

# TEMA 1

## Programación reactiva: Formularios reactivos

**Desarrollo front-end avanzado**

**Máster Universitario en Desarrollo de  
sitios y aplicaciones web**

UOC

### Contenido

- Programación reactiva
- Formularios reactivos vs dirigidos por el modelo
- Ejemplo formulario dirigido por el modelo
- Ejemplo formulario reactivo
- Ampliación validaciones formularios reactivos



# Introducción

A lo largo de este curso vamos a desarrollar una aplicación web completa. En cada práctica estudiaremos y aplicaremos un concepto nuevo e iremos añadiendo funcionalidades a nuestra aplicación progresivamente para que cada vez sea más compleja.

En este primer tema trataremos con los formularios web, es decir, con la forma que tiene el usuario de interactuar con nuestra aplicación para insertar información y recibir feedback a sus acciones.

Trataremos los formularios dirigidos por el modelo y posteriormente pondremos énfasis a los formularios reactivos que son con los que trabajaremos en nuestra aplicación definitiva.

**Este documento contiene la teoría y los ejercicios complementarios no evaluables a estudiar para tener los conocimientos suficientes para posteriormente desarrollar la primera práctica. El enunciado de la primera práctica se publicará en otro documento independiente.**

## Programación reactiva

La programación reactiva es un **paradigma de programación** que consta de dos fundamentos: 1) Manejo de flujos de datos asíncronos; y 2) Uso eficiente de recursos.

### 1) Manejo de flujos de datos asíncronos

La programación reactiva está basada en un patrón de diseño llamado **Observador**. Este patrón consiste en que cuando se produce un cambio en el estado de un objeto, otros objetos son notificados y actualizados acorde a dicho cambio. Es decir, se dispone de un conjunto de objetos observados y otros que están suscritos a los cambios (notificaciones) de estos objetos observados.

Por lo tanto, y más importante, los eventos se realizan de forma asíncrona. De esta manera los observadores no están constantemente preguntando al objeto observado si ha cambiado su estado.

### 2) Uso eficiente de recursos

Los sistemas reactivos permiten que los clientes (i.e. los objetos observadores que esperan el cambio sobre los observados) realicen otras tareas hasta que sean notificados del cambio, en lugar de hacer *polling*, es decir, en lugar de estar permanentemente comprobando si se ha producido un cambio.

Un claro ejemplo de uso de la programación reactiva son los diferentes eventos de **click** sobre la interfaz gráfica, ya que se genera un flujo de eventos asíncronos (puede hacerse click en un segundo y otro al cabo de 3 segundos o cualquier intervalo de tiempo), los cuales pueden ser observados y se puede reaccionar en consecuencia a lo que suceda

en dicho flujo (cada vez que se dispare uno, o cuando se acumulen unos cuantos eventos en un determinado intervalo de tiempo, etc.).

La programación reactiva suele ir acompañada de un conjunto de operadores que permiten la manipulación completa de los flujos de datos, y ahí es donde se puede aprovechar el verdadero potencial de este paradigma de programación. Este conjunto de operadores en nuestro contexto será proporcionado por la biblioteca Rx (ReactiveX), concretamente la versión para JavaScript (RxJS).

**En el siguiente tema profundizaremos sobre la programación reactiva utilizando RxJS. En este tema vamos a introducir los formularios reactivos existentes en el propio framework Angular y así sustituir los clásicos formularios dirigidos por el modelo.**

## Formularios reactivos vs dirigidos por el modelo

Para introducir la programación reactiva se puede comenzar utilizando los formularios reactivos proporcionados por Angular. En principio, Angular permite desarrollar formularios utilizando los dos paradigmas de programación, pero las principales características de uno y otro son las siguientes (<https://stackoverflow.com/a/41685151/3890755>):

- **Formularios dirigidos por el modelo (Template Driven Forms)**

- Fáciles de usar.
- Se ajustan adecuadamente en escenarios simples, pero no son útiles para escenarios complejos.
- Uso de formularios similar al utilizado en AngularJS (predecesor).
- Manejo automático del formulario y los datos por el propio framework de Angular.
- La creación de pruebas unitarias en este paradigma es compleja.

- **Formularios reactivos (Reactive Forms)**

- Más flexibles.
- Se ajustan adecuadamente en escenarios complejos.
- No existe *data-binding*, sino que se hace uso de un modelo de datos inmutable.
- Más código en el componente y menos en el lenguaje de marcas HTML.
- Se pueden realizar transformaciones reactivas, tales como:
  - Manejo de eventos basados en intervalos de tiempo.
  - Manejo de eventos cuando los componentes son distintos hasta que se produzca un cambio.
  - Añadir elementos dinámicamente.
- Fáciles de testear.

# Ejemplo de formulario dirigido por el modelo

Si bien vamos a explicar y trabajar los formularios reactivos posteriormente, y las prácticas evaluables las desarrollaremos con dicho paradigma, ahora vamos a poner un ejemplo de un formulario sencillo dirigido por el modelo para que podamos comparar tanto a nivel de teoría o concepto como a nivel de código como sería su implementación.

Además, seguramente en nuestro trabajo heredemos proyectos con implementaciones con este tipo de formularios, así que puede ser interesante mostrar una pincelada de este tipo de implementación.

Supongamos que queremos implementar un formulario para autenticarnos a una aplicación. Implementaremos un formulario sencillo con dos campos, un campo para el nombre de usuario y otro para la contraseña. Añadiremos un botón para lanzar la acción de autenticación (en nuestro caso la acción simplemente será mostrar un mensaje por la consola del navegador con los datos introducidos en el formulario) y dicho botón no estará activo hasta que el formulario sea válido, es decir, hasta que haya información introducida en los dos campos del formulario.

A continuación, enumeramos una propuesta de paso a paso a seguir:

- **Paso 1:**

Creamos un nuevo proyecto ejecutando el siguiente comando des de la consola:

```
H:\Projectes>ng new angular-test-driven-forms
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
```

**ng new << nombre\_proyecto >>**

Podemos indicar que si queremos que nos añada el **routing** de Angular y por ejemplo que utilizaremos el tipo de estilos **CSS**.

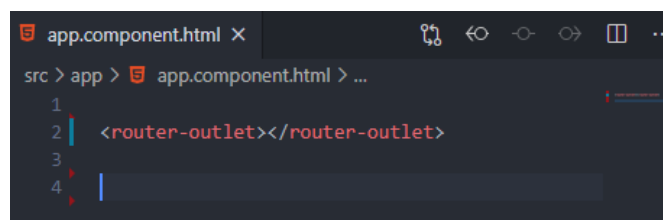
Ejecutamos el proyecto con la instrucción:

```
H:\Projectes\angular-test-driven-forms>ng serve --open
```

**ng serve --open**

Y validamos que vemos la pantalla inicial del proyecto en el navegador.

