

TEMA 1

Programación reactiva: Formularios reactivos

Desarrollo front-end avanzado

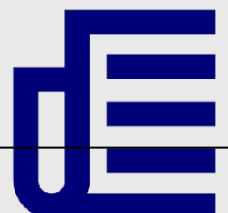
**Máster Universitario en Desarrollo de
sitios y aplicaciones web**

UOC

Universitat Oberta
de Catalunya

Contenido

- 1) Introducción (pág. 2)
- 2) Programación reactiva (pág. 2)
- 3) Formularios reactivos vs dirigidos por el modelo (pág. 3)
- 4) Ejemplo formulario dirigido por el modelo (pág. 4)
- 5) Ejemplo formulario reactivo (pág. 17)
- 6) Ampliación validaciones formularios reactivos (pág. 29)
- 7) Ejemplo MessageApp (con backend real) (pág. 34)



1) Introducción

A lo largo de esta asignatura iremos desarrollando una aplicación web completa que implementará diferentes funcionalidades. Para tal fin, en cada práctica de la asignatura estudiaremos y aplicaremos un concepto nuevo e iremos añadiendo funcionalidades a nuestra aplicación progresivamente para que cada vez sea más compleja. La idea de la asignatura será consolidar los conocimientos de la asignatura previa a esta e implementar funcionalidades o tecnologías más avanzadas que iremos estudiando a lo largo de la teoría y las prácticas.

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

Inicialmente estudiaremos los formularios dirigidos por el modelo y posteriormente pondremos énfasis a los formularios reactivos que son con los que trabajaremos en nuestra aplicación definitiva. Podríamos decir que los formularios dirigidos por el modelo son como se utilizaban los formularios web “antes”, y que lo que se utiliza actualmente son los formularios reactivos. Con estos últimos son los que deberíamos trabajar en proyectos reales y son con los que trabajaremos a lo largo del curso en nuestro proyecto. No obstante, no está de más repasar los formularios dirigidos por el modelo ya que así tenemos la posibilidad de comparar entre las dos metodologías y también tenemos una visión general de cómo funcionan por si se diera el caso que nos lo encontramos en un proyecto real en nuestro trabajo.

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.

2) Programación reactiva

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

a) 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.

b) 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 las extensiones 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.

3) 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.

4) 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 email del 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:**

Creemos un nuevo proyecto ejecutando el siguiente comando desde la consola:

ng new << nombre_proyecto >>

```
C:\Projects>ng new driven-forms-example
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS [ https://sass-lang.com/documentation/syntax#scss ]
```

Esto nos creará una carpeta con el nombre del proyecto, en mi caso “driven-forms-example” dentro del directorio “C:\Projects” donde estaba situado. Dentro de la carpeta con el nombre del proyecto tendremos todos los ficheros del proyecto Angular.

Cuando creamos el proyecto, para este caso de ejemplo, podemos indicar que si queremos que nos añada el **routing** de Angular y por ejemplo que utilizaremos el tipo de estilos **SCSS**. Aunque en estos ejemplos iniciales no llegaremos a hacer uso de rutas o estilos, o al menos, no de manera avanzada, así que no tiene mucha importancia esta elección. A comentar que, en proyectos reales sí que es interesante escoger **SCSS** en lugar del clásico **CSS** ya que **SCSS** nos permite utilizar/declarar variables en los estilos, con lo que podemos trabajar de manera más óptima, parametrizando ciertas propiedades, ...

Seguimos, ejecutamos el proyecto con la instrucción:

ng serve --open

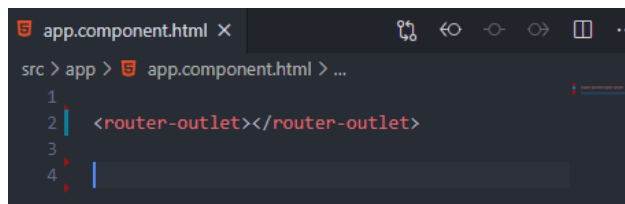
```
C:\Projects\driven-forms-example>ng serve --open
```

Y validamos que vemos la pantalla inicial del proyecto en el navegador, por defecto accediendo a la url:

<http://localhost:4200/>

Abriremos el proyecto con el **Visual Code**, y empezaremos la implementación.

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:



```
1 <router-outlet></router-outlet>
```

Podemos aprovechar este momento para aplicar la configuración de “inicialización de propiedades estricta de **TypeScript**”, concretamente añadiremos la siguiente línea en el fichero de configuración **tsconfig.json**:



```
1 /* To learn more about this file see: https://angular.io/config/tsconfig. */
2 {
3   "compileOnSave": false,
4   "compilerOptions": {
5     "baseUrl": "./",
6     "outDir": "./dist/out-tsc",
7     "forceConsistentCasingInFileNames": true,
8     "strict": true,
9     "noImplicitReturns": true,
10    "noFallthroughCasesInSwitch": true,
11    "sourceMap": true,
12    "declaration": false,
13    "downlevelIteration": true,
14    "experimentalDecorators": true,
15    "strictPropertyInitialization": true,
16    "moduleResolution": "node",
17    "importHelpers": true,
18    "target": "es2017",
19    "module": "es2020",
20    "lib": ["es2018", "dom"]
21  },
22   "angularCompilerOptions": {
23     "enableI18nLegacyMessageIdFormat": false,
24     "strictInjectionParameters": true,
25     "strictInputAccessModifiers": true,
26     "strictTemplates": true
27   }
28 }
```

Posteriormente veremos poco a poco en que nos implica esta configuración.

• 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**):

```

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

• Paso 3:

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

Podemos crearnos una carpeta **Models** y dentro la clase **UserDto**. En **TypeScript** es muy importante “tiparlo” todo, lo iremos viendo a lo largo de la asignatura. También iremos viendo cuando nos conviene utilizar una interfaz o cuando una clase, ahora para este ejemplo en el que sólo queremos mapear dos propiedades de una entidad usuario, con una clase nos sería suficiente. Finalmente hay que comentar que el **DTO** viene de **data transfer object**, es el convenio que se utiliza para identificar los objetos que utilizamos para mapear las entidades que nos vienen desde un backend. Todo esto lo iremos viendo poco a poco.

Dicha clase tendría el siguiente código:

```

1 export class UserDto {
2   email: string;
3   password: string;
4
5   constructor(email: string, password: string) {
6     this.email = email;
7     this.password = password;
8   }
9 }
10
  
```

Fijémonos que el hecho de aplicar la configuración anterior **"strictPropertyInitialization"** a **true** nos obliga a implementar un constructor. Otra alternativa sería no implementar el constructor y añadir el símbolo **!** detrás de cada propiedad, de esta manera no estaríamos obligados a implementar el constructor y podríamos inicializar las variables a posteriori, pero en todo caso, implementar el constructor e inicializar el objeto mediante éste, es una buena práctica, nosotros utilizaremos esta variante.

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. 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:

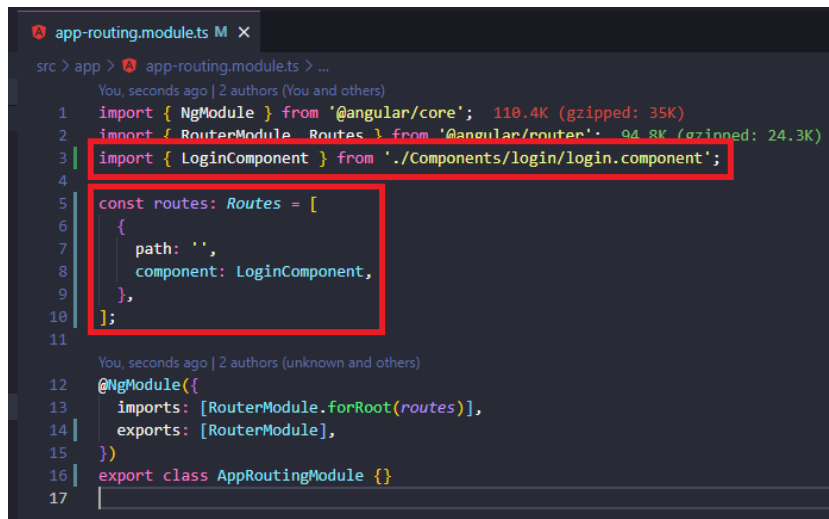
ng g c Components/login

```
C:\Projects\driven-forms-example>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 (272 bytes)
CREATE src/app/Components/login/login.component.scss (0 bytes)
UPDATE src/app/app.module.ts (544 bytes)
```

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 esté declarado en el **app.module.ts** para que lo podamos utilizar.

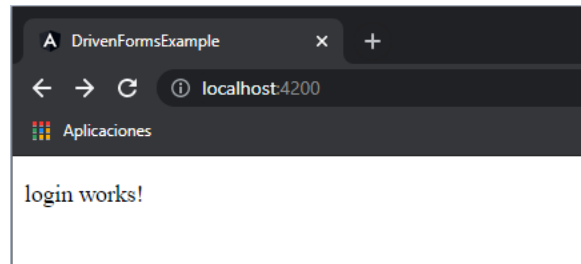
- **Paso 5:**

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**:



```
app-routing.module.ts M x
src > app > app-routing.module.ts > ...
You, seconds ago | 2 authors (You and others)
1 import { NgModule } from '@angular/core'; 110.4K (gzipped: 35K)
2 import { RouterModule, Routes } from '@angular/router'; 94.8K (gzipped: 24.3K)
3 import { LoginComponent } from './Components/login/login.component';
4
5 const routes: Routes = [
6   {
7     path: '',
8     component: LoginComponent,
9   },
10 ];
11
You, seconds ago | 2 authors (unknown and others)
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**:

```
login.component.ts U x
src > app > Components > login > login.component.ts > ...
1 import { Component, OnInit } from '@angular/core'; 110.6K (gzipped: 35.1K)
2 import { UserDTO } from 'src/app/Models/user.dto';
3
4 @Component({
5   selector: 'app-login',
6   templateUrl: './login.component.html',
7   styleUrls: ['./login.component.scss'],
8 })
9 export class LoginComponent implements OnInit {
10   user: UserDTO; (A)
11
12   constructor() {
13     this.user = new UserDTO('', ''); (B)
14   }
15
16   ngOnInit(): void {} (C)
17
18   checkLogin(): void {
19     console.log(
20       'User email --> ' +
21       this.user.email +
22       'User password --> ' +
23       this.user.password
24     );
25   }
26 }
```

A. Sección para declarar las variables

- Declaramos una variable pública **user** y le asignamos el tipo, en este caso nuestro modelo **UserDTO**.
- CLEAN CODE:** Siempre definiremos el tipo, evitaremos poner **ANY** en nuestro código.
- CLEAN CODE:** Si son variables públicas (accesibles desde la vista **.html**), no hace falta escribir la palabra reservada **public**, es implícito. Si son variables privadas entonces sí, escribiríamos la palabra reservada **private**.

B. Inicializamos la clase `user`.

- a. **CLEAN CODE:** En el constructor inicializamos las variables u objetos implicados en nuestro componente. Fijémonos que tenemos que inicializar la clase **UserDTO** mediante el constructor pasando por parámetro los valores “.

Si son inicializaciones de variables sencillas, se puede hacer en el mismo constructor. Si tuviéramos objetos complejos podríamos declarar un método privado dentro del constructor tipo:

i. `this.initializeData();`

- ii. E implementaremos el método anterior en la parte final del fichero, tal que así:

```
private initializeData(): void {  
    ...  
}
```

- iii. Ordenaremos primero todos los métodos o funciones públicas y luego los privados.

C. Normalmente en el **ngOnInit** tendremos código implementado, pero si se diera el caso de que no nos hiciera falta, eliminaremos el método y también eliminaremos el **implements OnInit** en la declaración de la clase. *Nota: Desde la versión 15 de Angular el CLI no genera el componente con la implementación del interfaz OnInit.*

D. Seguidamente, implementamos el método público **checkLogin** (accesible desde la vista) 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 email y la contraseña del usuario.

- a. **CLEAN CODE:** De igual manera que en las variables, los métodos o funciones públicas no hace falta utilizar la palabra reservada **public**. En cambio, si son métodos o funciones privadas si, escribiéramos delante la palabra reservada **private**.

b. **CLEAN CODE:**

- i. Si se trata de un método, la definición tiene que terminar en **void**:

1. Ejemplo: **checkLogin(): void { ... }**

- ii. Si se trata de una función, la definición tiene que terminar con el **tipo que retorna la función**, sea un tipo simple, objeto, promesa, ... Esto lo iremos viendo a medida que avancemos en la asignatura, ahora para que nos vayamos familiarizando, algunos ejemplos:

1. Ejemplo tipo simple:

```
isAdminProfile(): boolean {  
    ...  
    return (boolean variable)  
}
```

2. Ejemplo tipo objeto:

```
getUserByEmail(email: string): UserDTO {
```

```
...
```

```
return (UserDTO variable)
```

```
}
```

3. Ejemplo array de objetos:

```
getUsersByProfile(profileCode: string): UserDTO[] {
```

```
...
```

```
Return ( UserDTO array variable)
```

```
}
```

4. Ejemplo de petición al backend:

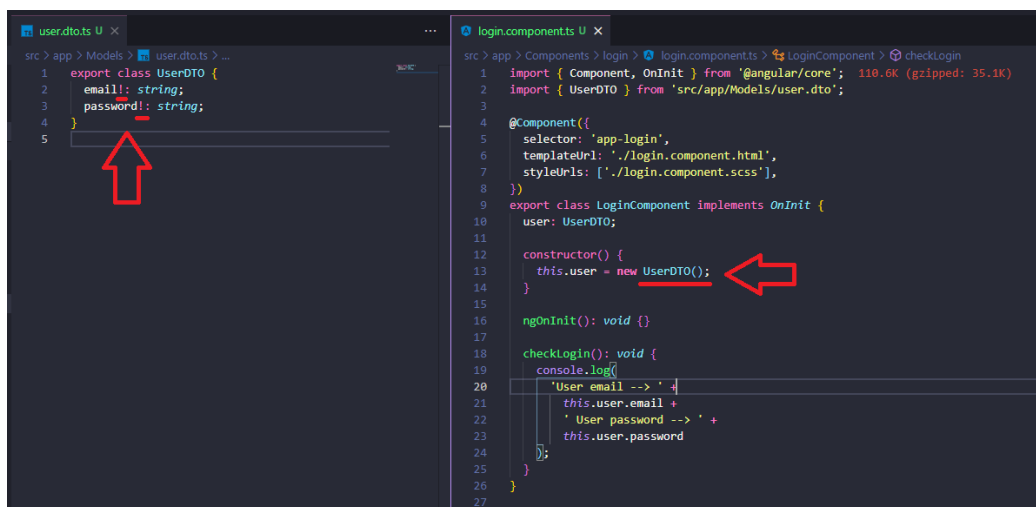
```
private async getUsers(): Promise<UserDTO[]> {
```

```
...
```

```
return await this.userService.getUsers();
```

```
}
```

Un apunte ...



Hemos comentado que al aplicar la configuración de inicialización de propiedades de manera estricta esto nos obliga a implementar el constructor y, por tanto, hacer uso de éste cuando utilicemos esta clase en algún componente para inicializarla.

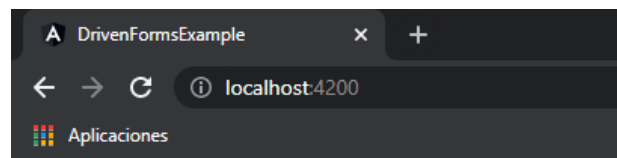
En la captura anterior ponemos el ejemplo de utilización de la marca **!**, así no estamos obligados a implementar el constructor. Sería otra opción.

