data', async () => {
    component.data = [
      { name: 'John Doe', email: 'john@example.com' },
      { name: 'Jane Smith', email: 'jane@example.com' },
    ];
    fixture.detectChanges();

    const { getByTestId, getAllByRole } = await render(TableComponent);
    const tableElement = getByTestId('table');
    const tableRows = getAllByRole('row');

    // Verificar la estructura de la tabla
    expect(tableElement).toBeTruthy();
    expect(tableRows.length).toBe(3); // Incluye la fila del encabezado

    // Verificar los datos de la tabla
    const tableCells = getAllByRole('cell');
    expect(tableCells[0].textContent).toContain('Name');
    expect(tableCells[1].textContent).toContain('Email');
    expect(tableCells[2].textContent).toContain('John Doe');
    expect(tableCells[3].textContent).toContain('john@example.com');
    expect(tableCells[4].textContent).toContain('Jane Smith');
    expect(tableCells[5].textContent).toContain('jane@example.com');
  });
```

Como se puede observar en component.data es un arreglo de objetos que puede ser el resultado de una petición a alguna API, tiene dos claves dado que en la tabla que estamos renderizando solo contiene esas dos claves.
El test nos dará “Success” si los datos renderizados dentro del HTML son los mismos que se esperan en la variable tableCells.

Elaborado por: Geovanny Rojas Lindao

Tercer Ejemplo

Para verificar que una etiqueta exista dentro de un componente, ya sea una etiqueta nativa de HTML o un “Selector” creado por nosotros se puede hacer lo siguiente:
Supongamos que tienes un componente llamado AppComponent que contiene una etiqueta <h1> en su plantilla HTML.

```
<!-- app.component.html -->
<h1 id="myHeading">Hello, World!</h1>
```

En el archivo **app.component.spec.ts** (o el archivo de prueba correspondiente a tu componente) y escribimos lo siguiente:

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

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

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

    fixture = TestBed.createComponent(AppComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should contain a specific HTML tag', () => {
    const element = fixture.nativeElement.querySelector('h1#myHeading');
    expect(element).toBeTruthy();
  });
});
```

En este ejemplo, primero configuramos el componente utilizando **TestBed.configureTestingModule** y creamos una instancia del componente utilizando **TestBed.createComponent**. Luego, utilizamos **fixture.detectChanges()** para actualizar la vista del componente.
Dentro del bloque **it**, utilizamos **fixture.nativeElement.querySelector** para buscar la etiqueta **<h1>** con el id **myHeading**. Luego, verificamos que el elemento exista utilizando **expect(element).toBeTruthy()**.

Elaborado por: Geovanny Rojas Lindao

Puede ser un poco complicado al inicio, pero Angular ya nos brinda plantillas de testing en su documentación incluid en los enlaces de interés para enfocarnos en el desarrollo de la web.

Enlaces de interés

- Documentación oficial sobre testeo en Angular: https://angular.io/guide/testing
- Cómo testear servicios de Angular: https://angular.io/guide/testing-services
- Cómo testear componentes de Angular: https://angular.io/guide/testing-components-basics
- Documentación: https://angular.io/guide/testing

Elaborado por: Geovanny Rojas Lindao