15 - Formularios reactivos : FormGroup

Si tenemos un solo control de entrada de datos podemos como en el concepto anterior definir únicamente un objeto de la clase FormControl, pero un formulario que contiene un conjunto de controles la manera más efectiva es agruparlos definiendo un objeto de tipo FormGroup.

Problema

Confeccionar un formulario de contacto que permita cargar el nombre, mail y un mensaje. Cuando se presiona un botón mostrar mediante interpolación los datos cargados.

- Crearemos primero el proyecto:

```
ng new proyecto010
```

- Modificamos la vista de la componente que muestra el formulario reactivo (app.component.html):

```
<form [formGroup]="formularioContacto" (ngSubmit)="submit()">
  <p>Nombre:
    <input type="text" formControlName="nombre">
  </p>
  <p>Mail:
    <input type="text" formControlName="mail">
  </p>
  <p>Mensaje<br>
    <textarea rows="10" cols="70" formControlName="mensaje"></text
area>
  </p>
  <button type="submit">Confirmar</button>
</form>
<div>{{datos}}</div>
<router-outlet />
```

Analizaremos este archivo en conjunto luego de presentar 'app.component.ts'

- La clase asociada a la vista es (app.component.ts):

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ReactiveFormsModule, FormControl, FormGroup } from '@angu
lar/forms';

@Component({
  selector: 'app-root',
  imports: [RouterOutlet, ReactiveFormsModule],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  datos='';

  formularioContacto = new FormGroup({
    nombre: new FormControl(''),
    mail: new FormControl(''),
    mensaje: new FormControl('')
  });

  submit() {
    this.datos = `Nombre=${this.formularioContacto.value.nombre}
                  Mail=${this.formularioContacto.value.mail}
                  Mensaje=${this.formularioContacto.value.mensaje}
                  `;
  }
}
```

Importamos ahora:

```
import { ReactiveFormsModule, FormControl, FormGroup } from '@angular/forms';
```

Creamos un objeto de la clase FormGroup y le pasamos al constructor del mismo un objeto literal con la creación de un objeto de la clase FormControl por cada control visual de la vista:

```
  formularioContacto = new FormGroup({
    nombre: new FormControl(''),
    mail: new FormControl(''),
    mensaje: new FormControl('')
  });
```

Podemos pasar a cada constructor el valor inicial que tenga el control visual, en nuestro caso hemos pasado string vacíos.

En la vista agrupamos todos los controles mediante la etiqueta 'form' y definimos la directiva [formGroup] asignando la variable de tipo FormGroup definida en la clase:

```
<form [formGroup]="formularioContacto" (ngSubmit)="submit()">
```

Asociamos el evento (ngSubmit) con el método de la clase que captura el clic del botón del formulario.

Cada uno de los controles requiere definir la propiedad 'formControlName' con los nombres asignados en la clase cuando creamos cada FormControl:

```
<p>Nombre:
  <input type="text" formControlName="nombre">
</p>
<p>Mail:
  <input type="text" formControlName="mail">
</p>
<p>Mensaje<br>
  <textarea rows="10" cols="70" formControlName="mensaje"></textarea>
</p>
```

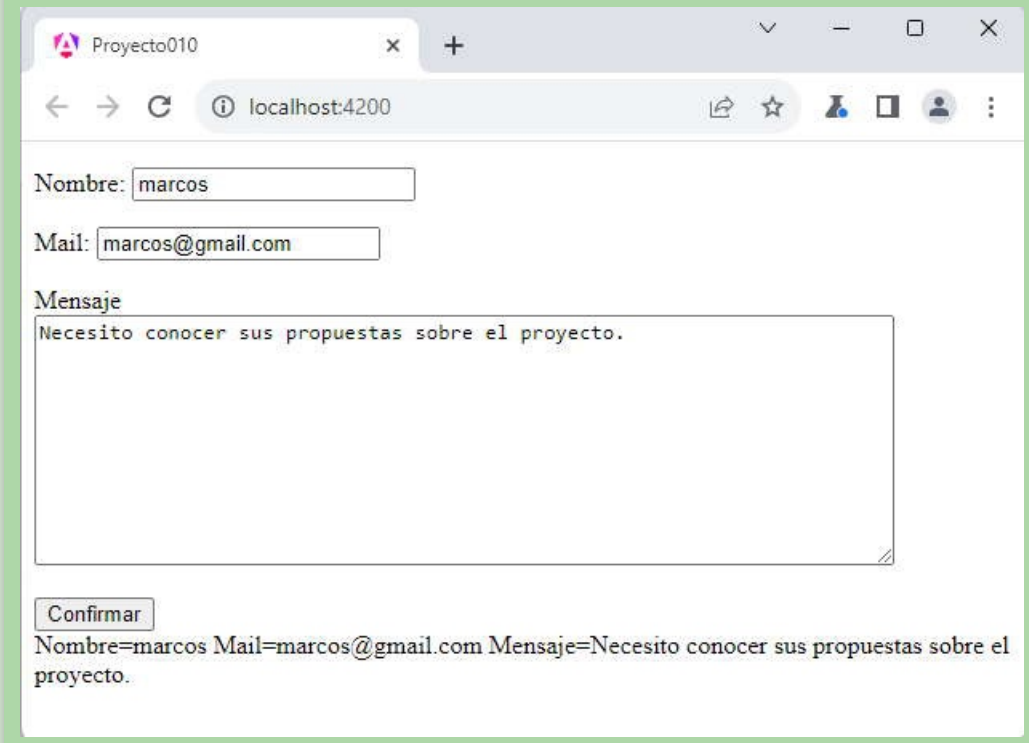
El método 'submit' procede a recuperar cada valor ingresado en el formulario y lo almacena en el atributo 'datos':

```
submit() {
  this.datos=`Nombre=${this.formularioContacto.value.nombre}
              Mail=${this.formularioContacto.value.mail}
              Mensaje=${this.formularioContacto.value.mensaje}
              `;
```

El atributo datos se muestra en la vista mediante interpolación:

```
<div>{{datos}}</div>
```

Si ejecutamos la aplicación tenemos una interfaz similar a:



Podemos probar esta aplicación en la web aquí.

Retornar