• Paso 7:

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

```
login.component.html U X
src > app > Components > login > login.component.html > ...
1 <div>
2   <h1>Test Driven Forms App</h1>
3   <form #loginForm="ngForm" (ngSubmit)="checkLogin()">
4     <div>
5       <label for="email">Email: </label>
6       <input
7         type="email"
8         name="email"
9         [(ngModel)]="user.email"
10        #email="ngModel"
11        required
12        placeholder="Email..."
13      />
14      <span [hidden]="email.valid || email.pristine" style="color: red">
15        >Email required</span>
16    </div>
17    <div>
18      <label for="password">Password: </label>
19      <input
20        type="password"
21        name="password"
22        [(ngModel)]="user.password"
23        #password="ngModel"
24        required
25        placeholder="Password..."
26      />
27      <span [hidden]="password.valid || password.pristine" style="color: red">
28        >Password required</span>
29    </div>
30    <div>
31      <button type="submit" [disabled]="!loginForm.form.valid">Login</button>
32    </div>
33  </form>
34</div>
35
36
37
```

Resultado:



Test Driven Forms App

Email:

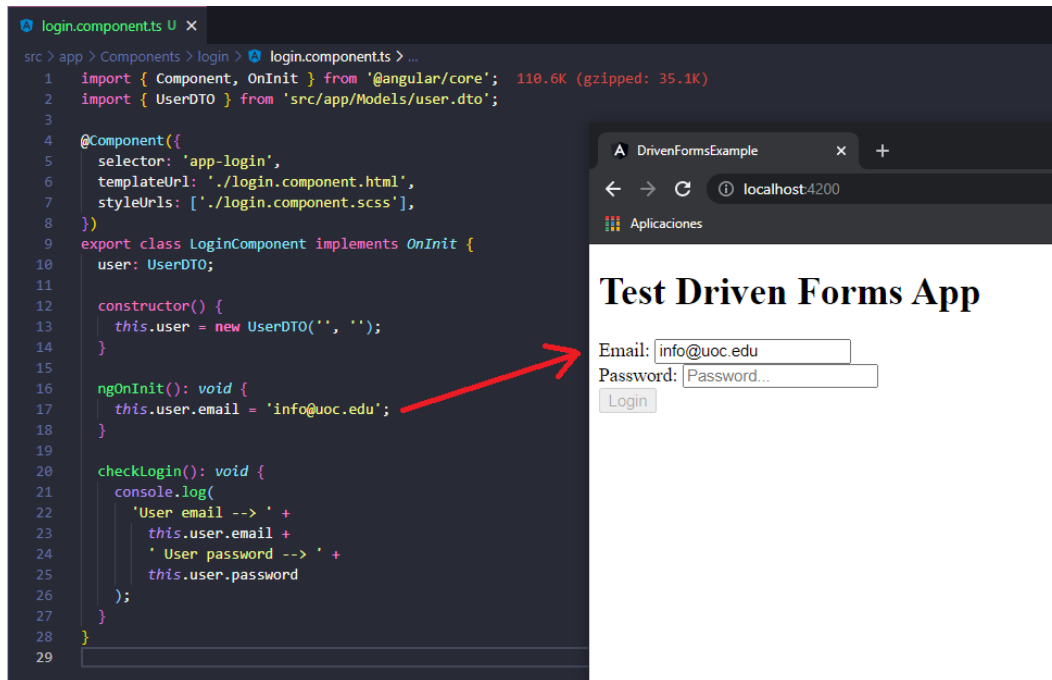
Password:

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 **email** y **password** del formulario se guardaran en nuestro modelo **user.email** y **user.password** respectivamente. Y si en

nuestro controlador asignamos directamente valores a nuestro modelo, se mostrarían en el formulario. Por ejemplo, si hiciéramos:



Asignamos el email **info@uoc.edu** a la variable **email** 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 **Email**.

Podríamos haber asignado este **email** directamente en el constructor si se tratara de una inicialización como tal. Realmente en el **ngOnInit** se modificaría el valor de la variable **email** si fuera un valor distinto al de la inicialización propiamente, siendo resultado de aplicar alguna lógica o de la carga de un dato de un servicio. Ahora simplemente es un caso de estudio.

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**, en nuestro caso, cada uno de los dos **inputs**. 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 inválido.

Resumimos los estados en la siguiente tabla:

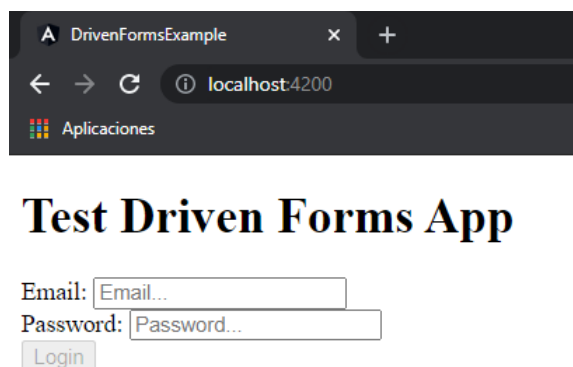
Estados de un campo		
Estado	Clase si el estado es cierto	Clase si el estado es falso
El control se ha visitado	ng-touched	ng-untouched
El valor del control ha cambiado	ng-dirty	ng-pristine
El valor del control es válido	ng-valid	ng-invalid

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 **#email="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 de la funcionalidad, este sería el estado inicial:



DrivenFormsExample x +

← → ↻ ⓘ localhost:4200

Aplicaciones

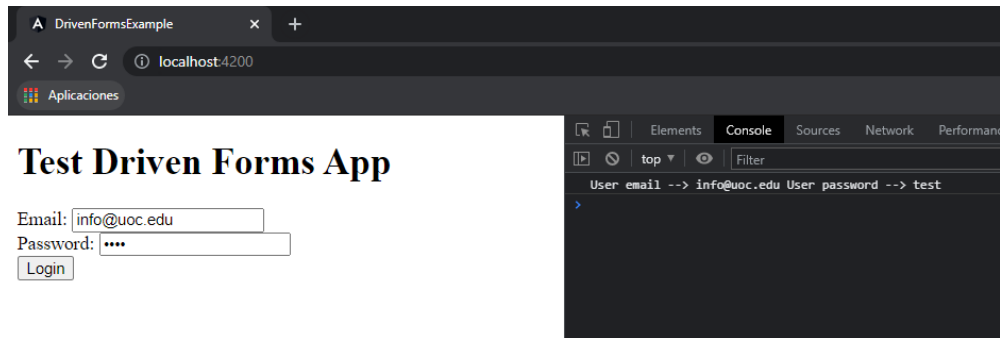
Test Driven Forms App

Email:

Password:

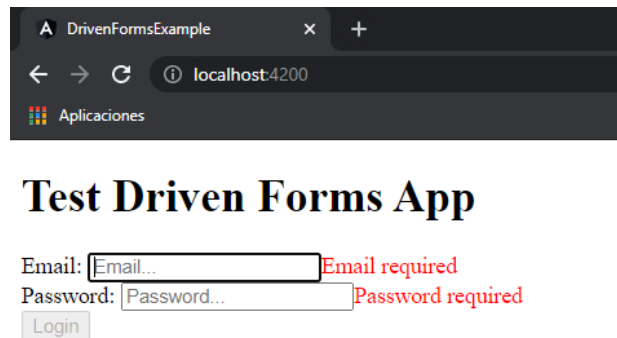
Login

Si introducimos un email 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**:



Intenta implementar el ejemplo primero sin utilizar la solución del campus.

Tienes este ejemplo implementado y subido al campus con el nombre:

driven-forms-example.zip

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, vamos a suponer que en lugar de hacer una autenticación necesitamos hacer un registro, con lo que te proponemos hacer las siguientes modificaciones para practicar, concretamente, te proponemos los siguientes pasos:

- Crea un componente nuevo **sign-in**.
- Crea una ruta nueva **/sign-in** que apunte al componente creado al paso anterior.
- Amplía el modelo **UserDTO** con los campos:
 - **name**: texto.
 - **surname1**: texto.
 - **surname2**: texto.
 - **alias**: texto.
 - **birthDate**: fecha.
- Crea el formulario de registro en el componente **sign-in**. Las validaciones que debe cumplir son las siguientes:
 - **Email**, **password**, **name**, **surname1**, **alias** y **birthDate** son campos obligatorios.
 - Los campos **name**, **surname1**, **surname2** y **alias** deben tener entre 5 y 25 caracteres.
 - La fecha debe tener un formato DD/MM/YYYY.
 - El **email** debe tener un formato de **email** válido.
- Muestra la nueva información por consola cuando se pulse el botón de **Join Now**. A diferencia del ejemplo anterior, en el que teníamos el botón de **submit** habilitado cuando el formulario era válido, vamos a hacer que se genere la validación del

formulario en el momento que pulsemos el botón. Es decir, el botón siempre estará habilitado, al pulsarlo se validará el formulario y si algún campo no cumple la validación se mostrará el mensaje en rojo determinado y si el formulario es correcto, lanzaremos la función de **joinNow** mostrando los diferentes campos por consola. De esta manera tendremos las dos versiones posibles de tratar el botón de **submit**.



“Comparte con tus compañer@s los resultados de esta actividad complementaria en el foro del aula.”

Intenta implementar el ejercicio primero sin utilizar la solución del campus.

Tienes este ejercicio implementado y subido al campus con el nombre:

driven-forms-example-extra.zip

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

5) 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 primer ejemplo, pero esta vez con el paradigma de formulario reactivo.

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

- **Paso 1:**

Creemos un nuevo proyecto ejecutando el siguiente comando desde la consola:

ng new << nombre_proyecto >>

```
C:\Projects>ng new reactive-forms-example
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS [ https://sass-lang.com/documentation/syntax#scss ]
```

Como comentamos en el ejemplo anterior de los formularios dirigidos por el modelo, escogeremos que nos añada el **routing** de Angular y utilizaremos el tipo de estilos **SCSS**.

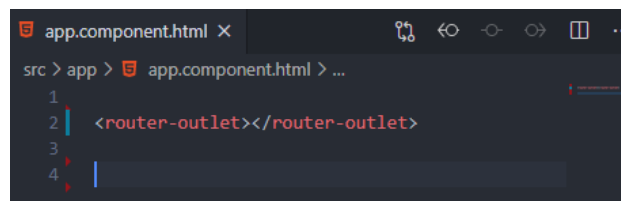
Ejecutamos el proyecto con la instrucción:

ng serve --open

```
C:\Projects\reactive-forms-example>ng serve --open
```

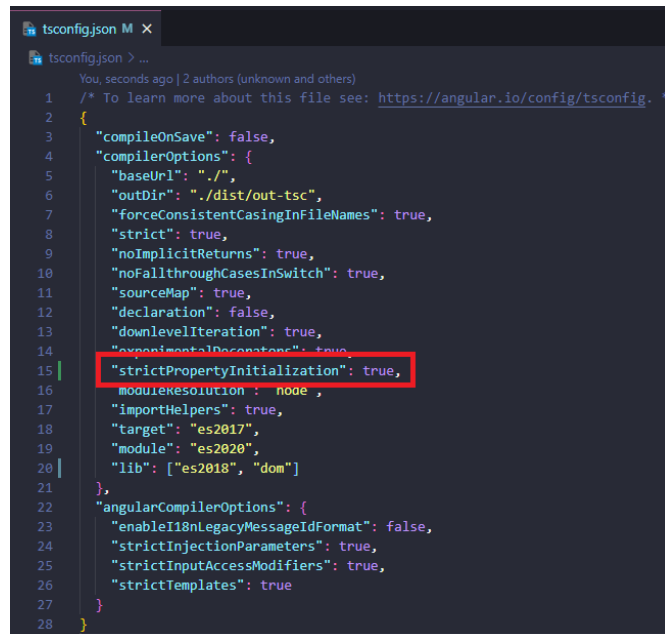
Y validamos que vemos la pantalla inicial del proyecto en el navegador.

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:



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

Podemos aprovechar este momento para aplicar la configuración de “inicialización de propiedades estricta de **TypeScript**”, concretamente añadiremos la siguiente línea en el fichero de configuración **tsconfig.json**:



```

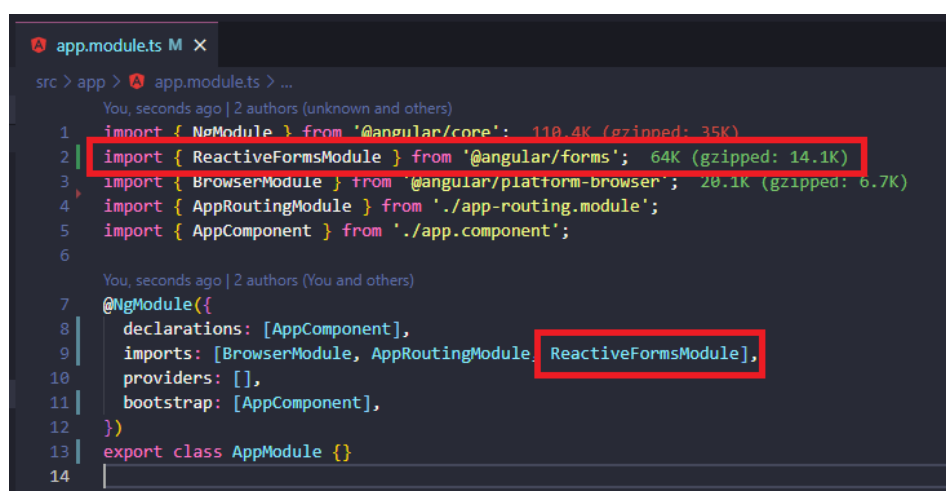
1  /* To learn more about this file see: https://angular.io/config/tsconfig. */
2  {
3    "compileOnSave": false,
4    "compilerOptions": {
5      "baseUrl": "./",
6      "outDir": "./dist/out-tsc",
7      "forceConsistentCasingInFileNames": true,
8      "strict": true,
9      "noImplicitReturns": true,
10     "noFallthroughCasesInSwitch": true,
11     "sourceMap": true,
12     "declaration": false,
13     "downlevelIteration": true,
14     "experimentalDecorators": true,
15     "strictPropertyInitialization": true,
16     "moduleResolution": "node",
17     "importHelpers": true,
18     "target": "es2017",
19     "module": "es2020",
20     "lib": ["es2018", "dom"]
21   },
22   "angularCompilerOptions": {
23     "enableI18nLegacyMessageIdFormat": false,
24     "strictInjectionParameters": true,
25     "strictInputAccessModifiers": true,
26     "strictTemplates": true
27   }
28 }

```

Vamos a programar en base a la configuración anterior, al igual que hemos aplicado estas pautas en el ejemplo anterior de formularios dirigidos por el modelo.

• Paso 2:

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



```

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

```

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

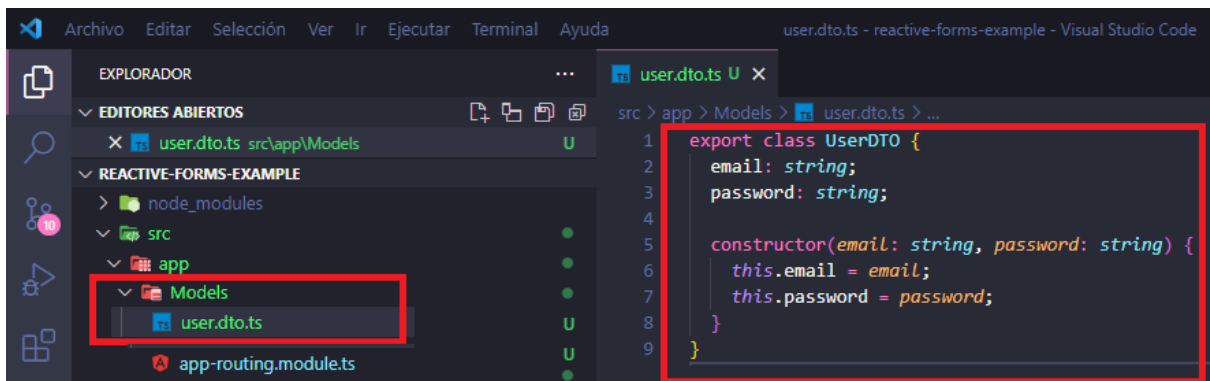
El **FormControl** lo utilizaremos para registrar un único campo de entrada del formulario. Por ejemplo, en nuestro caso que tendremos dos campos, **email** y **password**, tendremos dos

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 email y la contraseña del usuario, por lo tanto, crearemos un modelo para tal fin. Aquí podemos replicar lo que hicimos en el primer ejemplo.

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



The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar shows the project structure: 'src' > 'app' > 'Models'. The 'Models' folder is highlighted with a red box. In the center, the 'user.dto.ts' file is open in the editor. The code defines a class 'UserDTO' with two properties, 'email' and 'password', both of type 'string'. It also includes a constructor that takes 'email' and 'password' as arguments and assigns them to 'this.email' and 'this.password' respectively. The entire code block is highlighted with a red border.

```
1 export class UserDTO {  
2   email: string;  
3   password: string;  
4  
5   constructor(email: string, password: string) {  
6     this.email = email;  
7     this.password = password;  
8   }  
9 }
```

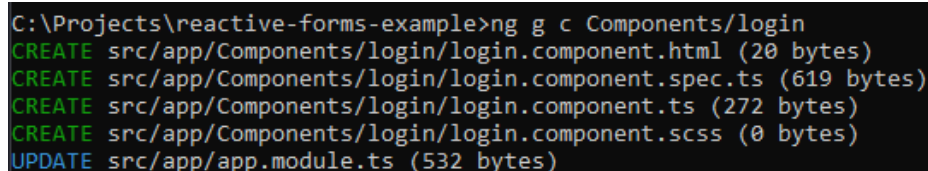
Fijémonos que como tenemos el modo estricto definido tenemos que implementar un constructor en la clase **UserDTO**. Tenemos la explicación más detallada en el ejemplo anterior, en el primer ejemplo de formulario dirigido por el modelo de este documento.