Veremos la pantalla inicial con mucha información de Angular que en principio no nos interesa, así que iremos al fichero **app.component.html** y eliminaremos todo su contenido a excepción de la línea:



```

src > app > app.component.html > ...
1
2 <router-outlet></router-outlet>
3
4

```

### • Paso 2:

Para empezar a trabajar con los formularios dirigidos por el modelo, necesitamos incluir **FormsModule** en el módulo principal de la aplicación (**app.module.ts**):



```

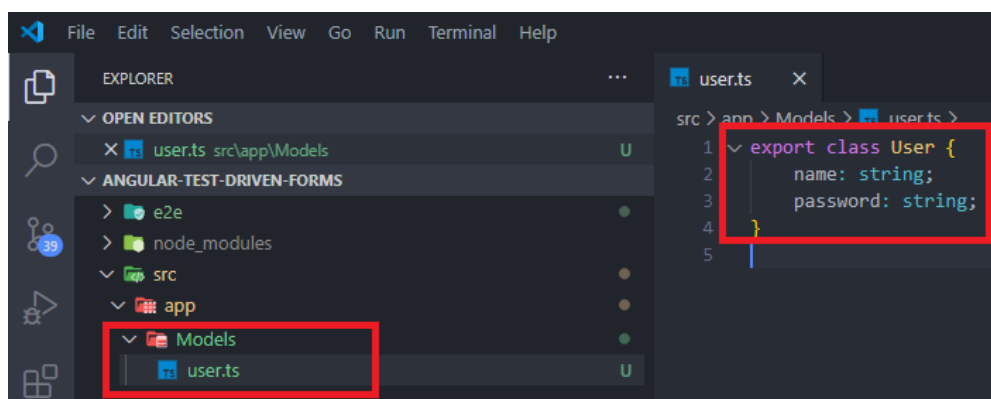
src > app > app.module.ts > ...
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppRoutingModule } from './app-routing.module';
5 import { AppComponent } from './app.component';
6 import { FormsModule } from '@angular/forms';
7
8 @NgModule({
9   declarations: [
10     AppComponent
11   ],
12   imports: [
13     BrowserModule,
14     AppRoutingModule,
15     FormsModule
16   ],
17   providers: [],
18   bootstrap: [AppComponent]
19 })
20 export class AppModule { }
21

```

### • Paso 3:

Como queremos hacer un formulario de autenticación, necesitaremos guardar al menos el nombre y la contraseña del usuario, por lo tanto, nos crearemos un modelo para tal fin.

Podemos crearnos una carpeta **Models** y dentro la clase **User**. Dicha clase tendría el siguiente código:



```

src > app > Models > user.ts >
1 export class User {
2   name: string;
3   password: string;
4 }
5

```

Para crear una clase así sencilla podemos hacer clic con el botón derecho del ratón en la carpeta **Models** y hacer **New File** y darle el nombre correspondiente.

- **Paso 4:**

Vamos a crear el componente de autenticación. Nos crearemos una carpeta a la misma altura que la carpeta anterior **Models** y la nombraremos **Components**. Acto seguido, desde la consola podemos ejecutar el siguiente comando:

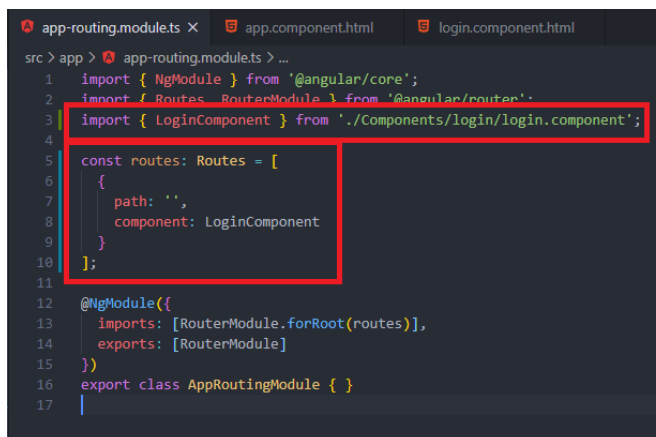
```
H:\Projectes\angular-test-driven-forms>ng g c Components/login
CREATE src/app/Components/login/login.component.html (20 bytes)
CREATE src/app/Components/login/login.component.spec.ts (619 bytes)
CREATE src/app/Components/login/login.component.ts (271 bytes)
CREATE src/app/Components/login/login.component.css (0 bytes)
UPDATE src/app/app.module.ts (545 bytes)
```

### ng g c Components/login

Al ejecutar esta instrucción podemos observar que nos ha incluido él mismo el componente **LoginComponent** dentro del apartado **declarations** del fichero **app.module.ts**. Debemos asegurarnos de que este nuevo componente este declarado en el **app.module.ts** para que lo podamos utilizar.

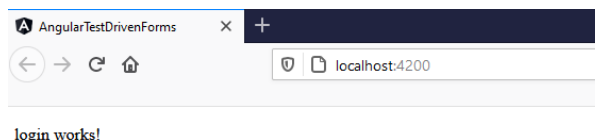
- **Paso 5:**

Validemos que el nuevo componente funciona. Vamos al fichero **app-routing.module.ts** y definimos que para la ruta de entrada **''** redirija al componente de autenticación **LoginComponent**:



```
src > app > app-routing.module.ts > ...
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3 import { LoginComponent } from '../Components/login/login.component';
4
5 const routes: Routes = [
6   {
7     path: '',
8     component: LoginComponent
9   }
10 ];
11
12 @NgModule({
13   imports: [RouterModule.forRoot(routes)],
14   exports: [RouterModule]
15 })
16 export class AppRoutingModule { }
17
```

Con esto, podremos ver al refrescarse nuestra aplicación algo así:



## • Paso 6:

Implementación del controlador **login.component.ts**:

```
src > app > Components > login > login.component.ts > ...
1 import { Component, OnInit } from '@angular/core';
2 import { User } from 'src/app/Models/user';
3
4 @Component({
5   selector: 'app-login',
6   templateUrl: './login.component.html',
7   styleUrls: ['./login.component.css']
8 })
9 export class LoginComponent implements OnInit {
10
11   public user: User = new User();
12
13   constructor() { }
14
15   ngOnInit(): void {
16   }
17
18   public checkLogin(){
19     console.log('User name --> ' + this.user.name + ' User password --> ' + this.user.password);
20   }
21
22 }
23
```

Primero declaramos una variable publica **user** y le asignamos nuestro modelo **User**.

Seguidamente, implementamos el método público **checkLogin** el cual se ejecutará cuando hagamos **submit** del formulario y su función será mostrar por la consola del navegador los valores de los campos del formulario, que en nuestro caso serán el nombre y la contraseña del usuario.

