Angular

Testing de aplicaciones Angular

CertiDevs

Índice de contenidos

1. Testing de servicios

Crear y desarrollar tests de **servicios** en Angular es una parte fundamental del proceso de testing, ya que los servicios son responsables de la lógica de negocio y la comunicación con APIs externas.

A continuación, se presenta una guía detallada para crear y desarrollar tests de servicios en Angular utilizando Jasmine y Karma:

1.1. Crear un servicio

Si aún no se ha creado el servicio que se desea probar, se puede generar uno utilizando Angular CLI con el siguiente comando:

```
ng generate service my-service
```

Esto creará dos archivos: `my-service.service.ts` y `my-service.service.spec.ts`.

El archivo `.spec.ts` contendrá el esqueleto básico para las pruebas del servicio.

1.2. Importar dependencias y herramientas de testing

En el archivo de prueba `my-service.service.spec.ts`, es necesario importar las dependencias y herramientas de testing necesarias para el servicio que se está probando.

Algunas de las importaciones comunes incluyen `TestBed`, `inject` y el **propio servicio**. Por ejemplo:

```
import { TestBed } from '@angular/core/testing';
import { MyService } from './my-service.service';
```

1.3. Configurar el entorno de prueba:

Antes de escribir las pruebas, se debe configurar el entorno de prueba utilizando `TestBed`.

`TestBed` es un módulo de Angular que proporciona un entorno de pruebas aislado para cada prueba.

En el bloque `beforeEach`, se configura el entorno de prueba registrando el servicio que se está probando y las dependencias necesarias. Por ejemplo:

```
beforeEach(() => {
    TestBed.configureTestingModule({
        providers: [MyService]
    });
});
```

1.4. Escribir pruebas

Una vez configurado el entorno de prueba, se pueden **escribir pruebas** para el servicio utilizando la **sintaxis de Jasmine**.

Para cada prueba, se crea un **bloque** `it` que describe el comportamiento esperado del servicio. Por ejemplo:

```
it('should return the sum of two numbers', () => {
    const service: MyService = TestBed.get(MyService);
    const result = service.add(1, 2);
    expect(result).toEqual(3);
});
```

En este ejemplo, se prueba un método ficticio `add` que suma dos números.

Primero, se obtiene una instancia del servicio utilizando `TestBed.get()`.

Luego, se invoca el método `add` y se verifica que el resultado sea el esperado utilizando el método `expect().toEqual()` de Jasmine.

1.5. Probar métodos asíncronos

Los **servicios** en Angular suelen interactuar con **APIs externas** y realizar **operaciones asíncronas**.

Para probar **métodos asíncronos**, Jasmine ofrece funciones especiales como `async`, `fakeAsync` y `tick`.

Por ejemplo, para probar un método que devuelve una **promesa**:

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

it('should return data from a promise', async(() => {
    const service: MyService = TestBed.get(MyService);
    service.getData().then((data) => {
        expect(data).toEqual('some data');
    });
}));
```

En este ejemplo, se utiliza la función `async` para manejar la ejecución de la prueba asíncrona.

El método ficticio `getData()` devuelve una promesa que resuelve con un valor de datos. Se verifica que los datos devueltos sean iguales al valor esperado.

1.6. Espías y dobles de prueba (mocks):

En muchas situaciones, es necesario aislar el servicio que se está probando de sus **dependencias**

externas para garantizar que las pruebas sean precisas y rápidas.

Para lograr esto, se pueden utilizar **espías** (spies) y **dobles de prueba** (mocks) proporcionados por Jasmine.

Un **espía** es una función que registra llamadas, argumentos y resultados, mientras que un **doble de prueba (mock)** es un objeto que imita el comportamiento de una dependencia real.

Por ejemplo, si el servicio tiene una dependencia de un servicio de API, se puede crear un doble de prueba para el servicio de API y utilizar un espía para simular la llamada al método:

```
import { HttpClientTestingModule, HttpTestingController } from
'@angular/common/http/testing';

describe('MyService', () => {
let service: MyService;
let httpTestingController: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [MyService]
    });

    service = TestBed.inject(MyService);
    httpTestingController = TestBed.inject(HttpTestingController);
  });

  afterEach(() => {
    httpTestingController.verify();
  });

  it('should fetch data from the API', () => {
    const testData = { key: 'value' };

    service.getData().subscribe((data) => {
      expect(data).toEqual(testData);
    });

    const req = httpTestingController.expectOne('api-url');
    expect(req.request.method).toEqual('GET');
    req.flush(testData);
  });
});
```

En este ejemplo, se utiliza `HttpClientTestingModule` para reemplazar el servicio `HttpClient` real con una versión de prueba.

Luego, se crea una instancia del servicio `HttpTestingController`, que permite controlar y verificar las solicitudes HTTP realizadas por el servicio.

En la prueba, se suscribe al método ficticio `getData()` del servicio y se verifica que los datos devueltos sean iguales a los datos de prueba.

Después, se utiliza `httpTestingController.expectOne()` para esperar y capturar la solicitud HTTP realizada por el servicio.

Se verifica que la solicitud sea un GET y, finalmente, se simula la respuesta utilizando `req.flush(testData)`.

1.7. Ejecutar las pruebas

Para ejecutar las pruebas, se utiliza el comando `ng test` en la línea de comandos desde la raíz del proyecto.

Karma iniciará y ejecutará las pruebas utilizando la configuración en `karma.conf.js`.

Por defecto, las pruebas se ejecutan en el navegador Chrome y se recargan automáticamente cuando se detectan cambios en los archivos de prueba.

2. Testing de componentes

Crear y desarrollar **tests de componentes** en Angular es crucial para garantizar que la lógica y la estructura de la interfaz de usuario de una aplicación funcionen correctamente.

A continuación, se presenta una guía detallada para crear y desarrollar tests de componentes en Angular utilizando Jasmine y Karma:

2.1. Crear un componente

```
ng generate component my-component
```

Esto creará varios archivos, incluidos `my-component.component.ts` y `my-component.component.spec.ts`.

El archivo `.spec.ts` contendrá el esqueleto básico para las pruebas del componente.

2.2. Importar dependencias y herramientas de testing:

En el archivo de prueba `my-component.component.spec.ts`, es necesario importar las dependencias y herramientas de testing necesarias para el componente que se está probando.

Algunas de las importaciones comunes incluyen `TestBed`, `ComponentFixture`, y el propio componente.

Por ejemplo:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { MyComponentComponent } from './my-component.component';
```

2.3. Configurar el entorno de prueba:

Antes de escribir las pruebas, se debe configurar el entorno de prueba utilizando `TestBed`.

`TestBed` es un módulo de Angular que proporciona un entorno de pruebas aislado para cada prueba.

En el bloque `beforeEach`, se configura el entorno de prueba compilando y registrando el componente que se está probando y las dependencias necesarias. Por ejemplo:

```
let component: MyComponentComponent;
let fixture: ComponentFixture<MyComponentComponent>;

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

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

En este ejemplo, se crea una instancia de `ComponentFixture`, que proporciona acceso al componente, el elemento DOM asociado y la detección de cambios. También se obtiene una instancia del componente que se está probando.

2.4. Escribir pruebas

Una vez configurado el **entorno de prueba**, se pueden **escribir pruebas** para el componente utilizando la sintaxis de Jasmine.

Para cada prueba, se crea un bloque `it` que describe el comportamiento esperado del componente. Por ejemplo:

```
it('should display the title', () => {
    const compiled = fixture.nativeElement;
    expect(compiled.querySelector('h1').textContent).toContain('My Component');
});
```

En este ejemplo, se prueba que el componente muestre el título correcto.

Primero, se obtiene el **elemento DOM** asociado al componente a través de `fixture.nativeElement`.

Luego, se verifica que el contenido del elemento `h1` contenga el título esperado utilizando el método

`expect().toContain()` de Jasmine.

2.5. Probar la interacción del usuario

Además de probar la estructura y el contenido del componente, es importante probar cómo **interactúa el usuario** con el componente.

Para **simular** y **probar** la **interacción del usuario**, se pueden utilizar eventos y espías de Jasmine.

Por ejemplo, para probar un botón que emite un evento:

```
it('should emit an event when button is clicked', () => {
    spyOn(component.myEvent, 'emit');
    const button = fixture.nativeElement.querySelector('button');
    button.click();
    expect(component.myEvent.emit).toHaveBeenCalled();
});
```

En este ejemplo, se utiliza un **espía** para observar la función de emisión del evento `myEvent`.

Luego, se obtiene el **botón** del **elemento DOM** y se simula un **clic** en el botón utilizando `button.click()`.

Finalmente, se verifica que el evento `myEvent.emit` haya sido llamado utilizando el método `expect().toHaveBeenCalled()` de Jasmine.

2.6. Probar cambios en la vista y en el modelo

Es importante probar cómo se actualiza la vista del componente en respuesta a los **cambios en el modelo** y viceversa.

Para hacer esto, se pueden utilizar la detección de cambios de Angular y las funciones de Jasmine.

Por ejemplo, para probar la actualización de un campo de entrada en el componente:

```
it('should update the input field value', () => {
    const inputElement = fixture.nativeElement.querySelector('input');
    inputElement.value = 'New value';
    inputElement.dispatchEvent(new Event('input'));
    fixture.detectChanges();
    expect(component.inputValue).toEqual('New value');
});
```

En este ejemplo, se obtiene el elemento de entrada del DOM y se **actualiza su valor**.

Luego, se dispara un evento de entrada utilizando `inputElement.dispatchEvent()` para simular la **interacción del usuario**.

Después, se llama a `fixture.detectChanges()` para que Angular **actualice la vista y el modelo** en respuesta al cambio en el valor del campo de entrada.

Finalmente, se verifica que el valor del modelo `component.inputValue` se haya actualizado correctamente utilizando el método `expect().toEqual()` de Jasmine.

2.7. Probar la integración con otros componentes y servicios

Los **componentes** en Angular a menudo **interactúan** con otros **componentes** y **servicios**, y es crucial probar estas **interacciones**.

Para hacer esto, se pueden utilizar **dobles de prueba (mocks)** y **espías** de Jasmine.

Por ejemplo, si un **componente** depende de un **servicio**, se puede proporcionar un **doble de prueba** del **servicio** al configurar el entorno de prueba en TestBed:

```
TestBed.configureTestingModule({
    declarations: [ MyComponentComponent ],
    providers: [
        { provide: MyService, useClass: MockMyService }
    ]
})
.compileComponents();
```

En este ejemplo, se proporciona un doble de prueba `MockMyService` en lugar del servicio real `MyService`.

Luego, se pueden utilizar espías y otros métodos de Jasmine para probar cómo el componente interactúa con el servicio simulado.

2.8. Ejecutar las pruebas

Para **ejecutar las pruebas**, se utiliza el comando `ng test` en la línea de comandos desde la raíz del proyecto.

Karma iniciará y ejecutará las pruebas utilizando la configuración en `karma.conf.js`.

Por defecto, las pruebas se ejecutan en el navegador Chrome y se recargan automáticamente cuando se detectan cambios en los archivos de prueba.

3. Testing de componentes

Supongamos que tenemos otro **componente** llamado `GreetingComponent` que muestra un saludo personalizado en función de la propiedad `name` del componente.

Archivo `greeting.component.ts`:

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

@Component({
    selector: 'app-greeting',
    template: `
        <div class="greeting">
        <h1>Hello, {{ name }}!</h1>
        </div>
    `,
})
export class GreetingComponent {
    @Input() name: string = 'Guest';
}
```

Para probar la estructura y el contenido de este componente, se debe seguir estos pasos:

Importar las dependencias necesarias en el archivo de prueba `greeting.component.spec.ts`:

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

Configurar el entorno de prueba utilizando TestBed y ComponentFixture:

```
let component: GreetingComponent;
let fixture: ComponentFixture<GreetingComponent>;

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

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

Escribir pruebas para verificar la estructura y el contenido del componente.

En este caso, se verificará que el componente muestre el **saludo correcto** en función del valor de la propiedad `name`:

```
it('should display the default greeting when no name is provided', () => {
    const compiled = fixture.nativeElement;
    expect(compiled.querySelector('.greeting h1').textContent).toContain('Hello,
Guest!');
```

```
});

it('should display a personalized greeting when a name is provided', () => {
    component.name = 'John Doe';
    fixture.detectChanges();
    const compiled = fixture.nativeElement;
    expect(compiled.querySelector('.greeting h1').textContent).toContain('Hello, John
Doe!');
});
```

En este ejemplo, se verifica que el **componente** muestre el saludo por defecto "Hello, Guest!" cuando no se proporciona un valor para la propiedad `name`.

Además, se prueba que el componente muestre un **saludo personalizado** cuando se proporciona un **valor** para `name`.

Estas pruebas ayudan a garantizar que el contenido y la estructura del componente se muestren correctamente en función de los datos proporcionados.

4. Testing de directivas y pipes

A continuación, se presenta un ejemplo completo de cómo probar la integración de un **componente** con **directivas personalizadas** y **pipes**.

Supongamos que tenemos un componente llamado `UserInfoComponent` que muestra información del usuario utilizando una **directiva** personalizada `highlight` y un **pipe** personalizado `capitalize`.

Archivo `user-info.component.ts`:

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

@Component({
    selector: 'app-user-info',
    template: `
        <div class="user-info">
        <p appHighlight>Name: {{ name | capitalize }}</p>
        <p>Age: {{ age }}</p>
        </div>
    `,
})
export class UserInfoComponent {
    @Input() name: string = '';
    @Input() age: number = 0;
}
```

Directiva personalizada `highlight.directive.ts`:

```
import { Directive, ElementRef, Renderer2 } from '@angular/core';
```

```
@Directive({
    selector: '[appHighlight]',
})
export class HighlightDirective {
    constructor(private el: ElementRef, private renderer: Renderer2) {
        renderer.setStyle(el.nativeElement, 'backgroundColor', 'yellow');
    }
}
```

Pipe personalizado `capitalize.pipe.ts`:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
    name: 'capitalize',
})
export class CapitalizePipe implements PipeTransform {
    transform(value: string): string {
        return value.charAt(0).toUpperCase() + value.slice(1).toLowerCase();
    }
}
```

Para probar la integración de este **componente** con la **directiva** personalizada y el **pipe**, siga estos pasos:

4.1. Dependencias necesarias

Importar las dependencias necesarias en el archivo de prueba `user-info.component.spec.ts`:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { UserInfoComponent } from './user-info.component';
import { HighlightDirective } from '../directives/highlight.directive';
import { CapitalizePipe } from '../pipes/capitalize.pipe';
```

4.2. Configurar el entorno de prueba

Configurar el entorno de prueba utilizando `TestBed` y `ComponentFixture`:

```
let component: UserInfoComponent;
let fixture: ComponentFixture<UserInfoComponent>;

beforeEach(async () => {
await TestBed.configureTestingModule({
declarations: [UserInfoComponent, HighlightDirective, CapitalizePipe],
}).compileComponents();
```

```
});

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

4.3. Escribir prueba

Escribir pruebas para verificar la integración del componente con la **directiva** personalizada y el **pipe**.

En este caso, se verificará que la **directiva** `highlight` se aplique correctamente y que el **pipe** `capitalize` funcione como se espera:

```
it('should apply the highlight directive to the name paragraph', () => {
    component.name = 'john doe';
    component.age = 25;
    fixture.detectChanges();
    const compiled = fixture.nativeElement;
    const nameParagraph = compiled.querySelector('.user-info p:first-child');

    expect(nameParagraph.style.backgroundColor).toEqual('yellow');
});

it('should capitalize the name using the capitalize pipe', () => {

    component.name = 'john doe';
    component.age = 25;
    fixture.detectChanges();
    const compiled = fixture.nativeElement;
    const nameParagraph = compiled.querySelector('.user-info p:first-child');

    expect(nameParagraph.textContent).toContain('Name: John doe');
});
```

En este ejemplo, se verifica que la **directiva** `highlight` se aplique correctamente al párrafo que contiene el nombre del usuario, cambiando el color de fondo a amarillo.

Además, se prueba que el **pipe** `capitalize` capitalice correctamente la primera letra.

5. Testing enrutamiento

Para probar el **enrutamiento** y la **navegación** entre componentes, se puede utilizar el **enrutador** de Angular y las utilidades de prueba proporcionadas por el framework.

Supongamos que tenemos una aplicación con **dos componentes**, `HomeComponent` y `AboutComponent`, y

una configuración de **enrutamiento simple** para navegar entre ellos.

Archivo `app-routing.module.ts`:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';

const routes: Routes = [
    { path: '', component: HomeComponent },
    { path: 'about', component: AboutComponent },
];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule],
})
export class AppRoutingModule {}
```

Componente `home.component.ts`:

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

@Component({
    selector: 'app-home',
    template: `
        <h1>Welcome to the Home Page</h1>
        <a routerLink="/about">Go to About</a>
    `,
})
export class HomeComponent {}
```

Componente `about.component.ts`:

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

@Component({
    selector: 'app-about',
    template: `
        <h1>About Us</h1>
        <a routerLink="/">Go to Home</a>
    `,
})
export class AboutComponent {}
```

Para probar la **navegación** entre estos **componentes**, siga estos pasos:

Importar las dependencias necesarias en el archivo de prueba `app.component.spec.ts`:

```
import { TestBed, async } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
```

Configurar el entorno de prueba utilizando `TestBed`:

```
beforeEach(async () => {
    await TestBed.configureTestingModule({
        imports: [RouterTestingModule],
        declarations: [AppComponent, HomeComponent, AboutComponent],
    }).compileComponents();
});
```

Escribir pruebas para verificar la **navegación** entre los componentes.

En este caso, se verificará que al **hacer clic** en el enlace "Go to About" en el componente `HomeComponent`, se navegue al componente `AboutComponent`:

```
it('should navigate to the AboutComponent when clicking the "Go to About" link', async
() => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();

    const compiled = fixture.nativeElement;
    const homeLink = compiled.querySelector('a');
    homeLink.click();

    fixture.detectChanges();
    await fixture.whenStable();

    const routerOutlet = fixture.debugElement.query(By.directive(RouterOutlet));
    const componentInstance = routerOutlet.componentInstance;

    expect(componentInstance.constructor.name).toBe('AboutComponent');
});
```

En este ejemplo, se crea una instancia del componente `AppComponent`, se hace clic en el enlace "Go to About" y se espera a que la navegación se complete.

Luego, se verifica que el componente activo dentro del `RouterOutlet` sea el `AboutComponent`.

Esta prueba garantiza que la navegación entre los componentes funcione correctamente y que la configuración de enrutamiento esté bien definida.

Dependiendo de la **complejidad** de la aplicación y sus rutas, se pueden agregar más pruebas para cubrir diferentes escenarios de **navegación** y **enrutamiento**.

6. Testing de formularios

Para probar **formularios reactivos**, se puede utilizar el módulo `ReactiveFormsModule` y las utilidades de prueba proporcionadas por el framework.

Supongamos que tenemos un componente `LoginFormComponent` que utiliza un **formulario reactivo** para la autenticación de usuarios.