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. 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:

ng g c Components/login



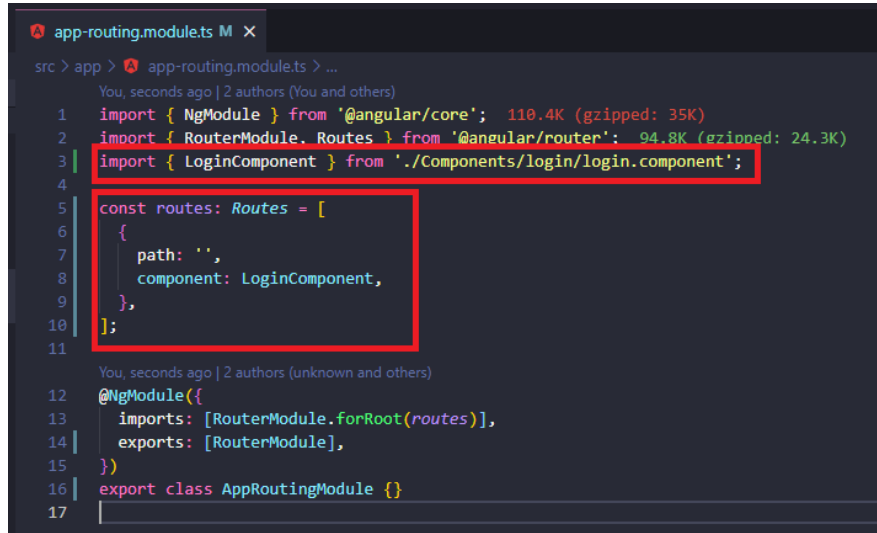
The terminal output shows the command 'ng g c Components/login' being executed. It lists the files created: 'login.component.html' (20 bytes), 'login.component.spec.ts' (619 bytes), 'login.component.ts' (272 bytes), and 'login.component.scss' (0 bytes). It also shows that 'app.module.ts' was updated (532 bytes).

```
C:\Projects\reactive-forms-example>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 (272 bytes)  
CREATE src/app/Components/login/login.component.scss (0 bytes)  
UPDATE src/app/app.module.ts (532 bytes)
```

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 esté declarado en el **app.module.ts** para que lo podamos utilizar.

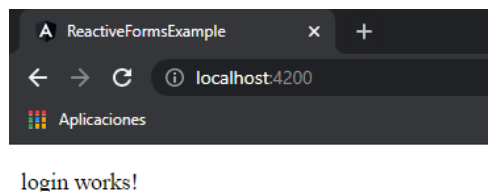
- **Paso 5:**

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**.



```
1 import { NgModule } from '@angular/core'; 110.4K (gzipped: 35K)
2 import { RouterModule, Routes } from '@angular/router'; 94.8K (gzipped: 24.3K)
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:

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

```
login.component.ts U X
src > app > Components > login > login.component.ts > ...
1  import { Component } from '@angular/core'; 110.4K (gzipped: 35K)
2  import { FormBuilder, FormControl, FormGroup } from '@angular/forms'; 64K (gzipped: 14.1K)
3  import { UserDTO } from 'src/app/Models/user.dto';
4
5  @Component({
6    selector: 'app-login',
7    templateUrl: './login.component.html',
8    styleUrls: ['./login.component.scss'],
9  })
10 export class LoginComponent {
11   user: UserDTO;
12   email: FormControl; (A)
13   password: FormControl;
14   loginForm: FormGroup;
15
16   constructor(private formBuilder: FormBuilder) { (B)
17     this.user = new UserDTO('', '');
18
19     this.email = new FormControl(this.user.email);
20
21     this.password = new FormControl(this.user.password); (C)
22
23     this.loginForm = this.formBuilder.group({
24       email: this.email,
25       password: this.password,
26     });
27   } (D)
28
29   checkLogin(): void {
30     this.user.email = this.email.value;
31     this.user.password = this.password.value;
32     console.log(
33       'User email --> ' +
34         this.user.email +
35         ' User password --> ' +
36         this.user.password
37     );
38   }
39 }
40
```

A. Sección para declarar las variables

- Declaramos una variable pública **user** y le asignamos el tipo, en este caso nuestro modelo **UserDTO**.
- Seguidamente, declaramos un **FormControl** por cada entrada de nuestro formulario, en este caso uno para el campo **email** 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.

Recordemos las buenas prácticas que comentamos en el primer ejemplo sobre las declaraciones de variables.

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

C. Inicializamos:

- El objeto **user** utilizando el constructor.
- Registramos los **FormControl** para los campos **email** y **password** y posteriormente los agrupamos dentro del grupo **loginForm**. Posteriormente en

la vista veremos que esta variable **loginForm** se la asignaremos a la directiva **[formGroup]** del formulario.

Recordemos las buenas prácticas que comentamos en el primer ejemplo sobre las inicializaciones de variables.

- D. Nótese que como no tenemos lógica en el **ngOnInit** no lo implementamos (si no hay código puede omitirse su implementación).
- E. 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.

Recordemos las buenas prácticas que comentamos en el primer ejemplo sobre las declaraciones y definiciones de métodos y funciones.

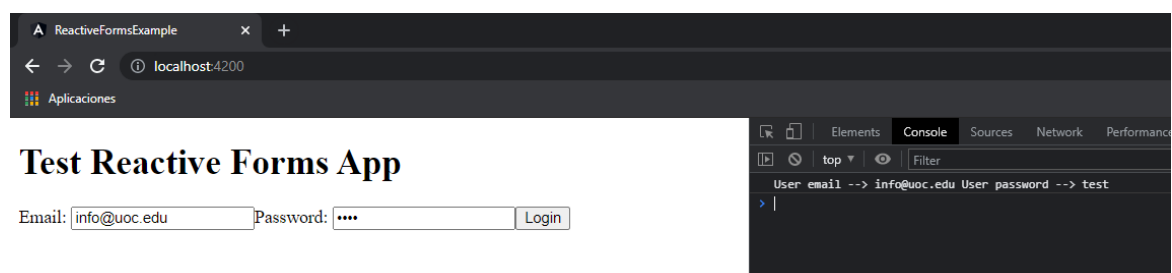
• Paso 7:

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

```
login.component.html U x
src > app > Components > login > login.component.html > ...
1 <div>
2 <h1>Test Reactive Forms App</h1>
3 <form [formGroup]="loginForm" (ngSubmit)="checkLogin()">
4 <label for="email">Email: </label>
5 <input type="email" [formControl]="email" />
6
7 <label for="password">Password: </label>
8 <input type="password" [formControl]="password" />
9
10 <button type="submit" [disabled]="!loginForm.valid">Login</button>
11 </form>
12 </div>
13
```

Si bien en el controlador hemos registrado los dos controles (**email** 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**.

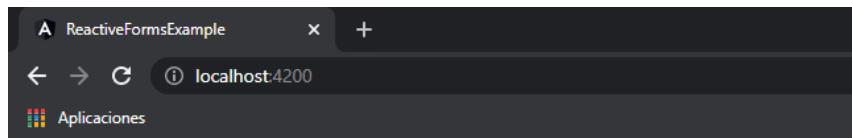
Tal y como tenemos la implementación hasta este momento, podemos ver la ejecución en el navegador. Podemos observar que, si introducimos valores en los dos campos y pulsamos el botón de **login**, se mostrará por consola los valores insertados.



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:

```
constructor(private FormBuilder: FormBuilder) {  
  this.user = new UserDTO('info@uoc.edu', '');  
  
  this.email = new FormControl(this.user.email);  
  
  this.password = new FormControl(this.user.password);  
  
  this.loginForm = this.formBuilder.group({  
    email: this.email,  
    password: this.password,  
  });  
}
```

Al refrescarse el navegador veríamos:



Test Reactive Forms App

Email: Password:

También podemos observar que el botón de **login** nunca está 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:

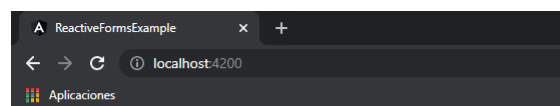
```
login.component.ts x
src > app > Components > login > login.component.ts > ...
1 import { Component } from '@angular/core'; 110.4K (gzipped: 35K)
2 import {
3   FormBuilder,
4   FormControl,
5   FormGroup,
6   Validators,
7 } from '@angular/forms'; 64.4K (gzipped: 14.2K)
8 import { UserDTO } from 'src/app/Models/user.dto';
9
10 @Component({
11   selector: 'app-login',
12   templateUrl: './login.component.html',
13   styleUrls: ['./login.component.scss'],
14 })
15 export class LoginComponent {
16   user: UserDTO;
17   email: FormControl;
18   password: FormControl;
19   loginForm: FormGroup;
20
21   constructor(private formBuilder: FormBuilder) {
22     this.user = new UserDTO('', '');
23
24     this.email = new FormControl(this.user.email, Validators.required);
25
26     this.password = new FormControl(this.user.password, Validators.required);
27
28     this.loginForm = this.formBuilder.group({
29       email: this.email,
30       password: this.password,
31     });
32   }
33
34   checkLogin(): void {
35     this.user.email = this.email.value;
36     this.user.password = this.password.value;
37     console.log(
38       'User email --> ' +
39         this.user.email +
40         ' User password --> ' +
41         this.user.password
42     );
43   }
44 }
```


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.html U X
src > app > Components > login > login.component.html > ...
1 <div>
2 <h1>Test Reactive Forms App</h1>
3 <form [formGroup]="loginForm" (ngSubmit)="checkLogin()">
4 <label for="email">Email: </label>
5 <input type="email" [formControl]="email" />
6
7 <div *ngIf="email.errors">
8 <span style="color: red" *ngIf="email.invalid && email?.errors?.required"
9 >Email is required</span>
10 </div>
11
12 <label for="password">Password: </label>
13 <input type="password" [formControl]="password" />
14
15 <div *ngIf="password.errors">
16 <span
17 style="color: red"
18 *ngIf="password.invalid && password?.errors?.required"
19 >Password is required</span>
20 </div>
21
22 <button type="submit" [disabled]="!loginForm.valid">Login</button>
23 </form>
24 </div>
25
26
```

Añadimos estos dos bloques a la vista para que si el campo no está informado muestre un mensaje de error en rojo.



Test Reactive Forms App

Email:
 Email is required
 Password:
 Password is required
 Login

Ahora si ejecutamos podemos ver:

Inicialmente nos indica los mensajes de error, que en este caso sería correcto. Si insertamos 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 U X
src > app > Components > login > login.component.ts > ...
1  import { Component } from '@angular/core'; 110.4K (gzipped: 35K)
2  import {
3      FormBuilder,
4      FormControl,
5      FormGroup,
6      Validators,
7  } from '@angular/forms'; 64.4K (gzipped: 14.2K)
8  import { UserDTO } from 'src/app/Models/user.dto';
9
10 @Component({
11     selector: 'app-login',
12     templateUrl: './login.component.html',
13     styleUrls: ['./login.component.scss'],
14 })
15 export class LoginComponent {
16     user: UserDTO;
17     email: FormControl;
18     password: FormControl;
19     loginForm: FormGroup;
20
21     constructor(private formBuilder: FormBuilder) {
22         this.user = new UserDTO('', '');
23
24         this.email = new FormControl(this.user.email, Validators.required);
25
26         this.password = new FormControl(this.user.password, [
27             Validators.required,
28             Validators.minLength(8),
29         ]);
30
31         this.loginForm = this.formBuilder.group({
32             email: this.email,
33             password: this.password,
34         });
35     }
36
37     checkLogin(): void {
38         this.user.email = this.email.value;
39         this.user.password = this.password.value;
40         console.log(
41             'User email --> ' +
42             this.user.email +
43             ' User password --> ' +
44             this.user.password
45         );
46     }
47 }
```

Vamos a aprovechar para añadir alguna validación más, en este caso, que el campo **password** tenga como mínimo 8 caracteres. 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.html U x
src > app > Components > login > login.component.html > ...
1 <div>
2 <h1>Test Reactive Forms App</h1>
3 <form [formGroup]="loginForm" (ngSubmit)="checkLogin()">
4   <label for="email">Email: </label>
5   <input type="email" [formControl]="email" />
6
7   <!--
8   <div *ngIf="email.errors">
9     <span style="color: red" *ngIf="email.invalid && email?.errors?.required">
10       Email is required</span>
11   </div>
12   -->
13
14   <label for="password">Password: </label>
15   <input type="password" [formControl]="password" />
16
17   <!--
18   <div *ngIf="password.errors">
19     <span
20       style="color: red"
21       *ngIf="password.invalid && password?.errors?.required"
22     >Password is required</span>
23   </div>
24   -->
25
26   <button type="submit" [disabled]="!loginForm.valid">Login</button>
27 </form>
28 </div>
29
30
31
32 <div *ngIf="email.errors">
33   <span
34     style="color: red"
35     *ngIf="email.errors && (email.touched || email.dirty)"
36   >
37     <span *ngIf="email.errors.required">Email is required</span>
38   </span>
39 </div>
40
41 <div *ngIf="password.errors">
42   <span
43     style="color: red"
44     *ngIf="password.errors && (password.touched || password.dirty)"
45   >
46     <span *ngIf="password.errors.required">Password is required</span>
47     <span *ngIf="password.errors.minlength">
48       Password must be greater tha 8 characters</span>
49   </span>
50 </div>
51 </div>
```

Si ahora ejecutamos la aplicación observaremos el siguiente comportamiento:

- De inicio no nos muestra mensajes de error y tendremos el botón de **login** deshabilitado.
- Si empezamos a escribir en el campo **password** nos aparecerá un mensaje de que el campo debe tener al menos 8 caracteres, si superamos dicho número el mensaje desaparecerá.
- Si quitamos lo que estábamos escribiendo, aparecerá el mensaje de campo requerido, tanto para el **email** como para el **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.

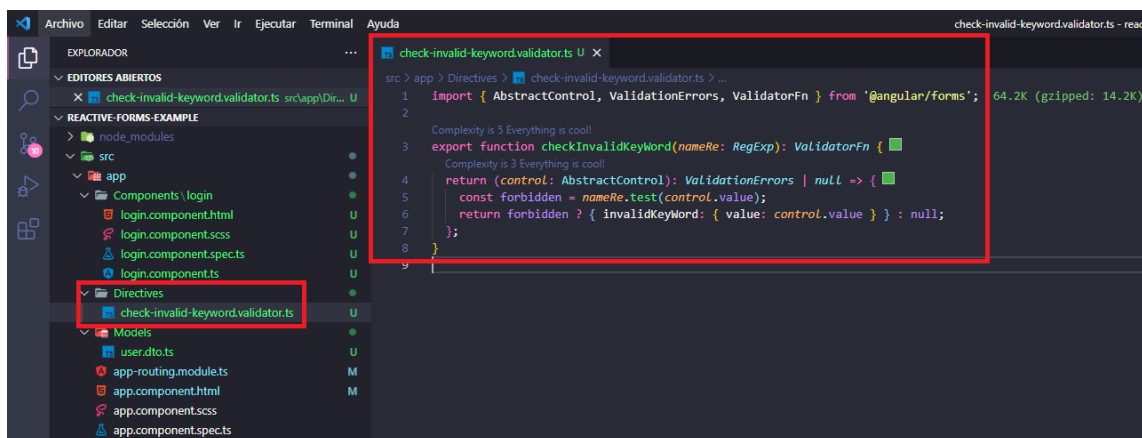
6) Ampliación validaciones formularios reactivos

Aunque la clase **Validators** permite disponer de un conjunto de validadores de base, a veces requerimos construir nuestros propios validadores (por ejemplo, que dos campos coinciden para asegurarnos que cuando pedimos repetir la contraseña en dos inputs sean iguales, 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 **email** se inserte el mail **info@uoc.edu**

Primero creamos una directiva que podamos reutilizar en los inputs que podamos tener en nuestro proyecto, de manera que le podamos pasar por parámetro la palabra reservada y que nos devuelva si el input determinado la contiene o no.

Podemos crear un directorio **Directives** y dentro crearnos el fichero **check-reserved-keyword.validator.ts** con el código siguiente:



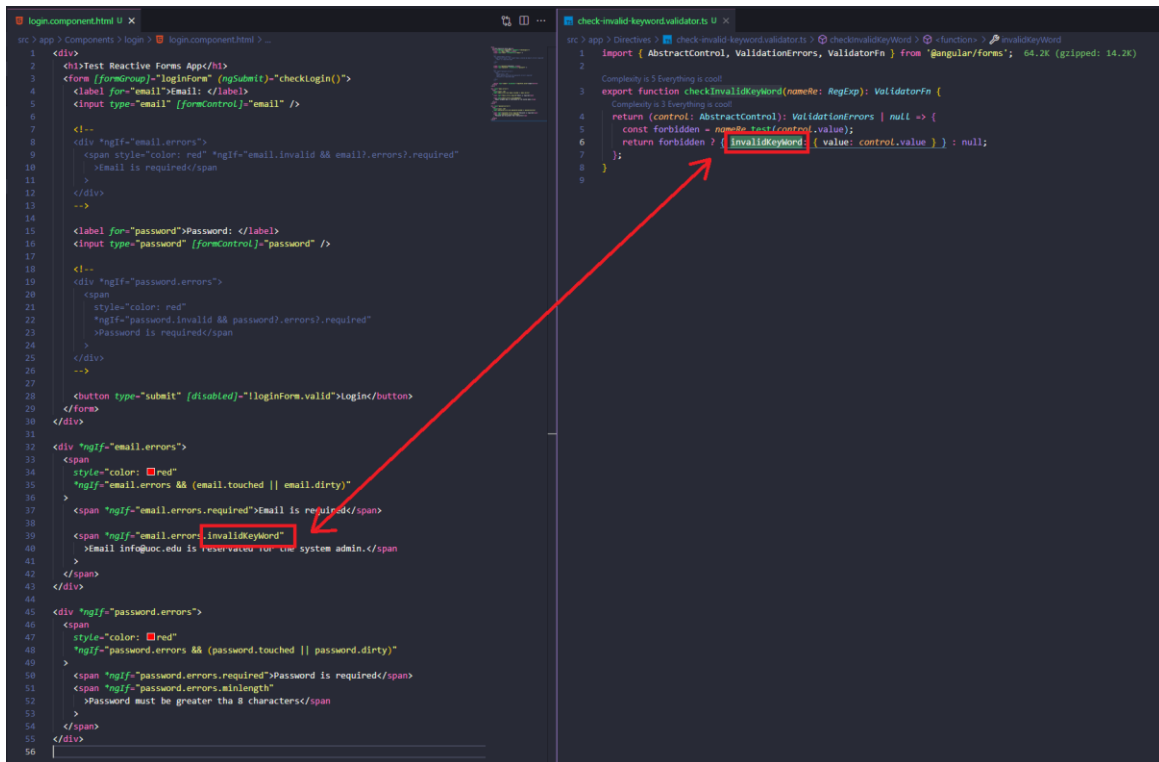
```
src > app > Directives > check-invalid-keyword.validator.ts > ...
1  import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms';
2
3  export function checkInvalidKeyword(nameRe: RegExp): ValidatorFn {
4      return (control: AbstractControl): ValidationErrors | null => {
5          const forbidden = nameRe.test(control.value);
6          return forbidden ? { invalidKeyword: { value: control.value } } : null;
7      };
8  }
```

Este código no hace falta entenderlo al 100%, pero a grandes rasgos lo que hace es, dada una expresión regular pasada por parámetro (posteriormente en el controlador veremos que le pasamos **/info@uoc.edu/**) guarda en la constante **forbidden** cierto o falso en función de si el valor del input del formulario es igual o no a la palabra prohibida pasada por parámetro. Si es igual devuelve la propiedad de error **invalidKeyword** para utilizarla en la vista y si no es igual y, por lo tanto, pasaría la validación, devuelve **null** ya que no tendríamos un error como tal.