## • Paso 7:

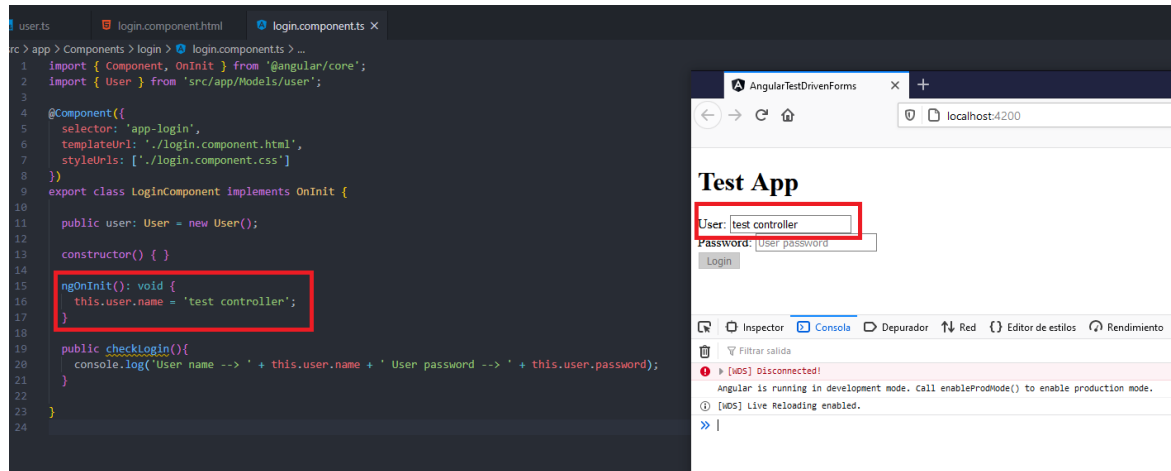
Implementación de la vista del componente **login.component.html**:

```
src > app > Components > login > login.component.html > ...
1 <div>
2   <h1>Test App</h1>
3   <form #loginForm="ngForm" (ngSubmit)="checkLogin()">
4     <div class="form-group">
5       <label for="name">User: </label>
6       <input type="text" name="name" [(ngModel)]="user.name" #name="ngModel" class="form-control" required placeholder="User name"/>
7       <span [hidden]="name.valid || name.pristine" style="color: red;">User name required</span>
8     </div>
9     <div class="form-group">
10      <label for="password">Password: </label>
11      <input type="password" name="password" [(ngModel)]="user.password" #password="ngModel" class="form-control" required placeholder="User password"/>
12      <span [hidden]="password.valid || password.pristine" style="color: red;">User password required</span>
13    </div>
14    <div class="form-group">
15      <button type="submit" class="btn btn-success" [disabled]="!loginForm.form.valid">Login</button>
16    </div>
17  </form>
18 </div>
19
```

## Explicación de los diferentes elementos:

Con la directiva **[(ngModel)]** de cada **<input>** vinculamos la vista con el controlador. Es una vinculación de doble sentido, esto quiere decir que cualquier cambio en el formulario se refleja en el modelo y viceversa.

En otras palabras, la información introducida en los campos **name** y **password** del formulario se guardaran en nuestro modelo **user.name** y **user.password** respectivamente. Y si en nuestro controlador asignáramos directamente valores a nuestro modelo, se mostrarían al formulario. Por ejemplo, si hiciéramos:



Asignamos el texto **'test controller'** a la variable **name** de nuestro modelo en el evento **ngOnInit** del controlador para que nada más iniciar el componente asigne esta información. Podemos ver como al cargar el formulario éste tiene el texto en el campo con la etiqueta **User**.

Por otra parte, con la directiva **ngForm** conseguimos tener el control del estado y la validez de los elementos del formulario que tienen la directiva **"ngModel"** y el atributo **name**. Esta directiva, a su vez, tiene su propia propiedad **valid** que es cierta cuando el formulario es válido, y podemos utilizar dicha variable, por ejemplo, para habilitar el botón de **submit** del formulario cuando todo sea correcto.

La directiva **"ngModel"** también la podemos utilizar para controlar si el usuario toca un campo, si el valor de dicho campo ha cambiado o si dicho valor se ha vuelto invalido.

Resumimos los estados en la siguiente tabla:

Estados de un campo		
Estado	Clase si el estado es cierto	Clase si el estado es falso
El control se ha visitado	ng-touched	ng-untouched
El valor del control ha cambiado	ng-dirty	ng-pristine
El valor del control es válido	ng-valid	ng-invalid

Algunos detalles importantes más de la vista:

- Nos fijamos que añadimos que los dos campos sean obligatorios con la etiqueta **required**

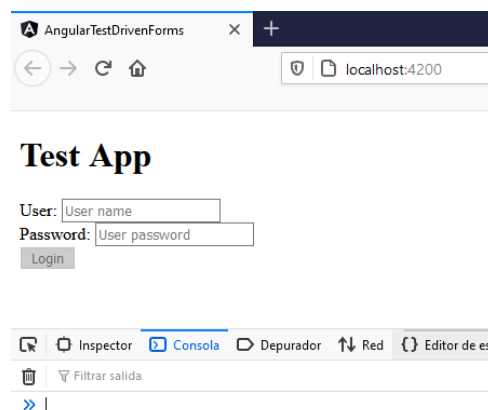


- Para poder controlar el estado de cada **input**, añadimos **#name="ngModel"** y **#password="ngModel"**
- Añadimos para cada **input** una etiqueta **span** con el mensaje de error. Con la directiva **hidden** conseguimos que se oculte el campo si el valor del campo asociado es válido (**password.valid**) o aún no ha sido tocado por el usuario (**password.pristine**)
- El botón de **submit** estará deshabilitado hasta que todos los campos del formulario sean válidos. Eso lo podemos realizar gracias a la variable **loginForm** que mantiene una referencia con la directiva **ngForm** y ésta controla el formulario como un todo.

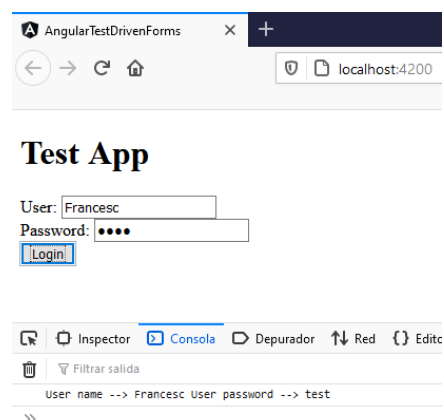
### • Paso 8:

Comprobación funcionalidad:

Este sería el estado inicial.

