Formularios Angular

Introducción

Los formularios son el aspecto más crucial de una aplicación web. Si bien a menudo obtenemos información a través de los eventos, acciones que realiza el usuario o la misma aplicación, es a través de los formularios en que obtenemos la mayoría de los datos de los usuarios.

Los Formularios pueden ser algo complejo, fundamentalmente por las siguientes razones:

- Los inputs de los forms implican que se van a modificar datos, ya sea en la página como en el servidor.
- Los cambios muchas veces deben ser reflejados en varias partes de la página.
- Los usuarios tienen mucha libertad de acción en relación a lo que ingresan, por lo tanto necesitamos validar los valores de entrada.
- La interfaz de usuario debe indicar claramente las expectativas y los errores, si los hay.
- Los campos dependientes pueden tener una lógica compleja.

Para todas estas cuestiones, Angular provee una serie de herramientas que nos ayudan en estas cuestiones:

- **FormControls**: Encapsula los inputs de nuestros formularios y nos devuelve objetos para que podamos trabajar con ellos.
- **Validators**: nos permite validar los inputs, de la forma en que creamos conveniente.
- **Observers**: podemos ver los cambios en nuestro formulario y responder de acuerdo a ellos.

Angular también provee dos enfoques diferentes para manejar los inputs de usuario en un formulario: *reactive* y *template-driven*. Ambos capturan los eventos de entrada del usuario desde la vista, validan dicha entrada, crean un modelo de formulario y un modelo de datos para actualizar, y proporcionan una forma de realizar un seguimiento de los cambios.

FormControls y FormGroups

Los dos objetos fundamentales en los formularios Angular Forms son **FormControl** y **FormGroup**.

FormControl

Un *FormControl* representa un solo campo de formulario. Es La unidad más pequeña dentro de los formularios Angular.

Los *FormControl* encapsulan el valor del campo y su estado, ya sea válido, sucio (cambiado) o si tiene errores.

FormGroup

La mayoría de los formularios tienen mas de un campo, por lo tanto necesitamos manejar múltiples *FormControls*. Si queremos chequear la validez de nuestro formulario, sería incómodo iterar sobre un array de *FormControls* y chequear cada uno para establecer su validez. Los *FormGroups* resuelven este problema proporcionando una interfaz que envuelve a una colección de *FormControls*.

Formularios Reactivos

Los formularios reactivos proporcionan un enfoque basado en modelos para manejar entradas de formularios cuyos valores cambian con el tiempo.

Este tipo de formularios utilizan un enfoque explícito e inmutable para gestionar el estado de un formulario en un momento dado. Cada cambio en el estado del formulario devuelve un nuevo estado, que mantiene la integridad del modelo entre los cambios.

Los Formularios Reactivos se construyen alrededor de observables, donde las entradas y valores de formularios se proporcionan como flujos de valores de entrada, a los que se puede acceder de forma sincrónica.

Los *Reactive Forms* tienen métodos que cambian el valor del valor programáticamente, el cual nos da flexibilidad para actualizar el valor sin necesidad de que haya interacción por parte del usuario.

Una instancia de *form control* tiene un método *setValue()* que actualizar el valor del control de formulario y valida la estructura del valor provisto en relación a la estructura de control.

Agrupando controles de formulario

Tal como una instancia de control de formulario nos da control sobre un campo de entrada, una instancia de *form group* rastrea el estado del formulario de un grupo de instancias de control de formulario. Cada control en una instancia de *form group* es rastreado por el nombre cuando creamos la instancia.

Veamos este ejemplo:

```
/* FormGroup */
registroForm = new FormGroup({
  primerNombre : new FormControl(''),
  apellido : new FormControl('')
});
```

Los Form Controls están agrupados en un grupo. Esta instancia de Form Group provee su valor de modelo como un objeto reducido de los valores de cada control en el grupo.

Una instancia de Form Group tienen las mismas propiedades y métodos que una instancia de Form Control.