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

```
login.component.ts U x
src > app > Components > login > login.component.ts > ...
1 import { Component } from '@angular/core'; 110.4K (gzipped: 35K)
2 import {
3   FormBuilder,
4   FormControl,
5   FormGroup,
6   Validators,
7 } from '@angular/forms'; 64.4K (gzipped: 14.2K)
8 import { checkInvalidKeyWord } from 'src/app/Directives/check-invalid-keyword.validator';
9 import { UserDTO } from 'src/app/Models/user.dto';
10
11 @Component({
12   selector: 'app-login',
13   templateUrl: './login.component.html',
14   styleUrls: ['./login.component.scss'],
15 })
16 export class LoginComponent {
17   user: UserDTO;
18   email: FormControl;
19   password: FormControl;
20   loginForm: FormGroup;
21
22   constructor(private formBuilder: FormBuilder) {
23     this.user = new UserDTO('', '');
24
25     this.email = new FormControl(this.user.email, [
26       Validators.required,
27       checkInvalidKeyWord(/info@uoc.edu/),
28     ]);
29
30     this.password = new FormControl(this.user.password, [
31       Validators.required,
32       Validators.minLength(8),
33     ]);
34
35     this.loginForm = this.formBuilder.group({
36       email: this.email,
37       password: this.password,
38     });
39   }
40
41   checkLogin(): void {
42     this.user.email = this.email.value;
43     this.user.password = this.password.value;
44     console.log(
45       'User email --> ' +
46       this.user.email +
47       ' User password --> ' +
48       this.user.password
49     );
50   }
51 }
```

Podemos ejecutar la aplicación y ver que el comportamiento es el esperado. Si insertamos el email **info@uoc.edu** en el campo **email** el botón de **login** se deshabilitará.



```

login.component.html
1 <div>
2 <!-- Test Reactive Forms App -->
3 <form [formGroup]="loginForm" (ngSubmit)="checkLogin()">
4   <label for="email">Email: </label>
5   <input type="email" [formControl]="email" />
6
7   <!--
8   <div *ngIf="email.errors">
9     <span style="color: red;" *ngIf="email.invalid && email.errors?.required">
10       Email is required</span>
11   </div>
12   -->
13
14   <label for="password">Password: </label>
15   <input type="password" [formControl]="password" />
16
17   <!--
18   <div *ngIf="password.errors">
19     <span style="color: red;"
20       *ngIf="password.invalid && password.errors?.required">
21       Password is required</span>
22   </div>
23   -->
24
25   <button type="submit" [disabled]="!loginForm.valid">login</button>
26 </form>
27
28 <div *ngIf="email.errors">
29   <span style="color: red;"
30     *ngIf="email.errors && (email.touched || email.dirty)"
31     *ngIf="email.errors.required">Email is required</span>
32   <span *ngIf="email.errors.invalidKeyword"
33     *ngIf="email.errors.invalidKeyword">Email info@uoc.edu is reserved for the system admin.</span>
34 </div>
35
36 <div *ngIf="password.errors">
37   <span style="color: red;"
38     *ngIf="password.errors && (password.touched || password.dirty)"
39     *ngIf="password.errors.required">Password is required</span>
40   <span *ngIf="password.errors.minLength">
41     Password must be greater than 8 characters</span>
42 </div>
43 </div>
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
25
```

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/>).

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 que realices la actividad complementaria anterior de los formularios dirigidos por el modelo donde se pedía ampliar el modelo **userDTO** y hacer un componente de registro, pero esta vez con formularios reactivos.

Recordamos los requisitos:

- Crea un componente nuevo **sign-in**
- Crea una ruta nueva **/sign-in** que apunte al componente creado al paso anterior
- Amplía el modelo **UserDTO** con los campos:
 - **name**: texto.
 - **surname1**: texto.
 - **surname2**: texto.
 - **alias**: texto.
 - **birthDate**: fecha.
- Crea el formulario de registro en el componente **sign-in**. Las validaciones que debe cumplir son las siguientes:
 - **Email**, **password**, **name**, **surname1**, **alias** y **birthDate** son campos obligatorios.

- Los campos **name**, **surname1**, **surname2** y **alias** deben tener entre 5 y 25 caracteres.
 - La fecha debe tener un formato DD/MM/YYYY.
 - El **email** debe tener un formato de **email** valido y no puede ser **info@uoc.educd**
- Muestra la nueva información por consola cuando se pulse el botón de **Join Now**. A diferencia del ejemplo anterior, en el que teníamos el botón de **submit** habilitado cuando el formulario era válido, vamos a hacer que se genere la validación del formulario en el momento que pulsemos el botón. Es decir, el botón siempre estará habilitado, al pulsarlo se validará el formulario y si algún campo no cumple la validación se mostrará el mensaje en rojo determinado y si el formulario es correcto, lanzaremos la función de **joinNow** mostrando los diferentes campos por consola. De esta manera tendremos las dos versiones posibles de tratar el botón de **submit**.



“Comparte con tus compañer@s los resultados de esta actividad complementaria en el foro del aula.”

Intenta implementar el ejercicio primero sin utilizar la solución del campus.

Tienes este ejercicio implementado y subido al campus con el nombre:

reactive-forms-example-extra.zip

7) Ejemplo MessageApp (con backend real)

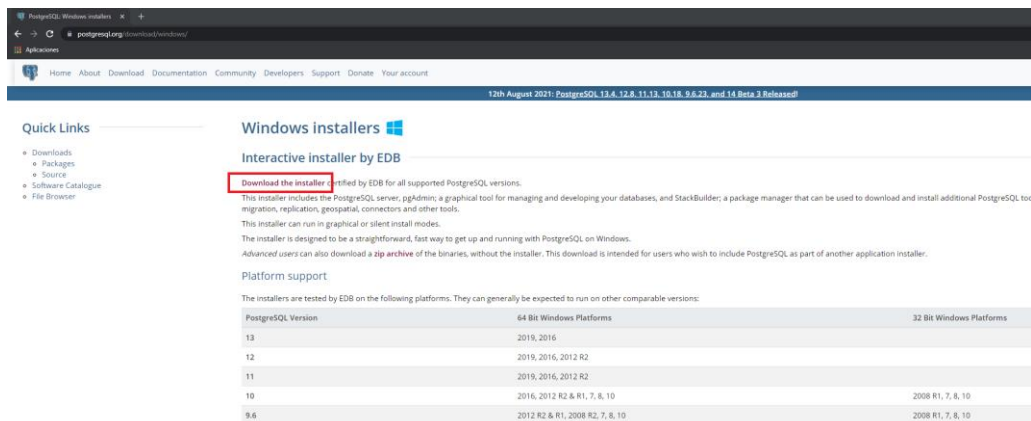
Vamos a hacer otro ejemplo para practicar con los formularios reactivos pero esta vez utilizando un entorno completo, es decir, implementaremos un **frontend** para hacer las operaciones básicas de alta, baja, actualizar y eliminar una entidad con un **backend** real. Este **backend** se conecta con una base de datos **PostgreSQL**. La parte de **backend** os la damos hecha, por lo tanto, lo primero que haremos es configurar el entorno para levantar el **backend** y consumiremos los **endpoints** con el **Postman**. Cuando validemos que todo nos funciona, implementaremos el **frontend**. **Configuremos el entorno:**

- **Configuración base de datos:**

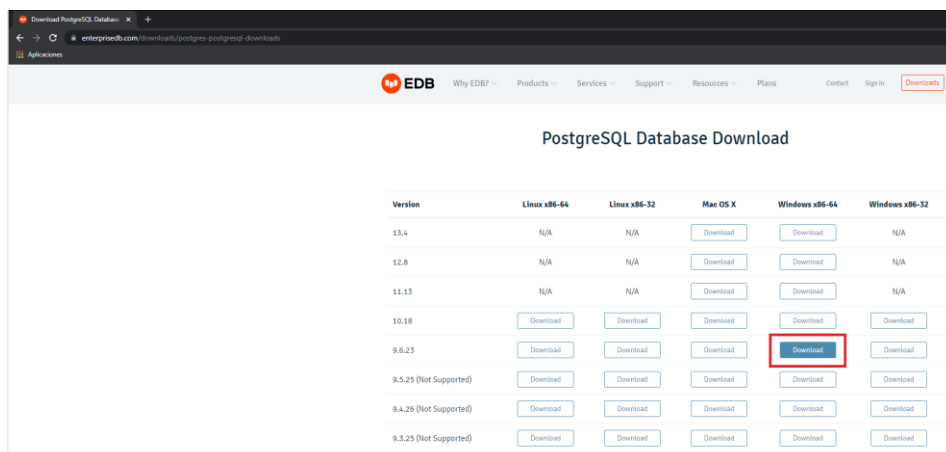
Como gestor de base de datos utilizaremos **PostgreSQL**. Por lo tanto, iremos a la página oficial:

<https://www.postgresql.org/download/>

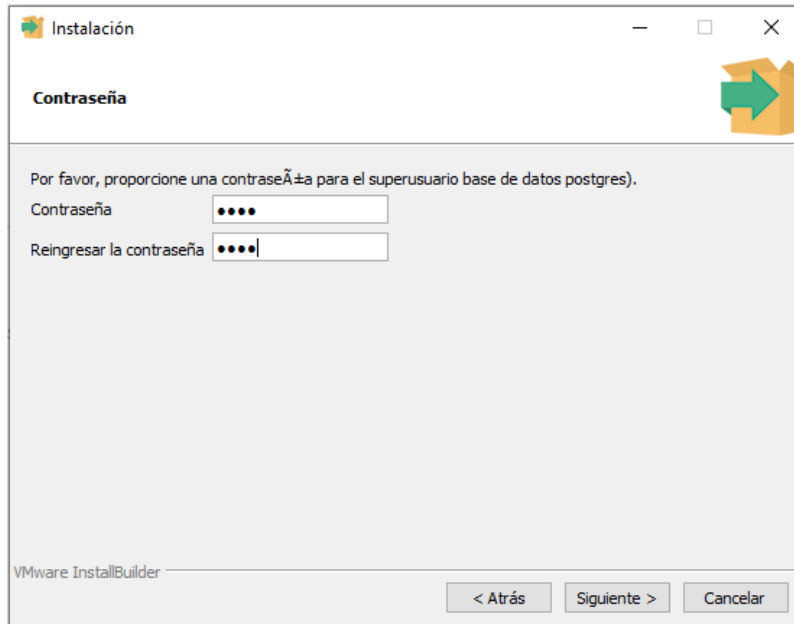
y nos bajaremos la versión acorde a nuestro entorno. En un entorno Windows de 64 bits recomendaría la versión 9.6, para ello haríamos clic en **Download the installer**:



Y nos bajamos la versión 9.6.23:

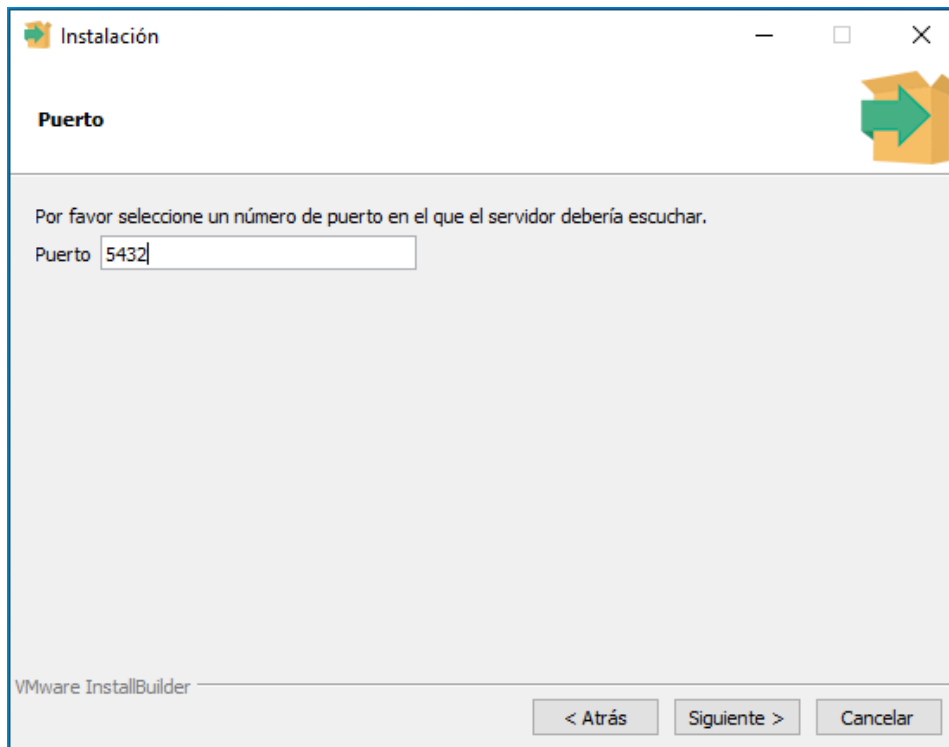


Esto nos descargará un instalador, básicamente sería abrirlo y seguir los pasos. Lo más destacado para tener en cuenta sería:

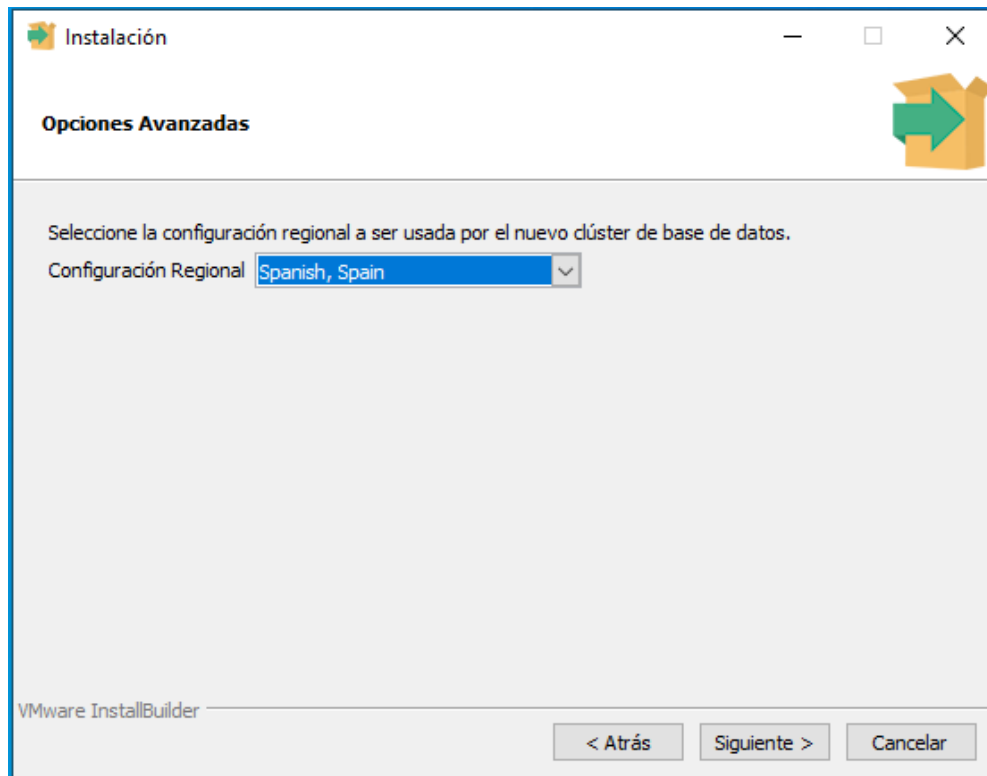


Indicar una contraseña para nuestro usuario **postgres**, esta contraseña la tendremos que recordar para posteriores desarrollos, para indicar en cadenas de conexión y demás, aquí para un entorno local se suele poner **root**, pero cada un@ el que quiera.