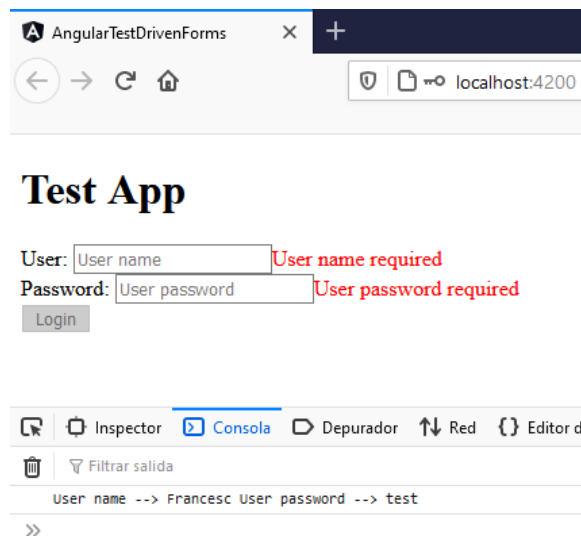


Si introducimos un usuario y una contraseña, se habilita el botón **Login**:



Y si hacemos clic en él veremos por consola los valores que se han introducido en los dos campos.

Si quitamos la información de uno o de los dos campos, se mostrará el mensaje de aviso en rojo y se deshabilitará el botón de **login**:



## Actividad complementaria



*“Hay que comentar que este tipo de actividades complementarias no son obligatorias ni evaluables, son ejercicios para practicar que recomendamos realizar para asimilar todos los conceptos progresivamente.”*

Una vez estudiado el formulario guiado por el modelo anterior, te proponemos ampliar un poco sus características para practicar, concretamente, te proponemos que:

- Amplíes el modelo con los campos apellidos, fecha de nacimiento, email y edad.
- Apliques las validaciones siguientes:
  - Los campos email y edad son obligatorios.
  - Los campos nombre y apellidos deben tener entre 5 y 25 caracteres.
  - La fecha debe tener un formato DD/MM/YYYY.
  - La edad debe ser superior a 18 años.
- Muestra la nueva información por consola cuando se pulse el botón de *Login*.



*“Comparte con tus compañeros los resultados de esta actividad complementaria en el foro del aula.”*

Con esto hemos visto un ejemplo sencillo de un formulario dirigido por el modelo. Ahora vamos a estudiar la implementación equivalente con formularios reactivos.

## Ejemplo de formulario reactivo

Una vez hemos podido estudiar los formularios dirigidos por el modelo, vamos a centrarnos en los formularios reactivos. Pensemos que dichos formularios son con los que deberemos implementar la práctica cuyo enunciado veremos en otro documento.

Vamos a implementar el mismo formulario de autenticación que en el ejemplo anterior pero esta vez con el paradigma de formulario reactivo.

A continuación, enumeramos una propuesta de paso a paso a seguir:

- **Paso 1:**

Creamos un nuevo proyecto ejecutando el siguiente comando des de la consola:

```
H:\Projectes>ng new angular-test-reactive-forms
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
```

**ng new << nombre\_proyecto >>**

Podemos indicar que si queremos que nos añada el **routing** de Angular y por ejemplo que utilizaremos el tipo de estilos **CSS**.

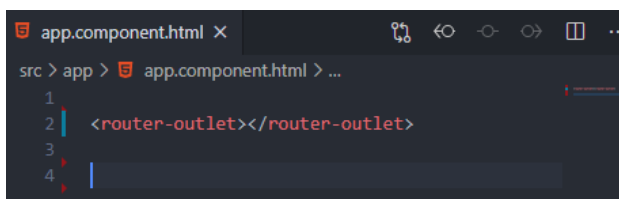
Ejecutamos el proyecto con la instrucción:

```
H:\Projectes\angular-test-reactive-forms>ng serve --open
```

**ng serve --open**

Y validamos que vemos la pantalla inicial del proyecto en el navegador.

Veremos la pantalla inicial con mucha información de Angular que en principio no nos interesa, así que iremos al fichero **app.component.html** y eliminaremos todo su contenido a excepción de la línea:



## • Paso 2:

Para empezar a trabajar con los formularios reactivos, necesitamos incluir **ReactiveFormsModule** en el módulo principal de la aplicación (**app.module.ts**):

```

src > app > app.module.ts > ...
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppRoutingModule } from './app-routing.module';
5  import { AppComponent } from './app.component';
6  import { ReactiveFormsModule } from '@angular/forms';
7
8  @NgModule({
9    declarations: [
10     AppComponent
11   ],
12   imports: [
13     BrowserModule,
14     AppRoutingModule,
15     ReactiveFormsModule
16   ],
17   providers: [],
18   bootstrap: [AppComponent]
19 })
20 export class AppModule { }
21

```

Dicho módulo nos permitirá acceder a los componentes, directivas y **providers** reactivos tales como **FormBuilder**, **FormGroup** y **FormControl**.

El **FormControl** lo utilizaremos para registrar un único campo de entrada del formulario. Por ejemplo, en nuestro caso que tendremos dos campos, **name** y **password**, tendremos dos bloques **FormControl**. El constructor de **FormControl** lo podremos utilizar para inicializar el valor del control.

## • Paso 3:

Como queremos hacer un formulario de autenticación, necesitaremos guardar al menos el nombre y la contraseña del usuario, por lo tanto, nos crearemos un modelo para tal fin.

Podemos crearnos una carpeta **Models** y dentro la clase **User**. Dicha clase tendría el siguiente código:

```

src > app > Models > user.ts > ...
1  export class User {
2    name: string;
3    password: string;
4  }
5

```

Para crear una clase así sencilla podemos hacer clic con el botón derecho del ratón en la carpeta **Models** y hacer **New File** y darle el nombre correspondiente.

#### • Paso 4:

Vamos a crear el componente de autenticación. Nos crearemos una carpeta a la misma altura que la carpeta anterior **Models** y la nombraremos **Components**. Acto seguido, desde la consola podemos ejecutar el siguiente comando:

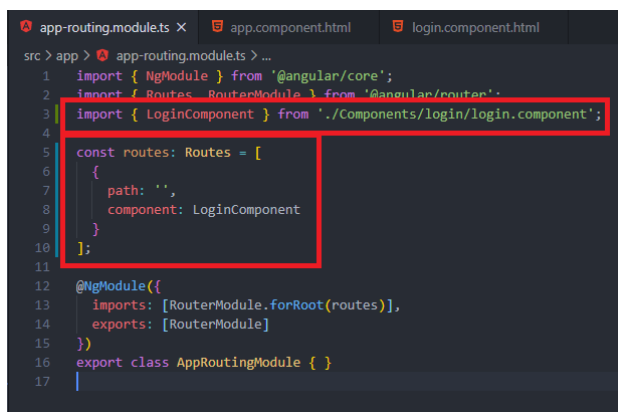
```
H:\Projectes\angular-test-reactive-forms>ng g c Components/login
CREATE src/app/Components/login/login.component.html (20 bytes)
CREATE src/app/Components/login/login.component.spec.ts (619 bytes)
CREATE src/app/Components/login/login.component.ts (271 bytes)
CREATE src/app/Components/login/login.component.css (0 bytes)
UPDATE src/app/app.module.ts (561 bytes)
```

**ng g c Components/login**

Al ejecutar esta instrucción podemos observar que nos ha incluido él mismo el componente **LoginComponent** dentro del apartado **declarations** del fichero **app.module.ts**. Debemos asegurarnos de que este nuevo componente este declarado en el **app.module.ts** para que lo podamos utilizar.

