Angular - Clase 4

Formularios

En esta clase veremos como se debe trabajar con Formularios en angular para llevar a cabo formularios desde simples a muy complejos, manejar estados y validaciones.

Formularios

En Angular, los formularios son una parte importante para la recopilación y gestión de datos ingresados por los usuarios. Permiten recopilar información como texto, números, fechas y selecciones de opciones, y luego procesar y enviar esos datos a través de la aplicación.

En un formulario de Angular, puedes utilizar una combinación de elementos HTML, directivas y lógica del componente para crear campos de entrada, botones de envío, validaciones y acciones asociadas.

Angular ofrece dos enfoques principales para trabajar con formularios: formularios basados en plantillas y formularios reactivos.

- **Formularios basados en plantillas:** Utilizan directivas especiales en la plantilla HTML del componente para crear y validar el formulario. Estas directivas incluyen `ngForm`, `ngModel`, `ngModelGroup`, entre otras. Los formularios basados en plantillas son más adecuados para casos simples y rápidos.
- **Formularios reactivos:** Utilizan un enfoque basado en código para construir y manipular el formulario. Se crean instancias de objetos FormGroup, FormControl y FormArray para representar el formulario y sus elementos. Los formularios reactivos ofrecen más flexibilidad y control, especialmente para formularios complejos y dinámicos.

Ambos enfoques ofrecen características como validación de datos, seguimiento de estado, manejo de eventos y envío de datos. La elección entre formularios basados en plantillas y formularios reactivos depende de las necesidades y complejidad de tu aplicación.

Formularios basados en plantillas

Los formularios basados en plantillas nos permiten crear formularios directamente sobre los elementos HTML. Para crear uno, lo primero que haremos es importar el módulo de formularios en el módulo donde estemos trabajando.

```JavaScript
import {FormsModule} from '@angular/forms';
@NgModule({
  declarations: [],
    imports: [
      FormsModule,
    ]
})
```

Ahora tenemos acceso a la directiva **NgModel**

NgModel

La directiva ngModel se utiliza para establecer un vínculo entre un elemento de entrada de formulario en la plantilla HTML y una propiedad o variable en el componente de Angular.

Cuando se utiliza la directiva ngModel, se establece un enlace bidireccional entre el valor del elemento de entrada (por ejemplo un input) y la propiedad especificada en el componente. Esto significa que cualquier cambio realizado en el elemento de entrada se reflejará inmediatamente en la propiedad del componente, y cualquier cambio realizado en la propiedad del componente se actualizará automáticamente en el elemento de entrada.

```javascript
<input type="text" [(ngModel)]="nombre">
<button (click)="imprimir()">imprimir</button>
<button (click)="eliminar()">eliminar</button>
```