El puerto por defecto que se utiliza con **PostgreSQL** suele ser:



Y en cuanto a idioma:



Al final de la instalación nos dirá de arrancar el **Stack Builder**, le podemos decir que no ya que ya tenemos instalado lo necesario. Con el **Stack Builder** entre otras cosas podríamos instalar extensiones para nuestra base de datos, por ejemplo, la extensión **postGIS** que nos ayuda a trabajar con coordenadas y mapas, pero para nuestro caso, no nos haría falta instalar nada más.

Una vez terminada la instalación, podremos acceder desde el menú de Windows al programa **pgAdmin 4**, desde él podremos gestionar nuestras bases de datos y lanzar consultas **SQL**. Lo iremos viendo poco a poco.

Cuando abramos el **pgAdmin 4** tendremos que introducir la contraseña que hemos puesto durante la instalación.

- **Configuración framework backend:**

Vale, una vez instalado el motor de base de datos, vamos a instalar el **framework** con el que está desarrollado el **backend**. Para implementar el **backend** he escogido el **framework Nestjs**.

A grandes rasgos, es un **Node.js** pero siguiendo la arquitectura de **Angular**, con lo que resulta muy cómodo implementar una **API Rest** para el/la que conozca **Angular**.

Aquí la web del **framework** por si queréis revisarlo:

<https://nestjs.com/>

En todo caso, para vosotr@s el **backend** es transparente, en el sentido que tendréis unos **endpoints** a consumir desde el **frontend**, no “tenéis” que saber cómo está implementado el **backend** como tal. Si que tendréis que saber la especificación de cada **endpoint**, pero esto os lo daremos nosotros.

Para poder utilizar este **backend** tenéis que instalaros el paquete de **nestjs**, para ello solo necesitáis lanzar la siguiente instrucción desde la consola:

```
npm i -g @nestjs/cli
```

Una vez instalado el **framework**, solo tendríais que bajaros la carpeta **blog-api.zip** que está en el apartado del tema 1 en el campus, descomprimirla, y en la raíz del proyecto hacer un **npm install**, como si de un proyecto en Angular se tratara:

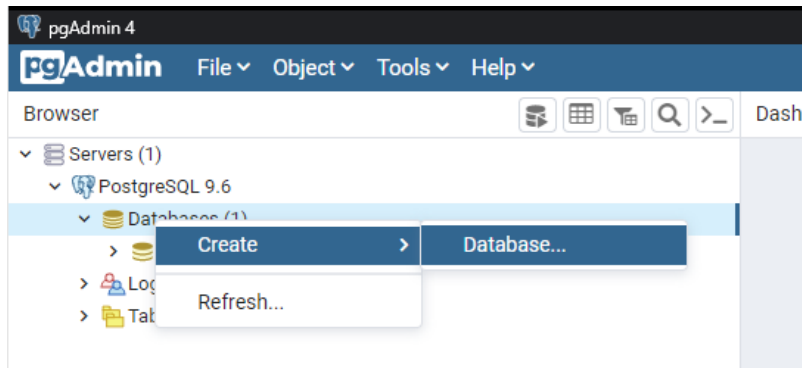
```
C:\Projects\blog-api>npm install
```

Ahora sólo nos quedaría “levantar” la api con la siguiente instrucción:

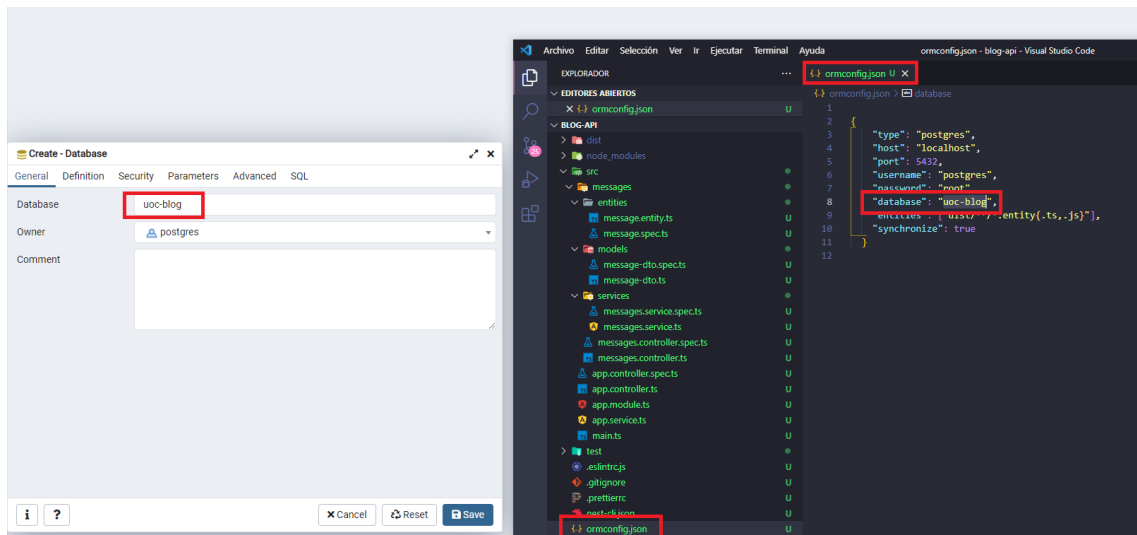
```
C:\Projects\blog-api>npm run start
```

Pero cuidado, antes tenemos que crear la base de datos para que al levantar la api por primera vez genere las tablas y todo lo necesario.

Para crear una base de datos solo necesitamos abrir el **pgAdmin 4** y:



Botón derecho sobre **Databases** -> **Create** -> **Database** y nombramos la base de datos como **uoc-blog**:



Es importante que sea **uoc-blog** ya que es la que está definida en el fichero de configuración **ormconfig.json** del proyecto **blog-api**. Podéis cambiar el nombre, no pasaría nada, pero tened en cuenta que la configuración del servidor de la base de datos está en este fichero.

Ahora sí, si hacemos:

```
C:\Projects\blog-api>npm run start
```

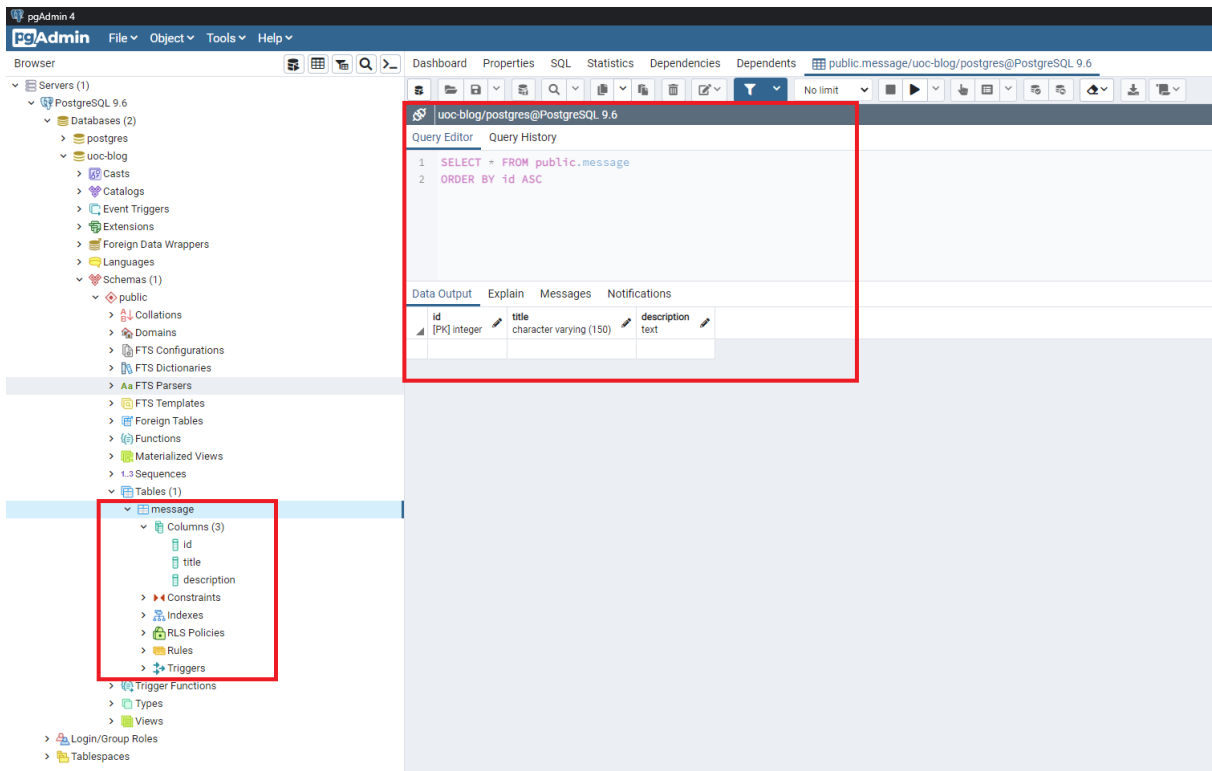
Levantamos la API y veremos la salida:

```
C:\Projects\blog-api>npm run start

> blog-api@0.0.1 start C:\Projects\blog-api
> nest start

[Nest] 15640 - 12/09/2021 16:07:16 LOG [NestFactory] Starting Nest application...
[Nest] 15640 - 12/09/2021 16:07:16 LOG [InstanceLoader] TypeOrmModule dependencies initialized +64ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [InstanceLoader] TypeOrmModule dependencies initialized +57ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [InstanceLoader] TypeOrmModule dependencies initialized +0ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [InstanceLoader] AppModule dependencies initialized +1ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [RoutesResolver] AppController {/}: +4ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [RouterExplorer] Mapped {/, GET} route +2ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [RoutesResolver] MessagesController {/messages}: +0ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [RouterExplorer] Mapped {/messages, GET} route +0ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [RouterExplorer] Mapped {/messages/:id, GET} route +1ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [RouterExplorer] Mapped {/messages, POST} route +0ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [RouterExplorer] Mapped {/messages/:id, PUT} route +1ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [RouterExplorer] Mapped {/messages/:id, DELETE} route +0ms
[Nest] 15640 - 12/09/2021 16:07:16 LOG [NestApplication] Nest application successfully started +2ms
```

Es decir, tenemos la api levantada, podemos ver los diferentes **endpoints** que tenemos disponibles (**crud** de la entidad **message**) y además, si vamos al **pgAdmin 4** podemos ver que nos ha creado la tabla necesaria:



Vale, tenemos el **backend** preparado.

Antes de ponernos a implementar el **frontend** para hacer el CRUD de la entidad **message**, vamos a ver una buena práctica que se suele utilizar en un entorno real, consumir el **frontend** sin picar una línea de código. Esto lo podemos hacer con el **Postman**.

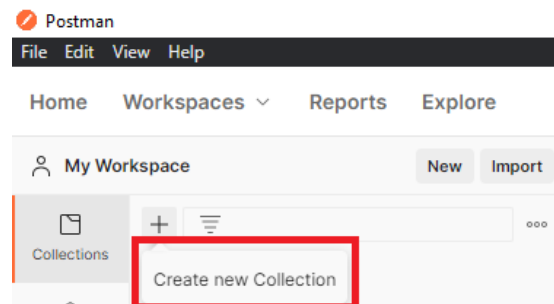
El **Postman** lo que nos da es que podemos consumir los **endpoints** y familiarizarnos con cada llamada, pasando los parámetros necesarios, revisando la respuesta de la llamada, ... en definitiva, es una herramienta muy útil para probar **apis**. La página oficial es la siguiente:

<https://www.postman.com/>

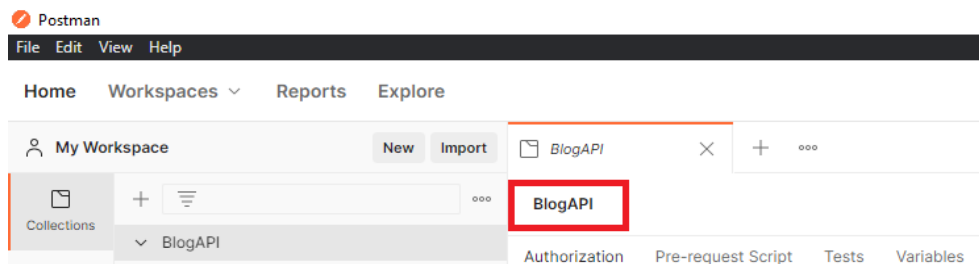
<https://www.postman.com/downloads/>

La idea sería que si no lo tenéis instalado os lo instaléis. En principio la instalación es sencilla. A continuación, os mostramos cómo “configurar” las llamadas que tenemos disponibles en la api que acabamos de levantar.

Para tener las llamadas organizadas en el **Postman**, lo ideal es agrupar las llamadas en colecciones, en nuestro caso, podríamos hacer lo siguiente:

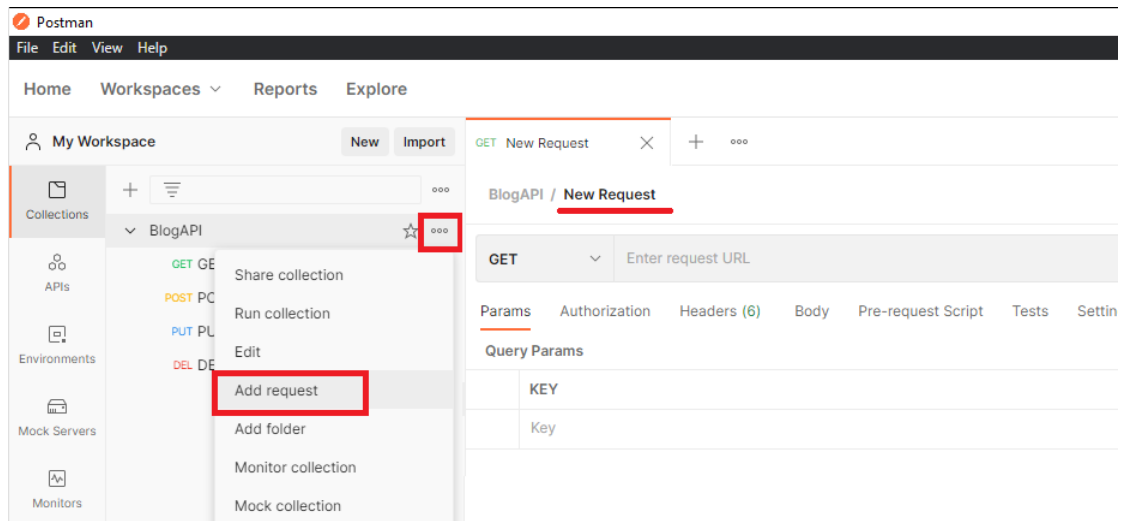


Y le damos un nombre:



En plan, agrupar todas las llamadas de este proyecto bajo la colección **BlogAPI**. Es simplemente una manera de tener la información ordenada ya que, en un entorno real de trabajo, seguramente tendremos varias **apis** a consumir. ¡Cuanto más organizado lo tengamos, mejor!