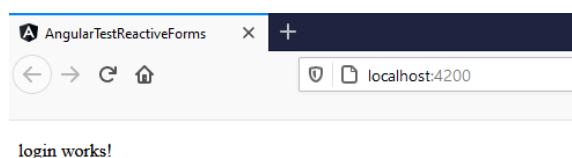
#### • Paso 5:

Validemos que el nuevo componente funciona. Vamos al fichero **app-routing.module.ts** y definimos que para la ruta de entrada **''** redirija al componente de autenticación **LoginComponent**.



```
src > app > app-routing.module.ts > ...
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3 import { LoginComponent } from '../Components/login/login.component';
4
5 const routes: Routes = [
6   {
7     path: '',
8     component: LoginComponent
9   }
10 ];
11
12 @NgModule({
13   imports: [RouterModule.forRoot(routes)],
14   exports: [RouterModule]
15 })
16 export class AppRoutingModule { }
17
```

Con esto, podremos ver al refrescarse nuestra aplicación algo así:



AngularTestReactiveForms x +

localhost:4200

login works!

## • Paso 6:

Primera versión de la implementación del controlador **login.component.ts**:

```
login.component.ts X
src > app > Components > login > login.component.ts > ...
1  import { Component, OnInit } from '@angular/core';
2  import { FormControl, FormGroup, FormBuilder } from '@angular/forms';
3  import { User } from 'src/app/Models/user';
4
5  @Component({
6    selector: 'app-login',
7    templateUrl: './login.component.html',
8    styleUrls: ['./login.component.css']
9  })
10 export class LoginComponent implements OnInit {
11
12    public user: User = new User();
13
14    public name: FormControl;
15    public password: FormControl;
16    public loginForm: FormGroup;
17
18    constructor( private FormBuilder: FormBuilder) { }
19
20    ngOnInit(): void {
21
22        this.name = new FormControl('');
23        this.password = new FormControl('');
24
25        this.loginForm = this.formBuilder.group({
26            name: this.name,
27            password: this.password
28        });
29    }
30
31
32    public checkLogin(){
33        this.user.name = this.name.value;
34        this.user.password = this.password.value;
35        console.log('User name --> ' + this.user.name + ' User password --> ' + this.user.password);
36    }
37
38
39 }
```

### Sección (A):

Primero declaramos una variable publica **user** y le asignamos nuestro modelo **User**.

Seguidamente, declaramos un **FormControl** por cada entrada de nuestro formulario, en este caso uno para el campo **name** y otro para el campo **password**. De esta manera, lo que conseguimos es registrar los controles del formulario.

Después declaramos un **FormGroup** que lo llamaremos **loginForm**. De esta manera, todas las entradas del formulario se agruparán dentro de este grupo.

### Sección (B):

Es importante resaltar que el servicio (*provider*) **FormBuilder** debe ser inyectado para poder construir los formularios haciendo uso de **FormGroup** y **FormControl**.

### Sección (C):

Registramos los **FormControl** para los campos **name** y **password** y posteriormente los agruparemos dentro del grupo **loginForm**. Posteriormente en la vista veremos que esta variable **loginForm** se la asignaremos a la directiva **[formGroup]** del formulario.

### Sección (D):

Finalmente, implementamos el método **checkLogin** que se llamará cuando se haga **submit** del formulario. Podemos ver que para acceder a los valores de los diferentes controles que están dentro de un grupo podemos acceder a la propiedad **value** y así hacer las acciones pertinentes, en nuestro caso, por ejemplo, mapear los datos a nuestro modelo y mostrar los datos por la consola del navegador.

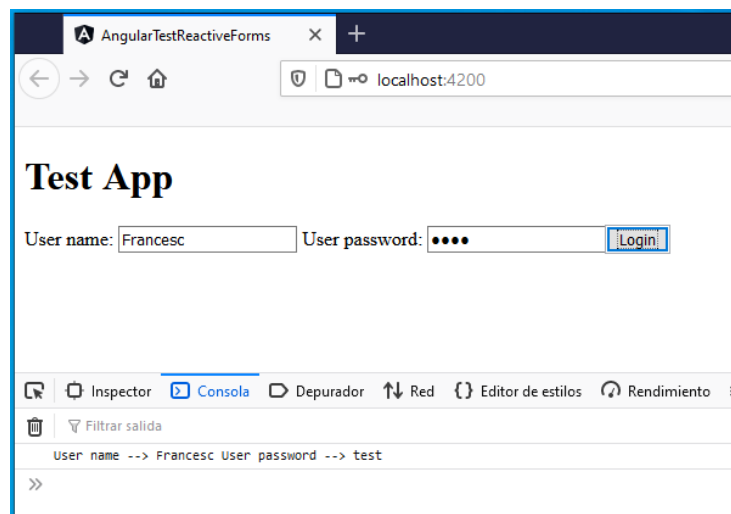
- **Paso 7:**

Primera versión de la implementación de la vista del componente **login.component.html**:

```
login.component.html X
src > app > Components > login > login.component.html > ...
1  <div>
2    <h1>Test App</h1>
3    <form [formGroup]="loginForm" (ngSubmit)="checkLogin()">
4
5      <label>User name:</label>
6      <input type="text" [formControl]="name"/>
7
8      <label> User password:</label>
9      <input type="password" [formControl]="password"/>
10
11     <button type="submit" [disabled]="!loginForm.valid">Login</button>
12
13   </form>
14 </div>
15
```

Si bien en el controlador hemos registrado los dos controles (**name** y **password**), aquí en la vista vinculamos dichos controles con la directiva **[formControl]**. Con esto obtenemos la comunicación entre vista y controlador, o, en otras palabras, entre el **formControl** y el elemento **DOM**.

Tal y como tenemos la implementación hasta este momento, podemos ver la ejecución en el navegador. Podemos observar que, si introducimos valores en los dos campos y pulsamos el botón de **login**, se mostraran por consola los valores insertados.