Luego tenemos que asociar el modelo del Form Group a la vista. Un Form Group rastrea el estado y los cambios de cada uno de los controles, por lo tanto si uno de los controles cambia, el control padre emite un nuevo estado. El modelo del grupo es mantenido por sus miembros.

```
<form [formGroup]="registroForm">
  <label for="">
    Nombre:
    <input type="text" formControlName="primerNombre">
  </label>
  <label for="">
    Apellido:
    <input type="text" formControlName="apellido">
  </label>
</form>
```

En este caso tenemos un Form Group que contiene un grupo de controles, el *registroForm* es unido al formulario a través de la directiva *FormGroup*, creando una comunicación entre el modelo y el formulario que contiene los inputs. el input *formControlName*, que es provisto por la directiva *FormControlName* une cada input individual al control del formulario definido en el *FormGroup*. Para poder capturar los datos del formulario, lo que hacemos es decirle al *FormGroup* que "escuche" el evento submit y emita un evento *ngSubmit* que podemos relacionar a un método callback.

```
<form [formGroup]="registroForm" (ngSubmit)="onSubmit()">
  <label for="">
    Nombre:
    <input type="text" formControlName="primerNombre">
  </label>
  <label for="">
    Apellido:
    <input type="text" formControlName="apellido">
  </label>
</form>
```

De esta manera, el método *onSubmit()* del componente va a capturar los valores actuales del *registroForm*.

```
onSubmit(){
  console.log(this.registroForm.value);
}
```

Usamos un boton para activar el envío del formulario.

```
<button type="submit" [disabled]="!registroForm.valid">Enviar</button>
```

Formularios manejados por el template (Template-Driven forms)

Angular provee directivas de formularios específicas que podemos usar para unir los datos introducidos en el formulario con el modelo del mismo. Estas directivas de formulario específicas agregan funcionalidades y comportamientos extras al formulario HTML simple. El resultado es un

template que tiene en cuenta tanto los valores unidos al modelo como así también la validación del formulario.

Para construir un formulario vamos a seguir los siguientes pasos:

- Creamos el componente que va a controlar el formulario.
- Creamos un template con un formulario inicial.
- Unimos las propiedades de cada control del formulario usando la sintaxis *ngModel*, que me permite establecer un doble enlazamiento de datos.
- Agregamos un atributo name a cada control input del formulario.
- Agregamos reglas propias de CSS para establecer una respuesta visual
- Manejamos el envío del formulario a través del *ngSubmit*.

```
<div class="container">
 <h1>Template-Driven Form</h1>
 <div class="row">
   <div class="col-6 offset-3">
     <form #regForm='ngForm' (ngSubmit)="Registrar(regForm)">
       <h2>Registro</h2>
       <br />
       <div class="form-group">
         <input type="text" class="form-control" placeholder="Nombre"
name="nombre" ngModel required>
       </div>
       <div class="form-group">
         <input type="text" class="form-control" placeholder="Apellido"
name="apellido" ngModel>
       </div>
       <div class="form-group">
         <input type="email" class="form-control" placeholder="email"
name="email" ngModel>
       </div>
       <div class="align-center">
         <button class="btn btn-default" type="submit"
id="registrar">Registrar</button>
       </div>
     </form>
   </div>
 </div>
</div>
```

NgForm

Es la directiva con la cual podemos crear controles manejados dentro del formulario. Se le agrega a la etiqueta html *<form>*, extendiendo las características normales del formulario html.

NgModel

Cuando agregamos la Directiva NgModel a un control de formulario, estamos registrándolo dentro del NgForm.

```
<input type="email" class="form-control" id="email" placeholder="Email"
name="email" ngModel>
```

Crea una instancia de la clase *FormControl*. Este ayuda a manejar la información ingresada por el usuario, y mantiene el estado de validación del control.
La función principal tanto del NgForm como del NgModel es permitirnos recuperar todos los valores de los controles asociados al formulario y recuperar el estado general de dichos controles en relación al formulario.

```
<form # (ngSubmit)="Register(regForm)" >
```

En este caso estamos exportando el valor del **ngForm** a una variable local llamada *regForm*. Si bien esto no es necesario, a través de esta variable vamos a poder acceder a algunas de las propiedades del formulario:

- **regForm.Value**: No devuelve un objeto que contiene todos los valores de los campos del formulario.
- **regForm**.**Valid**: Nos devuelve un valor indicando si el formulario es válido o no. Si es válido, este valor es true, de lo contrario, es false.
- **regFrom.touched**: Retorna verdadero o falso cuando uno de los campos del formulario fue modificado.
- **regForm.submitted**: Chequea si el formulario fue enviado o no. Retorna true o false.