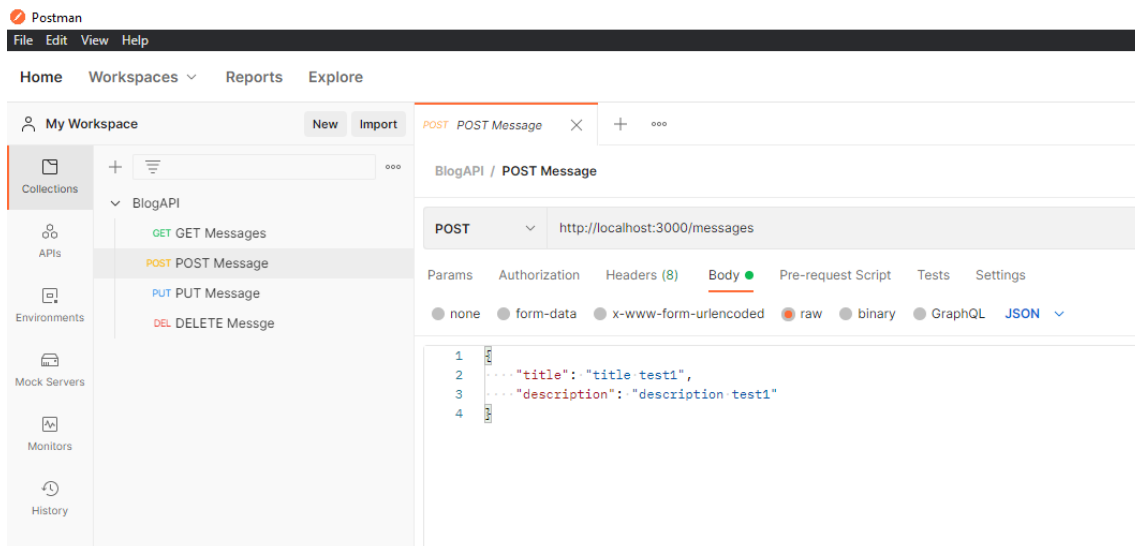
Sería cuestión de ir añadiendo las diferentes llamadas o **endpoints** a nuestra colección:



En nuestro caso tendremos 5 llamadas típicas de un **CRUD**:

- Llamada **POST** para dar de alta un mensaje nuevo, deberemos pasar en el cuerpo de la llamada un título y una descripción:

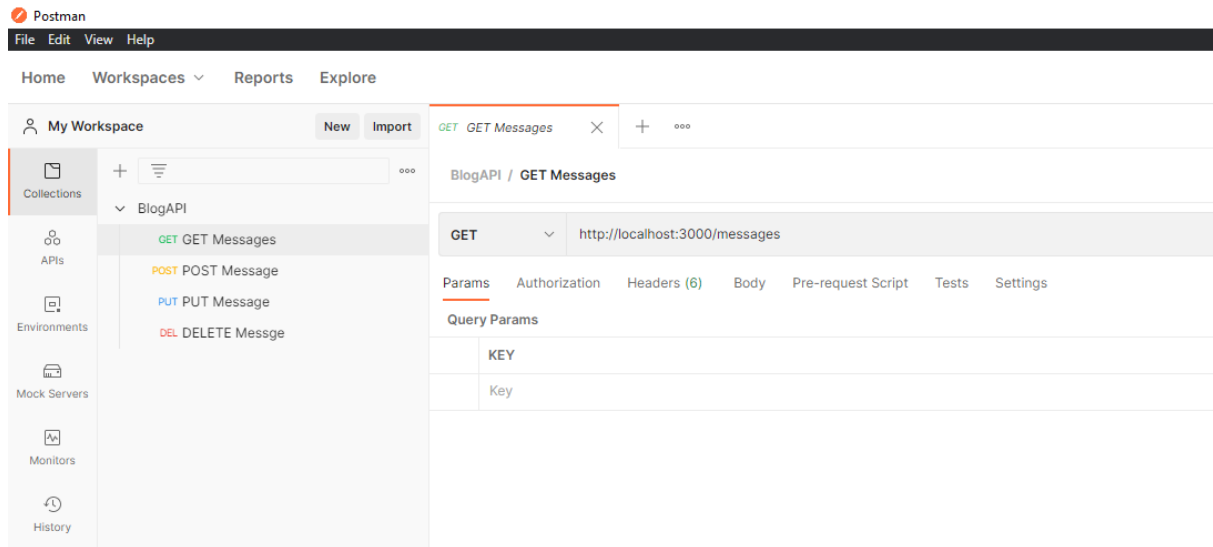
Fijaros que el **body** se tiene que configurar con la opción **raw** y de tipo **json**.



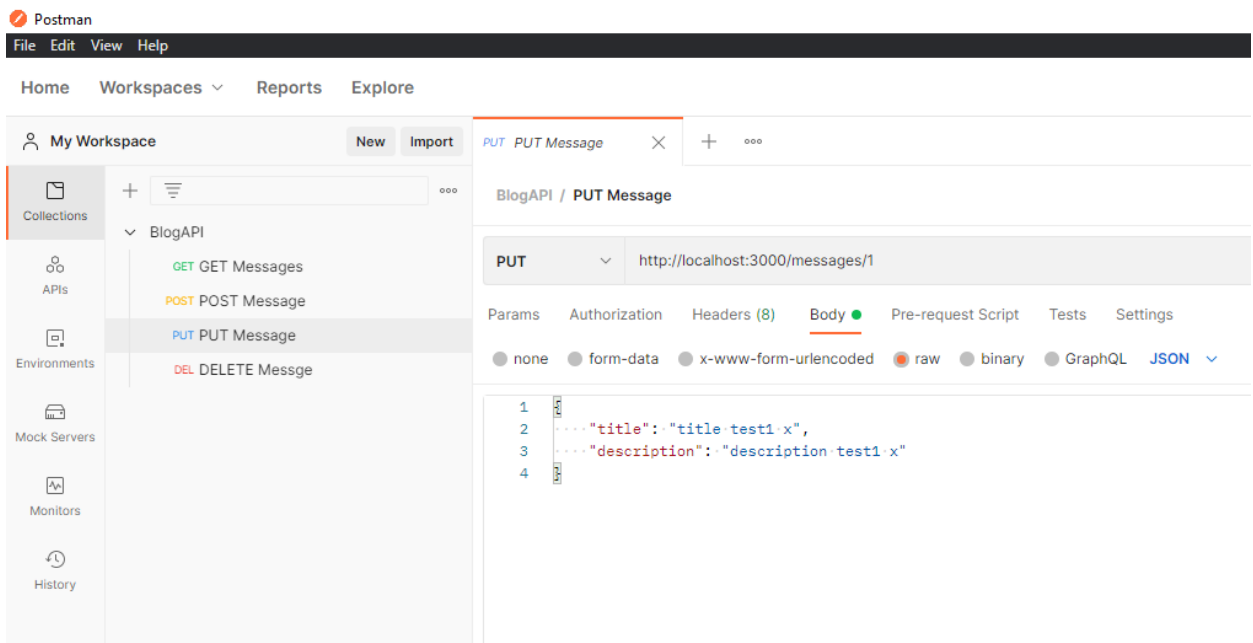
¡Un comentario!

*Fijaros que la **url** de las llamadas a la api están escuchando el puerto **3000**, esto viene de la configuración del **backend**, concretamente, del proyecto de la api **blog-api**, en el fichero **main.ts** está el puerto definido. Si por lo que sea necesitáis cambiar el puerto, lo cambiaríais en este fichero y listos. (si hacéis algún cambio en el **backend**, que a priori no os haría falta, pensad en parar la api y volver a levantarla para que apliquen los cambios.)*

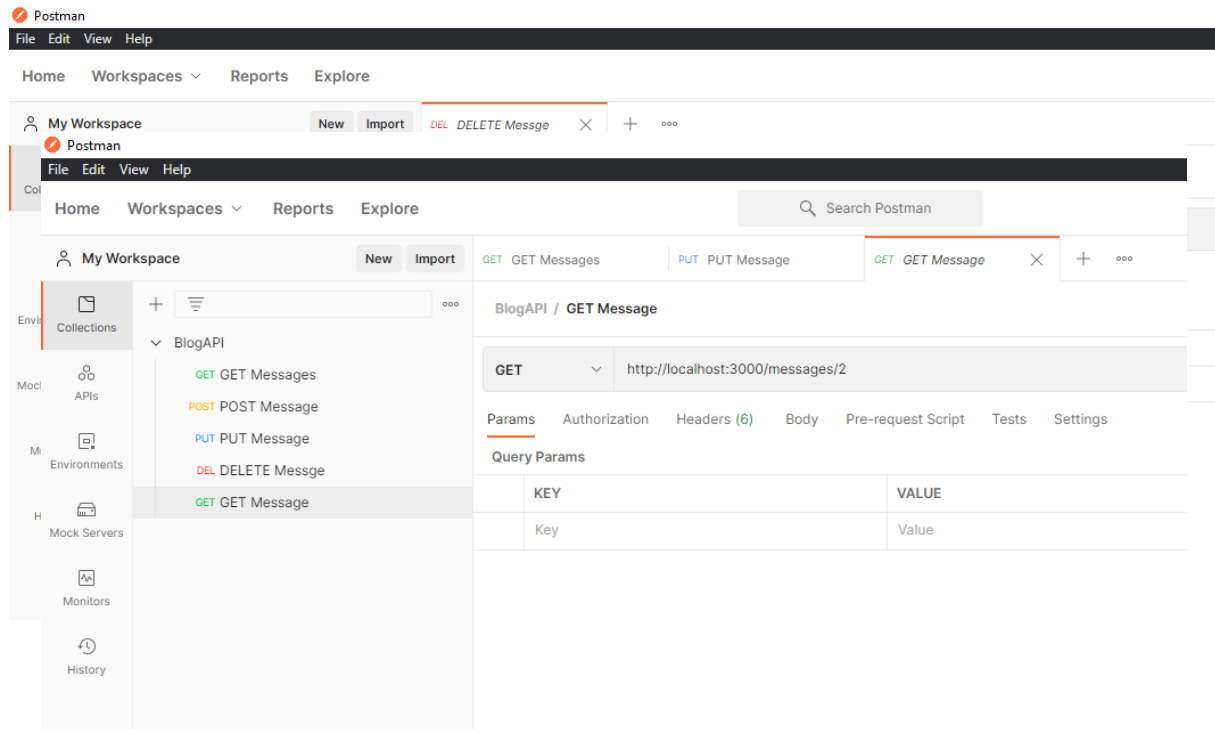
- Llamada GET que nos devolverá el listado de mensajes:



- Llamada PUT para modificar un mensaje existente, deberemos pasar por url el identificador del mensaje y en el cuerpo de la llamada un título y una descripción:



- Llamada **DELETE** para eliminar un mensaje existente, deberemos pasar por url el identificador del mensaje:

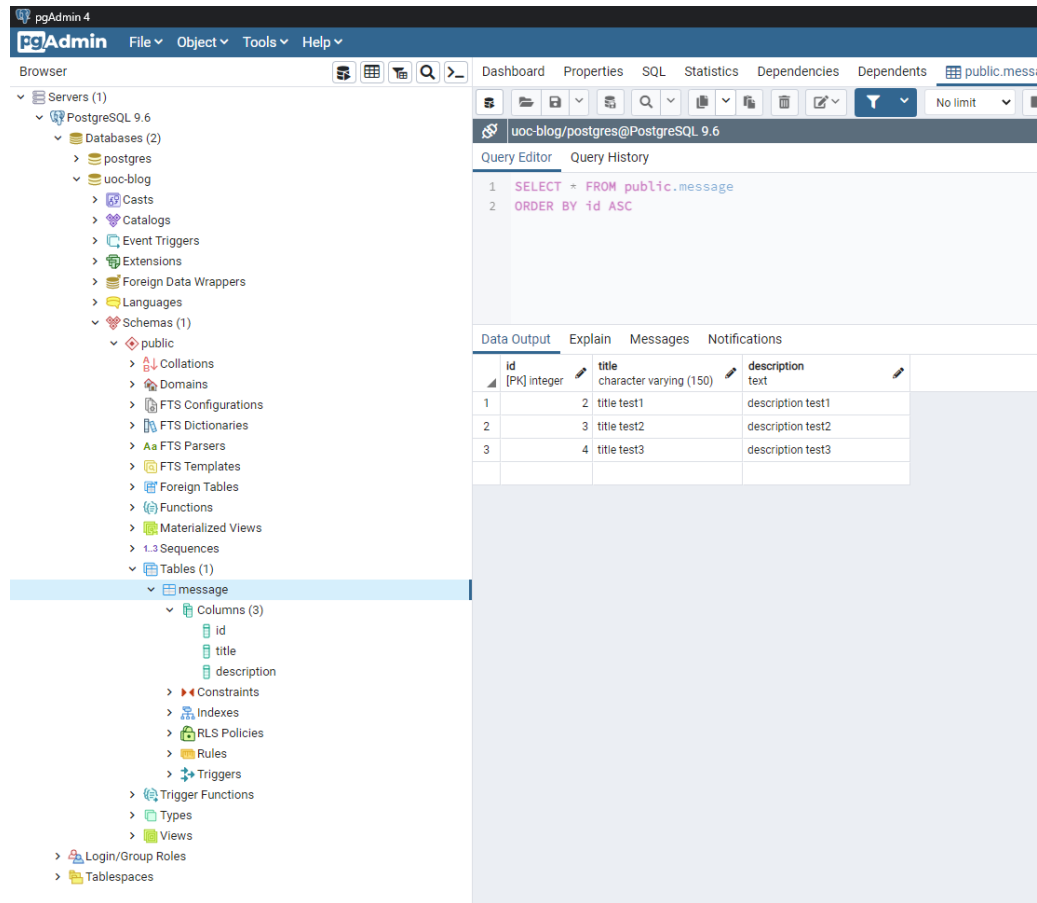


- Llamada **GET** para devolver la información de un mensaje en base a su id, necesario para cuando queramos actualizar, deberemos pasar por url el identificador del mensaje:

Ahora es momento de testar bien el **backend** desde el **Postman**.

Da de alta un mensaje, haz la llamada **GET** para ver que te devuelve el mensaje nuevo, haz una llamada **PUT**, vuelve a hacer el **GET**, da varios mensajes de alta, elimina alguno, ... observa la respuesta en cada caso que obtienes en el propio **Postman**.

Puedes ir validando cada paso también des de la base de datos:

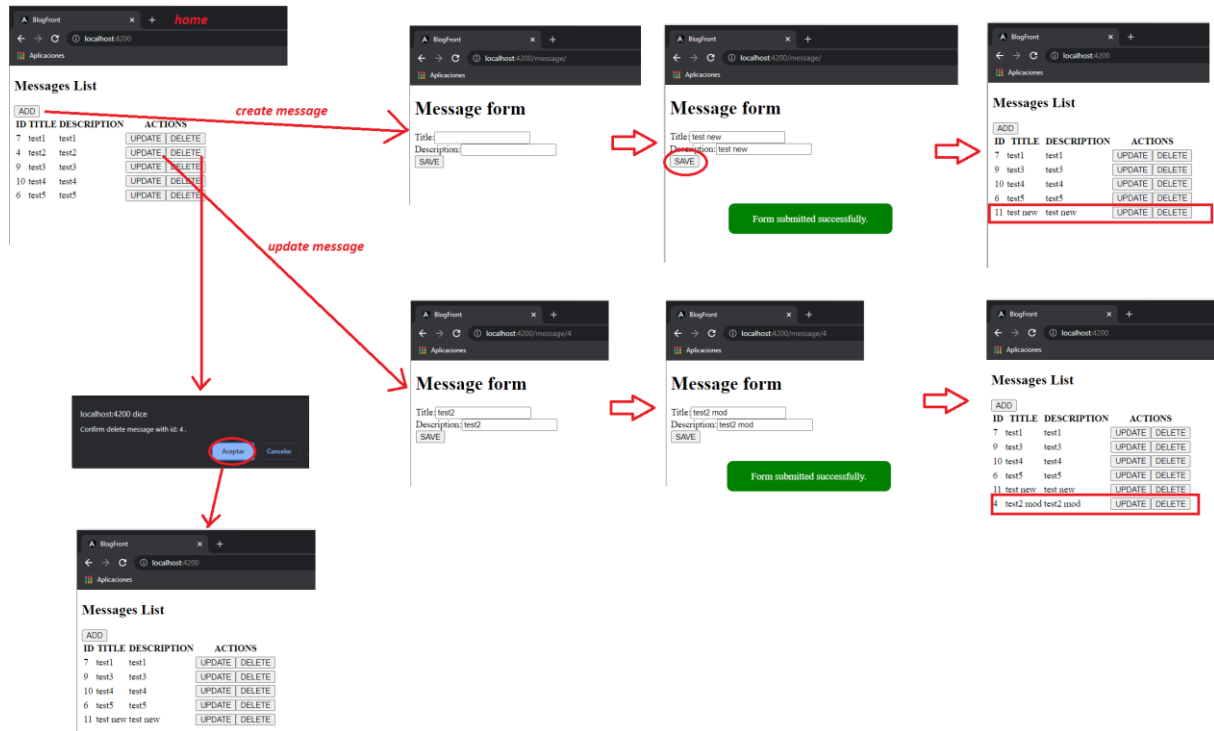


Como podemos ver el **Postman** es una herramienta muy útil, ahora tenemos un ejemplo de llamadas sencillas, pero en proyectos complejos se pueden utilizar variables y parametrizarse, por ejemplo, cuando hagamos llamadas en las que necesitemos un token de autenticación, a veces tendremos que pasar **headers**, ... todo esto lo podemos gestionar con este programa. Pero por lo pronto, tenemos las bases para seguir avanzando.

• Implementación del frontend:

Vamos a crear un **frontend** utilizando formularios reactivos que consuma la api anterior. Vamos a aprovechar esta implementación para repasar conceptos y coger buenas prácticas en cuanto a consumir una api con Angular.

Flujo del frontend:



La idea es la siguiente:

- Inicialmente accedemos a la **home**, donde tendremos el botón de añadir un nuevo mensaje y el listado de mensajes existentes.
- Para dar de alta un mensaje hacemos clic en el botón **ADD**, esto nos redirigirá a la página con el formulario de la entidad, al ser un alta, estarán los campos vacíos. Rellenaremos los campos y si pulsamos el botón de **SAVE** nos aparecerá un mensaje de confirmación y al cabo de 1,5 segundos nos redirigirá a la **home** con el listado actualizado con el último mensaje dado de alta.
- Para modificar un mensaje haremos clic en el botón **UPDATE** del mensaje que queramos modificar. Esto nos redirigirá a la página con el formulario de la entidad, al ser una modificación, estarán los campos cargados con los valores correspondientes. Modificaremos los campos necesarios y si pulsamos el botón de **SAVE** nos aparecerá un mensaje de confirmación y al cabo de 1,5 segundos nos redirigirá a la **home** con el listado actualizado con el mensaje con los campos modificados.
- Para eliminar un mensaje haremos clic en el botón **DELETE** del mensaje que queramos eliminar. Aparecerá un **alert** de confirmación, de manera que si pulsamos **cancelar** no se producirá ninguna acción, pero si pulsamos **aceptar** nos actualizará la **home** con el listado actualizado sin el mensaje que acabamos de eliminar.
-

Teniendo el flujo en mente, vamos a comentar la implementación.