Debemos tener en cuenta que **FormControl** acepta un **string** al constructor, con lo que podríamos utilizarlo para inicializar un campo con un valor determinado, por ejemplo, si escribimos:

```
this.name = new FormControl('Testing reactive forms!');
```

Al refrescarse el navegador veríamos:

## Test App

User name:  User password:

También podemos observar que el botón de **login** nunca esta deshabilitado, y esto es debido a que para que funcione tenemos que añadir algunas validaciones que haremos a continuación.

### • Paso 8:

#### Añadiendo validaciones básicas.

Angular Reactive Forms (<https://angular.io/guide/form-validation#reactiveform-validation>) incorpora una serie de validaciones que se definen a la hora de construir el formulario (mira <https://angular.io/api/forms/Validators>).

De esta manera es bastante simple definir en el control de un formulario, su valor inicial y un *array* con las diferentes validaciones que permitirán controlar el cambio de estado de dicho control.

Para poder trabajar con las validaciones necesitamos importar **Validators** en nuestro controlador:

```
login.component.ts x login.component.html
src > app > Components > login > login.component.ts > LoginComponent > ngOnInit
1 import { Component, OnInit } from '@angular/core';
2 import { FormControl, FormGroup, FormBuilder, Validators } from '@angular/forms';
3 import { User } from 'src/app/models/user';
4
5 @Component({
6   selector: 'app-login',
7   templateUrl: './login.component.html',
8   styleUrls: ['./login.component.css']
9 })
10 export class LoginComponent implements OnInit {
11
12   public user: User = new User();
13
14   public name: FormControl;
15   public password: FormControl;
16   public loginForm: FormGroup;
17
18   constructor( private formBuilder: FormBuilder ) { }
19
20   ngOnInit(): void {
21     this.name = new FormControl('', Validators.required);
22     this.password = new FormControl('', Validators.required);
23
24     this.loginForm = this.formBuilder.group({
25       name: this.name,
26       password: this.password
27     });
28
29   }
30
31   public checkLogin(){
32     this.user.name = this.name.value;
33     this.user.password = this.password.value;
34     console.log('User name --> ' + this.user.name + ' User password --> ' + this.user.password);
35   }
36 }
37
38 }
```

Y vamos a añadir que los dos campos del formulario sean obligatorios. Si ejecutamos ahora la aplicación, podremos ver que de entrada tenemos el botón de **login**



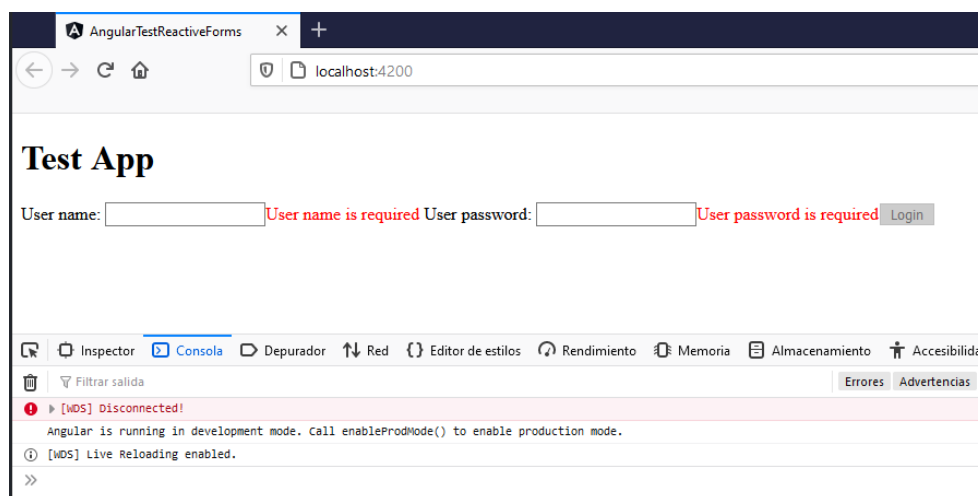
deshabilitado, y en el momento en que haya información en los dos campos éste se habilitará y si se pulsa mostrará la información correcta por la consola del navegador.

Ahora, vamos a darle un poco de feedback al usuario para que sepa porque no puede hacer clic al botón de **login** cuando lo tenga deshabilitado.

```
login.component.ts login.component.html x
src > app > Components > login > login.component.html > ...
1 <div>
2 <h1>Test App</h1>
3 <form [formGroup]="loginForm" (ngSubmit)="checkLogin()">
4
5 <label>User name: </label>
6 <input type="text" [formControl]="name"/>
7
8 <span style="color: red;" *ngIf="name.invalid && name?.errors.required">User name is required</span>
9
10 <label> User password: </label>
11 <input type="password" [formControl]="password"/>
12
13 <span style="color: red;" *ngIf="password.invalid && password?.errors.required">User password is required</span>
14
15 <button type="submit" [disabled]="!loginForm.valid">Login</button>
16
17 </form>
18 </div>
19
```

Añadimos estas dos líneas a la vista para que si el campo no esta informado muestre un mensaje de error en rojo.

Ahora si ejecutamos podemos ver:



Inicialmente nos indica los mensajes de error, que en este caso sería correcto. Si inserimos algo en un campo o en otro desaparecerá el mensaje de error y en el momento en que haya información en los dos campos se habilitará el botón de **login**.

Esto sería correcto, pero claro, no es correcto mostrar el mensaje de error nada más cargar la página ya que el usuario aún no ha interactuado con ella, por lo tanto, vamos a mejorar esta parte.

Fijémonos en la siguiente captura de pantalla:

```
login.component.ts X login.component.html
src > app > Components > login > login.component.ts > ...
1 import { Component, OnInit } from '@angular/core';
2 import { FormControl, FormGroup, FormBuilder, Validators } from '@angular/forms';
3 import { User } from 'src/app/Models/user';
4
5 @Component({
6   selector: 'app-login',
7   templateUrl: './login.component.html',
8   styleUrls: ['./login.component.css']
9 })
10 export class LoginComponent implements OnInit {
11
12   public user: User = new User();
13
14   public name: FormControl;
15   public password: FormControl;
16   public loginForm: FormGroup;
17
18   constructor( private formBuilder: FormBuilder) { }
19
20   ngOnInit(): void {
21
22     this.name = new FormControl('', [Validators.required, Validators.minLength(8)]);
23     this.password = new FormControl('', [Validators.required, Validators.minLength(4)]);
24
25     this.loginForm = this.formBuilder.group({
26       name: this.name,
27       password: this.password
28     });
29   }
30
31   public checkLogin(){
32     this.user.name = this.name.value;
33     this.user.password = this.password.value;
34     console.log('User name --> ' + this.user.name + ' User password --> ' + this.user.password);
35   }
36 }
37
38
39
```

Vamos a aprovechar para añadir alguna validación más, en este caso, que el campo **name** como mínimo tenga 8 caracteres y que el campo **password** como mínimo tenga 4. Nótese que en el momento en que queremos definir más de una validación al constructor del **FormControl** en el segundo parámetro se lo pasamos como un array de validaciones.