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

@Component({
  selector: 'app-dummy-page',
  templateUrl: './dummy-page.component.html',
  styleUrls: ['./dummy-page.component.scss']
})
export class DummyPageComponent {
  nombre = '';
  imprimir() {
    console.log('imprimir', this.name);
  }
  eliminar() {
    this.name = '';
    console.log('eliminar', this.name)
  }
}
```

Cabe destacar que los unicos tags html que pueden aceptar el uso de NgModel son: `<input>`, `<select>` y `<textarea>`.

Tenemos entonces un input con la directiva ngModel envuelta en corchetes y paréntesis (a veces llamado “banana box”) la cual posee como argumento una variable donde los cambios que realicemos en dicho input se van a almacenar.

A medida que ingresemos valores en el campo input, dichos valores se verán reflejados en la variable *nombre*. Podemos chequear esto haciendo click en “imprimir” para ver el contenido de la variable *nombre* en la consola.

Tal como mencionamos anteriormente, los cambios se reflejan desde ambas direcciones, por lo que si hacemos uso del botón “eliminar” el cual limpia el contenido de la variable, notaremos que el valor también se eliminó de la variable.

Formularios Reactivos

Los formularios de Plantilla nos son útiles cuando tenemos un formulario muy simple el cual no requiere validaciones muy importantes. Pero en el momento que necesitamos validaciones complejas, manejo de estados, sistemas de errores o formularios complejos, nos quedamos cortos con sólo formularios basados en plantillas.

Es allí cuando necesitamos Formularios Reactivos.

Los formularios reactivos en Angular son una forma de gestionar formularios en aplicaciones web. En lugar de depender de plantillas y directivas en HTML, los formularios reactivos se construyen utilizando clases TypeScript y observables.

Los formularios reactivos se basan en el patrón Observable y permiten mejor gestión de las validaciones, seguimiento de cambios y el manejo de eventos en los formularios.

En lugar de enlazar directamente los campos del formulario al modelo de datos, los formularios reactivos utilizan una representación del estado actual del formulario con una clase llamada FormGroup. El FormGroup contiene a su vez un set de instancias de FormControl para cada campo del formulario, los cuales se encargan de manejar el estado del mismo.

Algunas características clave de los formularios reactivos en Angular incluyen:

- **FormGroup**: Representa el estado del formulario y contiene FormControl para cada campo del formulario. Los FormControl gestionan el valor y la validación de cada campo individualmente.
- **FormControl**: Representa un campo individual del formulario. Puede contener el valor actual, la validación y el estado de cambio del campo.
- **Validators**: Provee una variedad de validadores predefinidos, como required, minLength, maxLength, pattern, etc., para validar los valores de los campos.
- **Observables**: Los formularios reactivos se basan en la programación reactiva y utilizan observables para seguir los cambios en el estado del formulario y en los valores de los campos.
- **Cambios dinámicos**: Los formularios reactivos permiten realizar cambios en la estructura del formulario de manera dinámica, agregando o eliminando campos según las necesidades de la aplicación.
- **Validación personalizada**: Además de los validadores predefinidos, los formularios reactivos también admiten la creación de validadores personalizados para adaptarse a requisitos específicos.

Aplicación de Formularios Reactivos

En primer lugar, vamos a importar el módulo de formularios reactivos en el módulo donde estemos trabajando.

```javascript
import {ReactiveFormsModule} from '@angular/forms';
import {MatFormFieldModule} from '@angular/material/form-field';
@NgModule({
  declarations: [],
  imports: [
    ReactiveFormsModule,
    MatFormFieldModule,
  ]
})
```

Una vez importado, vamos a crear un FormGroup. Para ello vamos a inyectar el servicio *UntypedFormBuilder* al componente y lo vamos a usar para que éste nos cree un form group.

NOTA: También importamos *MatFormFieldModule*, pero haremos uso del mismo un poco más adelante.

```javascript
import {UntypedFormBuilder, UntypedFormGroup} from
'@angular/forms';

export class DummyComponent {
  // primero creamos la variable que contendrá al form group
  form: UntypedFormGroup;
  constructor(
    // inyectamos el servicio
    private uFormBuilder: UntypedFormBuilder,
  ) {
    // ahora creamos el grupo usando la funcion group
    this.form = this.uFormBuilder.group({
      usuario: '',
      pass: ''
    });
  }
}
```

Con esto hemos creado el formGroup y como parámetro de la función group le pasamos un objeto el cual contendrá los campos de nuestro formulario.

Como le pasamos un string vacío, toma dicho string como valor inicial.

Ahora vamos a conectar el form group con el HTML

```JavaScript
<form [formGroup]="form">
  <mat-form-field>
    <mat-label>Nombre de Usuario</mat-label>
    <input matInput placeholder="Usuario"
formControlName="usuario">
  </mat-form-field>

  <mat-form-field>
    <mat-label>Contraseña</mat-label>
    <input matInput placeholder="Contraseña"
formControlName="pass">
  </mat-form-field>
</form>
```

En la tag *<form>* definimos la directiva *formGroup* con un binding [] y hacemos referencia al formGroup. De esta manera vinculamos el formGroup con la tag Form del documento HTML y todo lo que se defina dentro de dicha TAG tendrá acceso al group.

Nuevamente para vincular los campos lo podemos hacer en un número limitado de lugares, pero con la nueva directiva es mucho más simple dado que no estamos limitados a `<input>`, `<select>` y `<textarea>`. Ahora podemos incluirla en algunos componentes que forman parte de los formularios de Angular Material y usarlos como entradas del formulario.

Para vincular un campo específico a una tag html hacemos uso de la directiva *formControlName* a través de la cual vamos a indicar el nombre del campo que definimos en el group.

Testeando el Formulario

Vamos a ver cómo se comporta nuestro formulario actualmente. Para ello disponemos de dos formas de hacerlo: A través de observables o accediendo al valor. Esto lo podemos hacer sobre el FormGroup o el FormControl.

Observables

Con el método Observables nos podemos suscribir al observable del FormGroup el cual nos ofrece un stream que notifica cada vez que hay un cambio.