Como vemos, la etiqueta *form* no tiene ningún atributo *action*. Esto es porque estamos enlazando los campos del formulario con la clase *Componente* del template, por lo tanto los datos del formulario siempre van a ser procesados por esta clase.

Validaciones

Las validaciones son un aspecto importante en la programación. No podemos confirar en que el usuario siempre va a ingresar los datos correctamente. Por lo tanto usamos las validaciones por dos razones: para prevenir que el usuario ingrese información incorrecta y para mandarle mensajes al usuario solicitando que ingrese los datos en forma correcta.
Para hacer esto en los *template-driven forms* usamos una serie de validadores tales como *minlength, maxlength, required o email*. Lo que vamos a hacer es agregar la directiva necesaria para validar cada control del formulario.

```
<input type="text" class="form-control" placeholder="First Name"
name="firstname" required ngModel>
```

Para validar el estado del control, en relación al atributo required, vamos a hacer uso de las clases CSS que Angular aplica y remueve cada vez que el formulario varía su estado. Estas clases son las que van a ser agregadas cada vez que el estado del formulario cambie:

- *ng-touched*: El control fue "visitado".
- *ng-untouched*: El control no fue nunca "visitado".
- *ng-dirty*: El valor del control fue modificado.
- *ng-pristine*: El valor del control no varió.
- *ng-valid*: El valor del control es válido.
- *ng-invalid*: El valor del control es inválido.

Vamos a poder utilizar estas clases incluso para definir nuestros propios estilos:

```
input.ng-invalid.ng-touched
{
  border-color: red;
}
```

En este caso agregamos un borde rojo al control input cada vez que este tenga la clase *ng-invalid* y *ng-touched*.

```
<input type="email" class="form-control" id="email" placeholder="Email" name="email"
email required ngModel #email="ngModel">
<span class="help-bpx" *ngIf="email.touched && !email.valid ">Por favor ingrese un
email válido</span>
```

En este caso usamos la variable de referencia del template y se la asignamos al *NgModel*(es decir, el estado del control) a esa variable.

Es elemento <*span*> solo se va a mostrar cuando el email ingresado no sea válido. Para esto utilizamos la directiva *ngIf*. Esta directiva estructural va a mostrar el elemento cuando el email se inválido.

Validación de Formularios Reactivos

En los formularios Reactivos la fuente de la verdad es la clase componente. En lugar de agregar validadores a través de los atributos en el template, agregamos validadores directamente al modelo del *form control* en la clase componente. Angular va a llamar a esas funciones cada vez que el valor de un control cambie.

Funciones Validadoras

Hay dos tipos de funciones validadores: validadores sincrónicos y validadores asincrónicos.

- *Validadores Sincrónicos*: son aquellas funciones que toman una instancia del control e inmediatamente retornan un set de errores de validación o null en su defecto. Estas funciones se pueden pasar como segundo parámetro cada vez que instanciamos un objeto del tipo *FormControl*.
- *Validadores Asincrónicos*: son aquellas funciones que toman una instancia de control de formulario y retornan una promesa o un objeto *Observable* para luego emitir un set de

errores de validación o null en su defecto. Estas funciones pueden ser pasadas como tercer argumento cada vez que instanciamos un objeto del tipo *FormControl*.

Validadores Incorporados

Podemos o, crear nuestros propios validadores, o bien podemos utilizar los validadores ya incorporados al framework.

Los mismos validadores incorporados que están disponibles como atributos en los formularios *template-driven*, tales como *required* o *minlength*. pueden ser utilizados en las funciones de las clases validadoras. (Listado de validadores incorporados).

```
export class ValidadorReactiveComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

  registroForm = new FormGroup({
    'nombre': new FormControl('',[
      Validators.required,
      Validators.minLength(4)
    ]),
    'apellido':new FormControl('',Validators.required)
  });

  onSubmit(){
    console.log(this.registroForm.value);
  }

  get nombre() { return this.registroForm.get('nombre'); }
  get apellido() { return this.registroForm.get('apellido'); }

}
```

Como podemos observar, en el código anterior, se establecieron dos elementos del tipo *FormControl*, ambos definidos dentro de una instancia de una clase del tipo FormGroup.