Como queremos dar feedback al usuario de los errores cuando haya interactuado con el formulario, vamos a modificar la vista de la siguiente manera:

```
login.component.ts X login.component.html X
src > app > Components > login > login.component.html > ...
1 <div>
2   <h1>Test App</h1>
3   <form [formGroup]="loginForm" (ngSubmit)="checklogin()">
4
5     <label>User name: </label>
6     <input type="text" [formControl]="name"/>
7
8     <label> User password: </label>
9     <input type="password" [formControl]="password"/>
10
11     <button type="submit" [disabled]="!loginForm.valid">Login</button>
12
13   </form>
14 </div>
15
16 <span style="color: red;" *ngIf="loginForm.get('name').errors && (loginForm.get('name').touched || loginForm.get('name').dirty)">
17   <span *ngIf="loginForm.get('name').errors.required">User name is required</span>
18   <span *ngIf="loginForm.get('name').errors.minlength || loginForm.get('name').errors.maxlength">User name must be greater than 8 characters</span>
19 </span>
20
21 <span style="color: red;" *ngIf="loginForm.get('password').errors && (loginForm.get('password').touched || loginForm.get('password').dirty)">
22   <span *ngIf="loginForm.get('password').errors.required">User password is required</span>
23   <span *ngIf="loginForm.get('password').errors.minlength || loginForm.get('password').errors.maxlength">User password must be greater than 4 characters</span>
24 </span>
25
```

Si ahora ejecutamos la aplicación observaremos el siguiente comportamiento:

- De un inicio no nos muestra mensajes de error y tendremos el botón de **login** deshabilitado.
- Si empezamos a escribir en el campo **name** nos aparecerá un mensaje de que el campo debe tener al menos 8 caracteres, si superamos dicho numero el mensaje desaparecerá.
- Si quitamos lo que estábamos escribiendo, aparecerá el mensaje de campo requerido.
- De manera semejante funciona el campo **password**.
- Cuando tengamos información correcta en los dos campos se habilitará el botón de **login**.

Podemos ver que hemos jugado con los estados **touched** y **dirty**.

El control del estado del formulario tanto para dar feedback al usuario como para realizar diferentes validaciones viene definido por los siguientes estados:

```

/* field value is valid */
.ng-valid {}

/* field value is invalid */
.ng-invalid {}

/* field has not been clicked in, tapped on, or tabbed over */
.ng-untouched {}

/* field has been previously entered */
.ng-touched {}

/* field value is unchanged from the default value */
.ng-pristine {}

/* field value has been modified from the default */
.ng-dirty {}

```

Como puedes observar, existen los siguientes pares:

- valid / invalid
- untouched / touched
- pristine / dirty

Además, si lees la documentación oficial sobre la clase **FormControl** (<https://angular.io/api/forms/FormControl>), puedes ver que existen los atributos valid, invalid, pristine, dirty, touched, untouched para poder gestionar programáticamente el control.

Igualmente, puedes leer la documentación del **FormGroup** (<https://angular.io/api/forms/FormGroup>), el cual permite saber si el grupo completamente es válido (cumple con las validaciones) y así permitir habilitar un botón de envío del formulario.

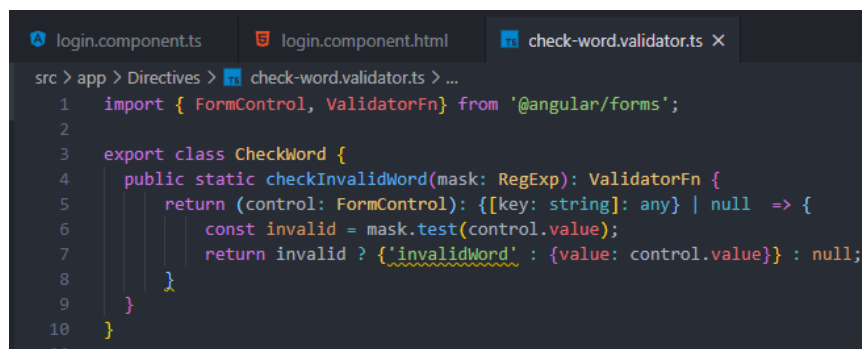
### • Paso 9:

#### Añadiendo validaciones propias.

Aunque la clase **Validators** permite disponer de un conjunto de validadores de base, a veces requerimos construir nuestros propios validadores (por ejemplo, dos campos coinciden para el uso de *passwords*, o cualquier otra idea). Para ello necesitamos recurrir a crear nuestros propios validadores.

Vamos a añadir una validación que mantenga el botón de **login** deshabilitado en caso de que en el campo **name** se inserte la palabra **administrator**.

Primero nos creamos una directiva:



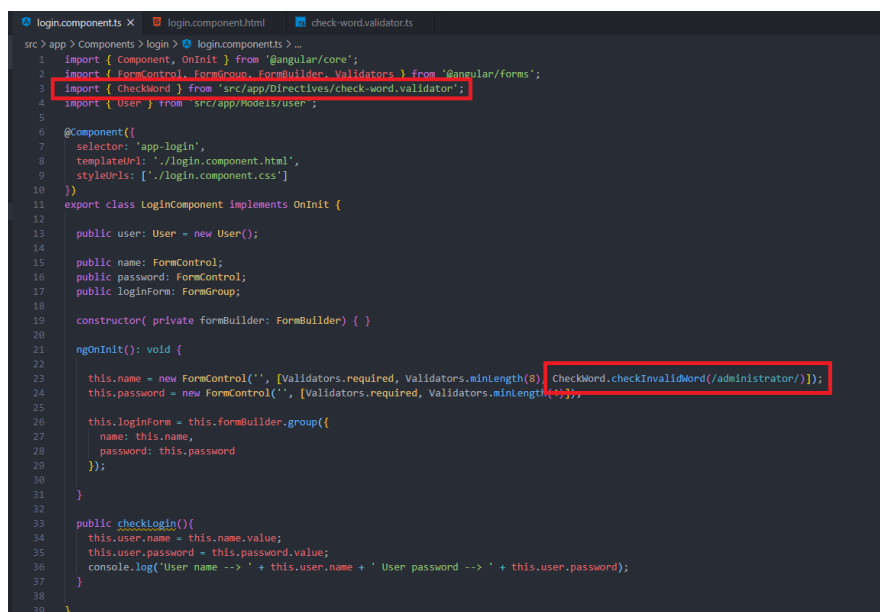
```

src > app > Directives > check-word.validator.ts > ...
1  import { FormControl, ValidatorFn } from '@angular/forms';
2
3  export class CheckWord {
4      public static checkInvalidWord(mask: RegExp): ValidatorFn {
5          return (control: FormControl): {[key: string]: any} | null => {
6              const invalid = mask.test(control.value);
7              return invalid ? {'invalidWord': {value: control.value}} : null;
8          }
9      }
10 }
11

```

Podemos crear un directorio **Directives** y dentro creamos la clase **CheckWord** con el código de la captura anterior.