Tienes este proyecto implementado y subido al campus con los nombres:

blog-api.zip

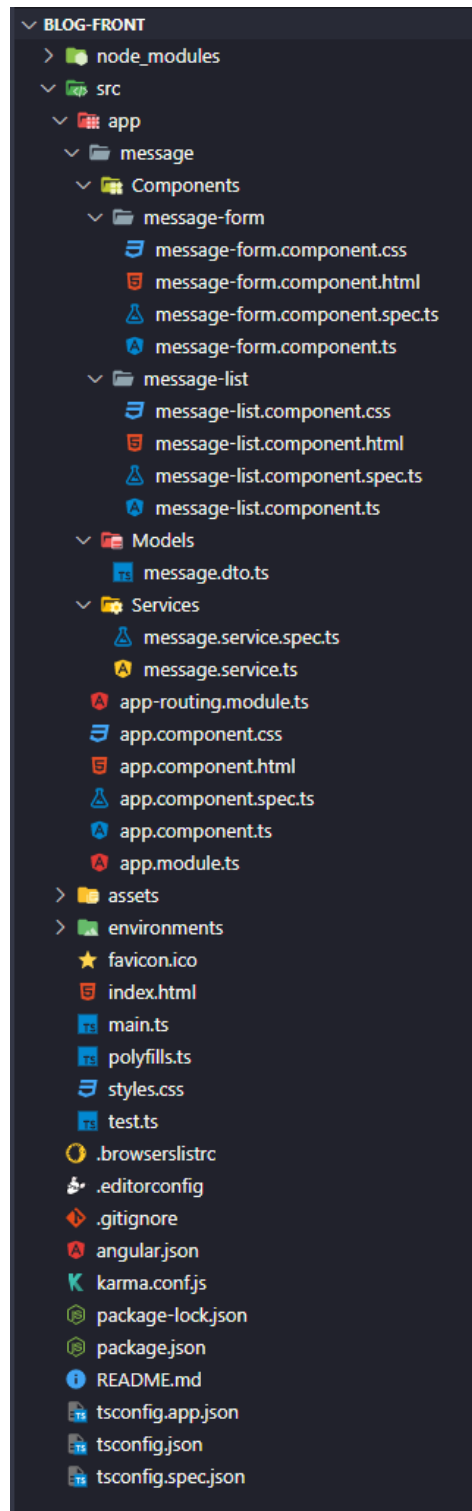
blog-front.zip

En este punto estaría bien bajarse el **frontend** del campus, hacer el **npm install** y el **ng serve** y validar que funciona correctamente la comunicación con el **backend**.

Estudiar cómo se ha implementado, posteriormente implementarlo.

Este ejercicio sirve de base para consolidar lo estudiado en la asignatura previa a esta y afrontar con más garantías la primera práctica.

Estructura de ficheros resultante del **frontend**:



- Fichero **app.module.ts**:

```

app.module.ts X
src > app > app.module.ts > ...
1  import { HttpClientModule } from '@angular/common/http'; 80.3K (gzipped: 24K)
2  import { NgModule } from '@angular/core'; 110.4K (gzipped: 35K)
3  import { ReactiveFormsModule } from '@angular/forms'; 64K (gzipped: 14.1K)
4  import { BrowserModule } from '@angular/platform-browser'; 20.1K (gzipped: 6.7K)
5  import { AppRoutingModule } from './app-routing.module';
6  import { AppComponent } from './app.component';
7  import { MessageFormComponent } from './message/Components/message-form/message-form.component';
8  import { MessageListComponent } from './message/Components/message-list/message-list.component';
9
10 @NgModule({
11   declarations: [AppComponent, MessageListComponent, MessageFormComponent],
12   imports: [
13     BrowserModule,
14     AppRoutingModule,
15     HttpClientModule,
16     ReactiveFormsModule, (B)
17   ],
18   providers: [],
19   bootstrap: [AppComponent],
20 })
21 export class AppModule {}

```

A. Para desarrollar este **frontend** sólo necesitaremos crear dos componentes, uno para el listado de mensajes y otro para crear/editar un mensaje. Cuando creamos los componentes **MessageListComponent** y **MessageFormComponent** nos insertará automáticamente las declaraciones de estos dos componentes en el **app.module**, en todo caso, no está de más revisarlo.

B. Para trabajar con los formularios reactivos tendremos que importar el módulo **ReactiveFormsModule** y para poder hacer peticiones **http** tendremos que importar el módulo **HttpClientModule**.

- Fichero **app-routing.module.ts**:

```

app-routing.module.ts X
src > app > app-routing.module.ts > ...
1  import { NgModule } from '@angular/core'; 110.4K (gzipped: 35K)
2  import { RouterModule, Routes } from '@angular/router'; 94.8K (gzipped: 24.3K)
3  import { MessageFormComponent } from './message/Components/message-form/message-form.component';
4  import { MessageListComponent } from './message/Components/message-list/message-list.component';
5
6  const routes: Routes = [
7    {
8      path: '',
9      component: MessageListComponent, (A)
10    },
11    {
12      path: 'message/:id',
13      component: MessageFormComponent, (B)
14    },
15  ];
16
17 @NgModule({
18   imports: [RouterModule.forRoot(routes)],
19   exports: [RouterModule],
20 })
21 export class AppRoutingModule {}

```


- A. La ruta raíz “ será nuestra **home**, que apuntará al componente **MessageListComponent** que contendrá el listado de mensajes de nuestra app y los botones para poder dar de alta, modificar o eliminar mensajes.
- B. Definiremos la ruta a la que iremos si hacemos clic al botón de añadir mensaje o al botón de modificar en alguno de los mensajes.

La idea es redirigir al componente **MessageFormComponent** de manera que:

- Si es una modificación, le enviaremos por **url** el **id** del mensaje a modificar. Con este **id**, el componente hará la llamada **getMessageById** para cargar los datos del mensaje en el formulario y poder editarlo.
- Si es un alta, no le enviaremos ningún **id** por **url** ya que al ser una alta no tenemos, con lo que inicialmente el formulario estará vacío, podremos insertar el título y la descripción determinada y guardar el nuevo mensaje.
- Haremos una pequeña lógica para discriminar entre alta o edición en el componente **MessageFormComponent**, posteriormente lo veremos.

● Fichero **message.dto.ts**

```

message.dto.ts X
src > app > message > Models > message.dto.ts > ...
1  export class MessageDTO {
2      id!: number;
3      title: string;
4      description: string;
5
6      constructor(title: string, description: string) {
7          this.title = title;
8          this.description = description;
9      }
10 }

```

Clase a la que le definimos las diferentes propiedades que necesitamos gestionar, en nuestro caso **id**, título y descripción. Implementamos su correspondiente constructor también.

Nótese que el campo **id** (que corresponde al **id** de la base de datos, es decir, a la clave primaria del registro) le añadimos el símbolo **!** y no utilizamos este campo **id** en el constructor. Esto es porque cuando creamos un mensaje nuevo no tenemos **id**, por lo tanto, necesitamos identificar este campo como opcional y esta sería la manera.

● Fichero **message.service.ts**

En el servicio **message.service.ts** implementaremos todas las peticiones a la api. Este podríamos decir que es el primer paso para comunicarnos con una api, es decir, desde el servicio hacemos las peticiones **http** y devolvemos la respuesta del **backend** mapeada. Es muy importante **tipar** los datos. Es decir, si el servidor nos devuelve un listado de mensajes, nosotros lo mapeamos/guardamos en un **MessageDTO[]**, es decir, en un array del tipo **MessageDTO**. Si el servidor nos devuelve 1 mensaje, pues lo mapeamos en una variable de tipo **MessageDTO**. Lo iremos viendo, lo importante es entender lo importante de aprovechar **typescript** en cuanto a definir el tipo de **todo** y evitar poner **any**.

Luego, desde el controlador (**message-list.component.ts** o **message-form.component.ts**) llamaremos al método/función determinado del servicio y trataremos la respuesta sea la esperada o sea de error con un **try/catch**. Por lo pronto, comentemos la implementación del servicio, teniendo en cuenta que utilizaremos **promesas** en lugar de **observables**, para que tengáis un ejemplo:

```

message.service.ts
src > app > message > Services > message.service.ts > ...
1 import { HttpClient, HttpResponse } from '@angular/common/http';
2 import { Injectable } from '@angular/core';
3 import { MessageDTO } from '../Models/message.dto';
4
5 interface deleteResponse {
6   affected: number;
7 }
8
9 @Injectable({
10   providedIn: 'root',
11 })
12 export class MessageService {
13   private urlMessageApi: string; (A)
14
15   constructor(private http: HttpClient) {
16     this.urlMessageApi = 'http://localhost:3000/messages'; (B)
17   }
18
19   getMessages(): Promise<MessageDTO[]> {
20     return this.http.get<MessageDTO[]>(this.urlMessageApi).toPromise(); (C)
21   }
22
23   getMessageById(msgId: number): Promise<MessageDTO> {
24     return this.http (D)
25       .get<MessageDTO>(this.urlMessageApi + '/' + msgId)
26       .toPromise();
27   }
28
29   createMessage(msg: MessageDTO): Promise<MessageDTO> { (E)
30     return this.http.post<MessageDTO>(this.urlMessageApi, msg).toPromise();
31   }
32
33   updateMessage(msgId: number, msg: MessageDTO): Promise<MessageDTO> {
34     return this.http (F)
35       .put<MessageDTO>(this.urlMessageApi + '/' + msgId, msg)
36       .toPromise();
37   }
38
39   deleteMessage(msgId: number): Promise<deleteResponse> { (G)
40     return this.http
41       .delete<deleteResponse>(this.urlMessageApi + '/' + msgId)
42       .toPromise();
43   }
44
45   errorLog(error: HttpResponse): void {
46     console.error('An error occurred:', error.error.msg);
47     console.error('Backend returned code:', error.status); (H)
48     console.error('Complete message was:', error.message);
49   }
50
51   async wait(ms: number) {
52     return new Promise((resolve) => { (I)
53       setTimeout(resolve, ms);
54     });
55   }

```

- A. Definimos una variable privada para reutilizar la **url** de las llamadas a la api en diferentes partes del código.
- B. Inyectamos el servicio **HttpClient** para poder hacer las llamadas **http** e inicializamos la variable **urlMessageApi**.
- C. Implementación de la llamada que devuelve todos los mensajes de la api:
 - a. No le tenemos que pasar ningún parámetro

- b. Hacemos una petición **http.get** con la **url** base. Fijaros muy importante, que tipamos la respuesta, tanto en el **this.http.get<MessageDTO[]>** como en la definición de la función definiendo que devuelve **Promise<MessageDTO[]>**, es decir, que devuelve una promesa de un array de **MessageDTO**.
- c. La función es de tipo pública, ya que necesitaremos llamarla desde el controlador (en nuestro caso, des del fichero **message-list.component.ts**)

D. Implementación de la llamada que devuelve un mensaje en función de su **id:**

- a. Le pasamos por parámetro el **id** del mensaje que queremos devolver
- b. Hacemos una petición **http.get** con la **url** base y concatenamos el **id**. Fijaros igual que antes, que tipamos la respuesta, tanto en el **this.http.get<MessageDTO>...** como en la definición de la función definiendo que devuelve **Promise<MessageDTO>**, es decir, devolverá el mensaje determinado que es de tipo **MessageDTO**.
- c. La función es de tipo pública, ya que la llamaremos desde el controlador.

E. Implementación de la llamada que crea un nuevo mensaje en la base de datos:

- a. Le pasaremos por parámetro el mensaje que queremos dar de alta. Este parámetro será el **body** de la petición **post**
- b. Hacemos una petición **http.post** con la **url** base y como **body** le pasaremos el mensaje que queremos dar de alta. Fijaros igual que antes, que tipamos la respuesta, tanto en el **this.http.post<MessageDTO>...** como en la definición de la función definiendo que devuelve **Promise<MessageDTO>**, es decir, que devuelve una promesa del **MessageDTO** una vez creado.
- c. ¿Esto qué quiere decir? Que cuando demos de alta un nuevo mensaje, una vez hecha la llamada **post**, el servidor nos responderá con el mensaje recién dado de alta. Esto nos puede ser útil si queremos utilizar este mensaje dado de alta para hacer alguna acción. En nuestro caso, lo veremos después, nosotr@s cuando demos de alta un mensaje, no nos hará falta tratar la respuesta a este **post** ya que redirigiremos a la **home**, y este componente **message-list** hará la llamada **getMessages** y actualizará la lista con todos los mensajes, incluido el que acabamos de dar de alta. Esto dependerá del comportamiento que queramos darle a nuestra aplicación.
- d. La función es de tipo pública, ya que la llamaremos desde el controlador.

F. Implementación de la llamada que actualiza un mensaje existente en la base de datos:

- a. Le pasaremos por parámetro el **id** del mensaje que queremos modificar y un objeto tipo **MessageDTO** con las propiedades que hayamos podido modificar. Este objeto será el **body** de la petición **put**
- b. Hacemos una petición **http.put** con la **url** base y concatenamos el **id** y como **body** le pasaremos el mensaje que queremos modificar. Fijaros igual que antes, que tipamos la respuesta, tanto en el **this.http.put<MessageDTO>...** como en la definición de la función definiendo que devuelve

Promise<MessageDTO>, es decir, que devuelve una promesa del **MessageDTO** una vez modificado.

- c. Al igual que el alta, cuando hacemos una modificación la api nos devuelve el mensaje modificado, nosotr@s lo podremos utilizar o no.
- d. La función es de tipo pública, ya que la llamaremos desde el controlador.

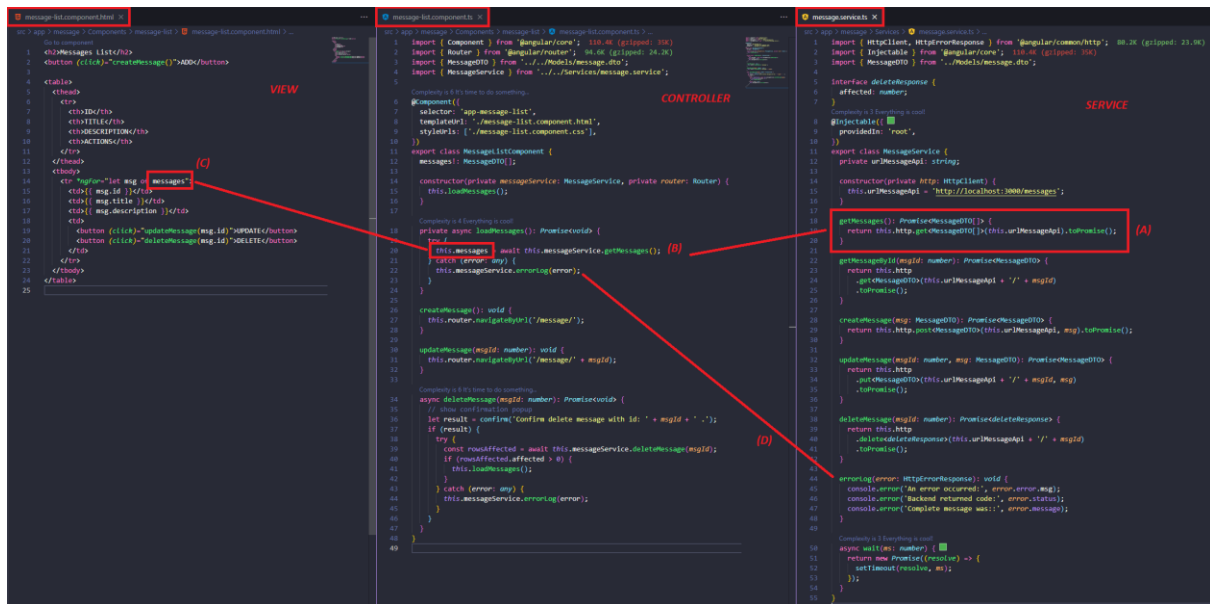
G. Implementación de la llamada que elimina un mensaje de la base de datos:

- a. Le pasaremos por parámetro el **id** del mensaje que queremos eliminar.
- b. Hacemos una petición **http.delete** con la **url** base y concatenamos el **id** del mensaje que queremos eliminar. Fijaros igual que antes, que tipamos la respuesta, tanto en el **this.http.put<deleteResponse>...** como en la definición de la función definiendo que devuelve **Promise<deleteResponse>**, es decir, que devuelve una promesa de tipo **deleteResponse** una vez eliminado.
- c. En este caso, la llamada de eliminación devuelve un tipo concreto, si os fijáis en la parte superior del fichero **message.service.ts** hemos creado una interfaz para poder tipar esta respuesta en el caso de eliminar un registro. Básicamente nos sirve para mapear el número de filas afectadas de la base de datos. Esto nos puede venir bien para identificar que realmente se ha eliminado un mensaje o no, después lo veremos en el controlador.
- d. La función es de tipo pública, ya que la llamaremos desde el controlador.

H. Método para reutilizar en las diferentes llamadas del controlador para mostrar por consola del navegador el código de error y los mensajes asociados al error en caso de que se produzcan.