El control *nombre* establece dos validadores *incorporados* al framework, **Validators.required** y **Validators.minlength**.

Notemos que ambos se tratan de validadores sincrónicos,ya que fueron ingresados como segundo argumento del constructor de la clase *FormControl*.

En el caso del control *nombre*, le pasamos más de un validador, por lo tanto lo hacemos dentro de un array. En el caso del control *apellido*, solamente le estamos pasando un validador.

Las últimas líneas del código anterior involucran a dos métodos *getter*. En los formularios reactivos, podemos acceder siempre a cualquier instancia de *FormControl* a través del método Get de su Grupo padre, pero a veces resulta útil definir getters para manejarlos desde el template.

```
<div class="container">
  <h1>Reactive Forms Validación</h1>
  <div class="row">
    <div class="col-6">
      <h2>Registro</h2>
      <br />
      <form [formGroup]="registroForm" (ngSubmit)="onSubmit()">
        <div class="form-group">
          <input type="text" class="form-control" placeholder="Nombre" formControlName="nombre">
        </div>
        <div class="form-group">
          <input type="text" class="form-control" placeholder="Apellido" formControlName="apellido">
        </div>
        <div class="align-center">
          <button class="btn btn-success" type="submit">Registrar</button>
        </div>
      </form>
    </div>
  </div>
  <div *ngIf="nombre.invalid && (nombre.dirty || nombre.touched)">
    <div class="row">
      <div class="col-6">
        <div *ngIf="nombre.errors.required" class="alert alert-danger" role="alert">
          Debe ingresar un nombre
        </div>
        <div *ngIf="nombre.errors.minlength" class="alert alert-danger" role="alert">
          El nombre debe tener al menos 4 caracteres
        </div>
      </div>
    </div>
  </div>
</div>
```

Validaciones personalizadas

Hay ocasiones donde los validadores incorporados al framework resultan insuficientes para lo que estamos queriendo hacer, y eso nos lleva a la necesidad de diseñar nuestros propios validadores. Veamos el siguiente código:

```
export function validadorTelefono(phoneExp : RegExp): ValidatorFn {
    return (control: AbstractControl): {[key: string]: any} => {
        const phone = phoneExp.test(control.value);
        return !phone ? {'phoneNumber': {value: control.value}}: null;
    }
}
```

Este validador personalizado es una función que devuelve un *validatorFn*, es decir una función validadora, necesaria para poder ser utilizada en los formularios Angular. Esta función recibe como parámetro una expresión regular, que va a ser comparado con los datos ingresados por el usuario. La función *validadorTelefono* retorna la función validadora configurada. Esta función toma un objeto del tipo ControlForm de Angular y retorna *null* si el valor de dicho control es válido, o un objeto de error de validación en caso contrario. El objeto de error de validación tiene una propiedad cuya *key* de validación es, en este caso, 'phoneNumber', y cuyo valor es un diccionario de valores arbitrarios que se podría insertar en un mensaje de error.

```
registroForm = new FormGroup({
  'nombre': new FormControl('',[
    Validators.required,
    Validators.minLength(4)
  ]),
  'apellido':new FormControl('',Validators.required),
  'telefono': new FormControl('',[
    Validators.required,
    validadorTelefono(/^[456]{1}[0-9]{9}$/)
  ])
});
```

Como vemos en el código anterior, instanciamos un objeto del tipo *FormGroup*, con varios controles del tipo *FormControl*. El el caso del control *teléfono*, este va a recibir dos validadores, uno incorporado al framework, *Validators.required*, y uno creado por nosotros, *validatorTelefono*, al que le pasamos una expresión regular para que valide que solo se puedan ingresar datos numéricos al value del control.

```
<div class="form-group">
  <input type="text" class="form-control" placeholder="Teléfono" formControlName="telefono">
</div>
```

```
<div *ngIf="telefono.invalid && (telefono.dirty || telefono.touched)">
  <div class="row">
    <div *ngIf="telefono.errors.phoneNumber" class="alert alert-danger" role="alert">
      El teléfono ingresado no es válido
    </div>
  </div>
</div>
```