Después, para añadir nuestra validación propia simplemente en el controlador importaremos la clase anterior y le pasaremos nuestra función de validación en el array de validaciones del **FormControl** que necesitemos. Lo mostramos en la captura siguiente:

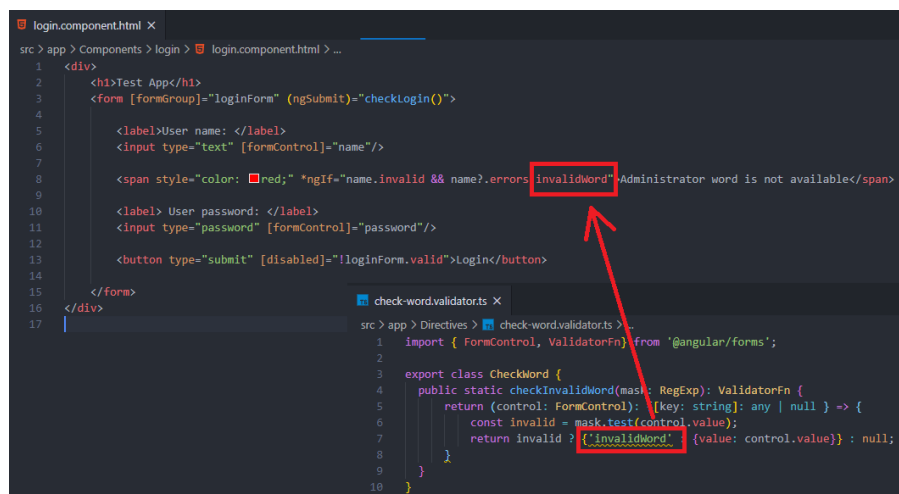


```

login.component.ts X login.component.html X check-word.validator.ts
src > app > Components > login > login.component.ts > ...
1  import { Component, OnInit } from '@angular/core';
2  import { FormControl, FormGroup, FormBuilder, Validators } from '@angular/forms';
3  import { CheckWord } from 'src/app/Directives/check-word.validator';
4  import { User } from 'src/app/Models/user';
5
6  @Component({
7      selector: 'app-login',
8      templateUrl: './login.component.html',
9      styleUrls: ['./login.component.css']
10 })
11 export class LoginComponent implements OnInit {
12
13     public user: User = new User();
14
15     public name: FormControl;
16     public password: FormControl;
17     public loginForm: FormGroup;
18
19     constructor( private formBuilder: FormBuilder ) { }
20
21     ngOnInit(): void {
22
23         this.name = new FormControl('', [Validators.required, Validators.minLength(8), CheckWord.checkInvalidWord(/administrator/)]);
24         this.password = new FormControl('', [Validators.required, Validators.minLength(8)]);
25
26         this.loginForm = this.formBuilder.group({
27             name: this.name,
28             password: this.password
29         });
30     }
31
32
33     public checkLogin(){
34         this.user.name = this.name.value;
35         this.user.password = this.password.value;
36         console.log('User name --> ' + this.user.name + ' User password --> ' + this.user.password);
37     }
38
39 }

```

Podemos ejecutar la aplicación y ver que el comportamiento es el esperado. Si insertamos una palabra en el campo **name** que contenga la palabra **administrator** el botón de **login** se deshabilitará.



```

login.component.html
1 <div>
2   <h1>Test App</h1>
3   <form [formGroup]="loginForm" (ngSubmit)="checkLogin()">
4
5     <label>User name: </label>
6     <input type="text" [formControl]="name"/>
7
8     <span style="color: red;" *ngIf="name.invalid && name.errors.invalidWord">Administrator word is not available</span>
9
10    <label> User password: </label>
11    <input type="password" [formControl]="password"/>
12
13    <button type="submit" [disabled]="!loginForm.valid">Login</button>
14  </form>
15 </div>
16
17
check-word.validators.ts
1 import { FormControl, ValidatorFn } from '@angular/forms';
2
3 export class CheckWord {
4   public static checkInvalidWord(mask: RegExp): ValidatorFn {
5     return (control: FormControl): { [key: string]: any | null } => {
6       const invalid = mask.test(control.value);
7       return invalid ? {invalidWord: {value: control.value}} : null;
8     }
9   }
10 }

```

Para finalizar, observamos que de nuestra validación personalizada devolvemos **invalidWord** cuando no se cumple la condición. Esto se traduce en que, en la vista, la propiedad **errors** tiene una propiedad **invalidWord** que se cumplirá cuando la validación de nuestra directiva **checkInvalidWord** falle.

Resultado:

## Test App

User name:  Administrator word is not available User password:

## Actividad complementaria



“Hay que comentar que este tipo de actividades complementarias no son obligatorias ni evaluables, son ejercicios para practicar que recomendamos realizar para asimilar todos los conceptos progresivamente.”

Una vez estudiado el formulario reactivo anterior, te proponemos ampliar un poco sus características para practicar, concretamente, te proponemos que:

- Amplíes el modelo con los campos **apellidos**, **fecha de nacimiento**, **email**, **edad**, **teléfono**, **descripción**.

- Apliques las validaciones siguientes:
  - Los campos email, edad y teléfono son obligatorios.
  - Los campos nombre y apellidos deben tener entre 5 y 25 caracteres.
  - El campo descripción no puede superar los 200 caracteres.
  - La fecha debe tener un formato DD/MM/YYYY.
  - La edad debe ser superior a 18 años.
  - Implementa una validación de manera que en el campo **nombre** y el campo **apellidos** no puedan ser iguales.
- Muestra la nueva información por consola cuando se pulse el botón de *Login*.



*“Comparte con tus compañeros los resultados de esta actividad complementaria en el foro del aula.”*

Con esto hemos visto un ejemplo sencillo de un formulario reactivo. Observa que los *templates* (archivos .html) quedan bastante limpios, puesto que solamente se definen las directivas **FormControl** y **FormGroup**.

Para tener una completa guía de los formularios reactivos puedes seguir el tutorial mostrado en la documentación oficial ( <https://angular.io/guide/reactive-forms> ) y finalmente, si quieres ver ejemplos en los que se comparan los formularios dirigidos por modelo con los reactivos puedes leer el siguiente tutorial ( <https://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/> ).

En el siguiente documento presentaremos el enunciado de la primera práctica evaluable.