Archivo `login-form.component.ts`:

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
    selector: 'app-login-form',
    template: `
        <form [formGroup]="loginForm" (ngSubmit)="onSubmit()">
            <label>
                Email:
                <input type="email" formControlName="email" />
            </label>
            <label>
                Password:
                <input type="password" formControlName="password" />
            </label>
            <button type="submit" [disabled]="loginForm.invalid">Login</button>
        </form>
    `,
})
export class LoginFormComponent {
    loginForm: FormGroup;

    constructor(private fb: FormBuilder) {
        this.loginForm = this.fb.group({
          email: ['', [Validators.required, Validators.email]],
          password: ['', [Validators.required, Validators.minLength(8)]],
        });
    }

    onSubmit(): void {
        if (this.loginForm.valid) {
          console.log('Form submitted:', this.loginForm.value);
        }
    }
}
```

Para probar este componente y su formulario reactivo, siga estos pasos:

6.1. Importar las dependencias

Importar las dependencias necesarias en el archivo de prueba `login-form.component.spec.ts`:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { LoginFormComponent } from './login-form.component';
```

Configurar el entorno de prueba utilizando `TestBed` y `ComponentFixture`:

```
let component: LoginFormComponent;
let fixture: ComponentFixture<LoginFormComponent>;

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

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

6.2. Escribir pruebas

Escribir pruebas para verificar el comportamiento del formulario reactivo.

En este caso, se probará que los campos del formulario estén configurados correctamente y que se validen según las reglas establecidas:

```
it('should create the login form with the correct initial state and validation rules',
() => {
    const emailControl = component.loginForm.get('email');
    const passwordControl = component.loginForm.get('password');

    expect(emailControl.value).toBe('');
    expect(emailControl.valid).toBeFalsy();
    expect(emailControl.errors.required).toBeTruthy();
    expect(emailControl.errors.email).toBeTruthy();

    expect(passwordControl.value).toBe('');
    expect(passwordControl.valid).toBeFalsy();
```

```
    expect(passwordControl.errors.required).toBeTruthy();
    expect(passwordControl.errors.minLength).toBeTruthy();
});

it('should validate the email and password fields correctly', () => {
    const emailControl = component.loginForm.get('email');
    const passwordControl = component.loginForm.get('password');

    emailControl.setValue('test@example.com');
    passwordControl.setValue('password123');

    expect(emailControl.valid).toBeTruthy();
    expect(passwordControl.valid).toBeTruthy();
});

it('should disable the submit button when the form is invalid', () => {
    fixture.detectChanges();
    const compiled = fixture.nativeElement;
    const submitButton = compiled.querySelector('button[type="submit"]');

    expect(submitButton.disabled).toBeTruthy();

    component.loginForm.setValue({
        email: 'test@example.com',
        password: 'password123',
    });
    fixture.detectChanges();
    expect(submitButton.disabled).toBeFalsy();
});
```

Escribir pruebas para verificar que el método `onSubmit` se ejecute correctamente cuando se envía el formulario:

```
it('should call onSubmit when the form is submitted', () => {
    spyOn(component, 'onSubmit');
    component.loginForm.setValue({
        email: 'test@example.com',
        password: 'password123',
    });

    fixture.detectChanges();
    const compiled = fixture.nativeElement;
    const formElement = compiled.querySelector('form');
    formElement.dispatchEvent(new Event('submit'));

    expect(component.onSubmit).toHaveBeenCalled();
});

it('should not call onSubmit when the form is invalid', () => {
    spyOn(component, 'onSubmit');
```

```
    component.loginForm.setValue({
        email: 'invalid-email',
        password: 'short',
    });

    fixture.detectChanges();
    const compiled = fixture.nativeElement;
    const formElement = compiled.querySelector('form');
    formElement.dispatchEvent(new Event('submit'));

    expect(component.onSubmit).not.toHaveBeenCalled();
});
```

En este ejemplo, se verifica que el formulario reactivo esté correctamente configurado y validado.

Se comprueba que los campos de **correo electrónico** y **contraseña** tengan las reglas de validación adecuadas y que el **botón de envío** esté deshabilitado cuando el formulario no sea válido.

También se prueba que el método `onSubmit` se ejecute correctamente al **enviar el formulario** y que no se ejecute cuando el formulario no sea válido.

Estas pruebas garantizan que el componente `LoginFormComponent` funcione correctamente y que su formulario reactivo esté bien configurado y validado.

Dependiendo de la complejidad del formulario y sus requisitos, se pueden agregar más pruebas para cubrir diferentes escenarios y casos de uso.