I. Método para reutilizar para simular un **delay**, lo utilizaremos para mostrar un mensaje de confirmación o de error cuando realicemos las acciones de alta o modificación y que al cabo de un tiempo determinado desaparezca el mensaje de confirmación u error y nos redirija a la **home**.

• Vemos el flujo completo de un ejemplo < servicio – controlador – vista >



Repasemos un flujo completo: ¿Cómo se hace la carga inicial de la HOME?

- A. Tenemos el servicio que devuelve todos los mensajes del sistema.
- B. Fijémonos en el controlador:
 - a. Cuando se carga el controlador y se lanza el **constructor** se llama a **loadMessages** para realizar la carga inicial de datos.
 - b. **loadMessages** es un método que llama al servicio **getMessages**.
 - c. Fijaros que utilizamos la estructura **try/catch**, de manera que intenta hacer la llamada, si es correcta sigue el flujo del código y si se produce un error, entraría dentro del **catch**, que, en este caso, simplemente llama al método que muestra los **logs** importantes por la consola del navegador. Esto es simplemente a modo de ejemplo. Normalmente los errores en las peticiones se gestionan desde un **interceptor** general, es decir, se interceptan todas las llamadas **http** y si se detecta un error (404 no disponible, 401 permisos, ...) se actúa en consecuencia, sea redirigir a una página de error o lo que sea, de esta manera no hace falta tratar el caso de error de cada llamada, lo tendríamos centralizado en este **interceptor** que comentaba. Pero para este caso de estudio ya nos valdría.
 - d. Importante aquí el **await** delante de la llamada. Es decir, estamos diciendo, espera a tener la respuesta a la llamada, cuando la tengas, asignas la respuesta a la variable **messages** y sigues el curso del código. El hecho de que pongamos un **await** porque queremos que no siga el código hasta que tengamos la respuesta, hace que tengamos que ponerle el **async** en la declaración del método ya que en este caso sería un método asíncrono. De manera parecida, tendríamos que poner el **Promise <void>** en la declaración del método.

- C. Cuando se obtenga la respuesta y **messages** tenga el contenido, se nos pintaría la vista mediante el bucle **ngFor** que pintaría una fila por cada mensaje que tengamos.
- D. Esta función simplemente es para mostrar los **logs** de errores por consola. Por ejemplo, pasadle un **ID** que no exista en la base de datos a la llamada de **updateMessage** y podréis ver por consola los errores (mensaje y código de error).

- Fichero **message-list.component.ts**

```

message-list.component.ts X
src > app > message > Components > message-list > message-list.component.ts > ...
1  import { Component } from '@angular/core'; 110.4K (gzipped: 35K)
2  import { Router } from '@angular/router'; 94.6K (gzipped: 24.2K)
3  import { MessageDTO } from '../Models/message.dto';
4  import { MessageService } from '../Services/message.service';
5
6  Complexity is 6 It's time to do something...
7  @Component({
8    selector: 'app-message-list',
9    templateUrl: './message-list.component.html',
10   styleUrls: ['./message-list.component.css'],
11 })
12 export class MessageListComponent {
13   messages!: MessageDTO[]; (A)
14
15   constructor(private messageService: MessageService, private router: Router) {
16     this.loadMessages(); (B)
17   } (C)
18
19   Complexity is 4 Everything is cool!
20   private async loadMessages(): Promise<void> {
21     try {
22       this.messages = await this.messageService.getMessages(); (D)
23     } catch (error: any) {
24       this.messageService.errorLog(error);
25     }
26   }
27
28   createMessage(): void {
29     this.router.navigateByUrl('/message/'); (E)
30   }
31
32   updateMessage(msgId: number): void {
33     this.router.navigateByUrl('/message/' + msgId); (F)
34   }
35
36   Complexity is 6 It's time to do something...
37   async deleteMessage(msgId: number): Promise<void> {
38     // show confirmation popup
39     let result = confirm('Confirm delete message with id: ' + msgId + '.');
40     if (result) {
41       try {
42         const rowsAffected = await this.messageService.deleteMessage(msgId);
43         if (rowsAffected.affected > 0) {
44           this.loadMessages(); (G)
45         }
46       } catch (error: any) {
47         this.messageService.errorLog(error);
48       }
49     }
50   }
51 }

```

- A. Definimos la variable pública **messages** que será en la que guardaremos el **array** de mensajes que mostraremos en la vista. Nótese que tenemos que añadir **!** para que nos deje limpiar un poco el código y poder “sacar” la carga de datos en un método **loadMessages**. Si no pusiéramos en **!** tendríamos que poner el código directamente en el constructor, así queda más limpio.
- B. Inyectamos nuestro servicio **MessageService** para hacer las llamadas necesarias a nuestra api y el servicio **Router** para poder navegar al componente **MessageFormComponent** cuando queramos dar de alta o modificar un mensaje.
- C. En el **constructor** lanzamos el método **loadMessages()** (explicado al punto anterior).
- D. Explicado al punto anterior.
- E. Si pulsamos al botón de **ADD** (añadir nuevo mensaje) navegaremos a la vista de alta/edición (componente **message-form**) sin pasarle **id** por **url**.
- F. Si pulsamos al botón de **UPDATE** (modificar mensaje) navegaremos a la vista de alta/edición (componente **message-form**) pasándole el **id** del mensaje en cuestión por **url**.
- G. Si pulsamos al botón de **DELETE** (eliminar mensaje) se producirá la siguiente lógica:
 - a. Mostraremos un mensaje de confirmación.
 - i. Si cancelamos no se realizará ninguna acción.
 - ii. Si aceptamos, se hará la llamada para eliminar el mensaje.
 - iii. Tenemos implementada la estructura **try / catch**:
 1. Si la llamada se hace correctamente se eliminará el mensaje y devolverá que se ha afectado a una fila, con lo que en **rowsAffected** tendremos el valor 1.
 2. Hacemos una pequeña validación (**rowsAffected > 0**) para asegurarnos que si realmente se ha eliminado el mensaje recargue la lista haciendo la llamada al método **loadMessages**.
 3. En caso de error entraríamos en el **catch** y mostraremos los mensajes y código de error por la consola del navegador.

Fijaros algunos detalles:

- Poner **await** delante de la llamada hace que hasta que no se obtenga la respuesta no se ejecuta el código que hay debajo, es decir, el condicional que compara **rowsAffected** con 0.
- El poner el **await** hace que tengamos que poner **async** y **Promise** en la declaración del método.
 - **Buenas prácticas:**
- Cuando definimos una variable no se utiliza **var**, es mejor utilizar **let** o **const**. Si no se va a reasignar la variable utilizamos **const**, en otro caso se usa **let**. Si declaramos una variable **const/let** al principio del método esta variable será accesible en todo el método, si declaramos una variable **const/let** dentro de un condicional, por ejemplo, su ámbito será sólo dentro de ese condicional.
- Otra cosa en los condicionales:

- Mejor **if(variable)** que **if(variable == true)**
- Mejor **if(!variable)** que **if(variable == false)**
- Siempre que se pueda, mejor utilizar los **“===”** que **“==”**
 - Así comparamos tanto valor como tipo.

- Fichero **message-list.component.html**

```
message-list.component.html X
src > app > message > Components > message-list > message-list.component.html > ...
1  <h2>Messages List</h2>
2  <button (click)="createMessage()">ADD</button>
3
4  <table>
5    <thead>
6      <tr>
7        <th>ID</th>
8        <th>TITLE</th>
9        <th>DESCRIPTION</th>
10       <th>ACTIONS</th>
11     </tr>
12   </thead>
13   <tbody>
14     <tr *ngFor="let msg of messages">
15       <td>{{ msg.id }}</td>
16       <td>{{ msg.title }}</td>
17       <td>{{ msg.description }}</td>
18       <td>
19         <button (click)="updateMessage(msg)">UPDATE</button>
20         <button (click)="deleteMessage(msg.id)">DELETE</button>
21       </td>
22     </tr>
23   </tbody>
24 </table>
```

Maquetamos una tabla en la que cada fila sea un mensaje. Para ello implementamos un **ngFor** para que por cada mensaje cree una fila de la tabla.

Por cada fila tendremos una columna de acciones donde podremos modificar o eliminar el mensaje determinado.

La vista no tiene mucho misterio. Aquí el comentario es que cuando trabajamos el tema 3 y estudiemos **Angular Material**, veremos que podemos utilizar componentes que nos ayudarán a maquetar rápido y con resultados profesionales.

- Fichero **message-form.component.ts**

```

1  import { Component, OnInit } from '@angular/core'; 110.6K (gzipped: 35.1K)
2  import {
3    FormBuilder,
4    FormControl,
5    FormGroup,
6    Validators,
7  } from '@angular/forms'; 64.5K (gzipped: 14.2K)
8  import { ActivatedRoute, Router } from '@angular/router'; 94.6K (gzipped: 24.2K)
9  import { MessageDTO } from '../Models/message.dto';
10 import { MessageService } from '../Services/message.service';
11
12 @Component({
13   selector: 'app-message-form',
14   templateUrl: './message-form.component.html',
15   styleUrls: ['./message-form.component.css'],
16 })
17 export class MessageFormComponent implements OnInit {
18   message: MessageDTO;
19   title: FormControl;
20   description: FormControl;
21   messageForm: FormGroup;
22   isValidForm: boolean | null; (A)
23
24   private isUpdateMode: boolean;
25   private validRequest: boolean;
26   private msgId: string | null;
27
28   constructor(
29     private activatedRoute: ActivatedRoute,
30     private messageService: MessageService,
31     private formBuilder: FormBuilder,
32     private router: Router
33   ) {} (B)
34
35   this.isValidForm = null;
36   this.msgId = this.activatedRoute.snapshot.paramMap.get('id');
37   this.message = new MessageDTO('', '');
38   this.isUpdateMode = false;
39   this.validRequest = false;
40
41   this.title = new FormControl(this.message.title, [
42     Validators.required,
43     Validators.maxLength(150),
44   ]);
45
46   this.description = new FormControl(
47     this.message.description,
48     Validators.required
49   );
50
51   this.messageForm = this.formBuilder.group({
52     title: this.title,
53     description: this.description,
54   });
55
56   async ngOnInit(): Promise<void> { (C)
57     // update
58     if (this.msgId) {
59       this.isUpdateMode = true;
60       try {
61         this.message = await this.messageService.getMessageById(+this.msgId);
62         this.title.setValue(this.message.title);
63         this.description.setValue(this.message.description);
64
65         this.messageForm = this.formBuilder.group({
66           title: this.title,
67           description: this.description,
68         });
69       } catch (error: any) {
70         this.messageService.errorLog(error);
71       }
72     }
73   }
74 }

```

```

76 private async managementToast(): Promise<void> { (G)
77   // It is just to give an example of a confirmation message using a toast
78   const toastMsg = document.getElementById('toastMessage');
79   if (toastMsg) {
80     // Request OK, show toast and go to home
81     if (this.validRequest) {
82       toastMsg.className = 'show requestOk';
83       toastMsg.textContent = 'Form submitted successfully.';
84       await this.messageService.wait(1500);
85       toastMsg.className = toastMsg.className.replace('show', '');
86       this.router.navigateByUrl('');
87     }
88     // Request KO, show toast and stay here
89     else {
90       toastMsg.className = 'show requestKo';
91       toastMsg.textContent = 'Error on form submitted, show logs.';
92       await this.messageService.wait(1500);
93       toastMsg.className = toastMsg.className.replace('show', '');
94     }
95   }
96 }
97
98 private async editMessage(): Promise<boolean> { (F)
99   let responseOK: boolean = false;
100   if (this.msgId) {
101     try {
102       await this.messageService.updateMessage(+this.msgId, this.message);
103       responseOK = true;
104     } catch (error: any) {
105       this.messageService.errorLog(error);
106     }
107   }
108   return responseOK;
109 }
110
111 private async createMessage(): Promise<boolean> { (E)
112   let responseOK: boolean = false;
113   try {
114     await this.messageService.createMessage(this.message);
115     responseOK = true;
116   } catch (error: any) {
117     this.messageService.errorLog(error);
118   }
119   return responseOK;
120 }
121
122 async saveMessage() { (D)
123   this.isValidForm = false;
124   if (this.messageForm.invalid) {
125     return;
126   }
127
128   this.isValidForm = true;
129   this.message = this.messageForm.value;
130
131   if (this.isUpdateMode) {
132     this.validRequest = await this.editMessage();
133   } else {
134     this.validRequest = await this.createMessage();
135   }
136
137   this.managementToast();
138 }
139
140 }
141

```

- A. Definimos todas las variables públicas y privadas, primero las públicas y luego las privadas.
 - B. Inyectamos todos los servicios que necesitemos e inicializamos las variables necesarias, incluidos los elementos del formulario (**FormControl** y **FormBuilder**). Además, recogemos el parámetro **id** que nos viene por **url** y lo asignamos a **msgId**
 - a. Hay que recordar que, si venimos desde un alta, este valor será nulo
 - b. Si venimos desde una modificación, el valor sí que estará informado
- Del constructor saldremos con el formulario inicializado con los campos vacíos para dar un alta. Luego en el **ngOnInit** se cargan los datos en el formulario **solo en caso de que se tratara de una actualización**.
- C. Al navegar al componente, sea por un alta o por una modificación, tenemos que ser capaces de discriminar si estamos haciendo un alta o una modificación.
 - a. Básicamente podemos comprobar si **msgId** tiene información o no, si la tiene estamos ante una modificación, sino, ante un alta nueva. Si es una alta no tenemos que hacer nada, puesto que tenemos el formulario inicializado del constructor. Si es una modificación, tendremos que pedir los datos del mensaje en cuestión y cargarlos en el formulario para poder editarlos.
 - b. Una vez sabemos esto, si es una modificación pediremos el mensaje a la api con el **msgId** para recuperar sus datos e inicializar los inputs del formulario reactivo. Si os fijáis hay un signo **+** delante de **msgId** cuando lo pasamos a la función **getMessageById**. Esto es porque la variable **msgId** la tenemos definida como **string** ya que cuando recogemos el **id** de la **url** nos devuelve un **string**, con el **+** lo que conseguimos es que nos lo convierta a numérico, que es el tipo que espera la función **getMessageById**
 - c. Nótese que si es una actualización posteriormente a la llamada **getMessageById** tendremos que guardar los valores de la respuesta en los correspondientes **FormControl** mediante **setValue** y volver a construir el **formBuilder**, de esta manera tendremos los datos cargados en el formulario.
 - D. Cuando pulsamos el botón de guardar validamos el formulario.
 - a. Si no es correcto, se mostrarán los mensajes de error (campo requerido, ...)
 - b. Si es correcto se lanza la llamada a **editMessage** o **createMessage** en función de si es alta o modificación y retorna **true** o **false** en función de si se ha realizado correctamente o no.
 - E. Si es un alta hacemos el **try / catch** de manera que si va bien devolvemos un **true** y si se produce un error devolvemos **false**
 - F. Si es una modificación hacemos el **try / catch** de manera que si va bien devolvemos un **true** y si se produce un error devolvemos **false**
 - G. Esto simplemente es para manejar algún tipo de aviso:
 - a. Sí va mal muestra el siguiente aviso y al cabo de 1,5 segundos desaparece y no realiza ninguna acción:

Error on form submitted, show logs.

- b. Sí va bien muestra el siguiente aviso y al cabo de 1,5 segundo redirige a la vista del listado:

Form submitted successfully.

- Fichero **message-form.component.html**

```

message-form.component.html X
src > app > message > Components > message-form > message-form.component.html > ...
1  <div>
2    <h1>Message form</h1>
3    <p id="toastMessage"></p>
4    <form
5      *ngIf="messageForm"
6      [formGroup]="messageForm"
7      (ngSubmit)="saveMessage()"
8    >
9      <div>
10       <label for="title">Title:</label>
11       <input type="text" [formControl]="title" />
12
13       <div *ngIf="title.errors">
14         <div
15           *ngIf="title.errors && isValidForm != null && !isValidForm"
16           [ngClass]="'error'"
17         >
18           <div *ngIf="title.errors.required">Title is required</div>
19
20           <div *ngIf="title.errors.maxlength">
21             Title can be max 150 characters long.
22           </div>
23         </div>
24       </div>
25     </div>
26
27     <div>
28       <label for="description">Description:</label>
29       <input type="text" [formControl]="description" />
30
31       <div *ngIf="description.errors">
32         <div
33           *ngIf="description.errors && isValidForm != null && !isValidForm"
34           [ngClass]="'error'"
35         >
36           <div *ngIf="description.errors.required">Description is required</div>
37         </div>
38       </div>
39     </div>
40
41     <button type="submit">SAVE</button>
42   </form>
43 </div>
44

```

Maquetación del formulario reactivo para modificar o dar de alta un mensaje.

Dos comentarios respecto a las vistas de estos ejemplos:

- Nunca deberíamos tener texto plano en las vistas, esto que ahora tengo **Message Form, Title:, Title is required**, ... esto se debe tratar con el paquete multi idioma **i18n** de Angular, pero lo veremos en la primera práctica.
- Por otra parte, como se ha comentado antes, ahora estamos utilizando en la vista **html** sin más, ya veremos cuando utilicemos **Angular Material** como podemos maquetar de manera más ágil y con un aspecto mucho más profesional, pero poco a poco.