```JavaScript
this.form.valueChanges.subscribe({
  next: formValue => {
    console.log(formValue);
  }
});
// {usuario: '', pass: ''}
// {usuario: 'M', pass: ''}
// {usuario: 'Ma', pass: ''}
// {usuario: 'Mar', pass: ''}
// {usuario: 'Mari', pass: ''}
// {usuario: 'Mario', pass: ''}
```

Cada vez que ingresamos una letra, el observable nos notifica el estado del objeto inicial.

Acceder al valor de forma Sincrónica

Para acceder al valor de forma sincronica (es decir sin necesidad de observables) podemos hacerlo con la función *getValue.*

```javascript
console.log(this.form.getRawValue());
// {usuario: 'Mario', pass: ''}
```

Ambos métodos son igual de útiles dependiendo de lo que necesitamos lograr. En caso de necesitar reaccionar al cambio. Si necesitamos realizar un autocompletado o una lista de sugerencias, podemos suscribirnos al valor y reaccionar cada vez que cambia. Por otro lado, si simplemente queremos el valor luego de hacer click en un botón de “Aceptar”, podemos usar el método sincrónico.

Acceder al valor del campo

Estos dos métodos, también son válidos a nivel control. En caso de que necesitemos un control específico para reaccionar u obtener su valor, lo podemos hacer llamando al método *get* del form group.

```javascript
const usuario = this.form.get('usuario');
if (usuario) {
  usuario.valueChanges.subscribe();
  usuario.getRawValue();
}
```

De esta manera tenemos acceso a los controles individuales, pero con un pequeño inconveniente. El método get nos devuelve el control pero tiene chance de devolver null en caso de que no encuentre el control que le pasamos. Es por eso que tenemos que hacer uso del IF, dado que el IDE no nos permitirá usar la variable a no ser que chequemos si es o no NULL. Para evitar este problema podemos hacer lo siguiente:

Uso de declaraciones de campos

```javascript
import {UntypedFormControl} from '@angular/forms';

// --componente
usuarioCtrl = new UntypedFormControl('');
passwordCtrl = new UntypedFormControl('');
constructor(
  private uFormBuilder: UntypedFormBuilder,
) {
  this.form = this.uFormBuilder.group({
    usuario: this.usuarioCtrl,
    pass: this.passwordCtrl,
  });
}
```

Al definir los campos en una variable y definirla inmediatamente, la misma no puede ser null, por lo que al acceder al campo, no es necesario hacer un nullcheck si no que podemos hacer uso de la variable sin más.

```javascript
this.usuarioCtrl.valueChanges.subscribe();
this.usuarioCtrl.getRawValue();
```

Validaciones

Las validaciones son parte muy importante dentro de los formularios reactivos. Gracias a ellas podemos definir una serie de reglas que deben cumplirse antes de que el formulario sea marcado como “válido”.

Disponemos de una amplia variedad de validaciones creadas por el propio angular. Algunas de ellas son:

- **Min:** Número mínimo en un input numérico.
- **Max:** Número máximo en un input numérico.
- **Required:** Marca el campo como requerido, por lo que debe poseer al menos un valor de cualquier tipo.
- **Email:** El texto ingresado debe tener formato de Mail.
- **MinLength:** Número mínimo de caracteres en un input de texto.
- **MaxLength:** Número máximo de caracteres en un input de texto.
- **Pattern:** Permite ingresar una expresión regular y es válida si dicha expresión regular se cumple.

Con esta serie de validaciones podemos cumplir la mayoría de requisitos que nos vamos a encontrar en el día a día. En caso de no ser así, también podemos crear nuestras propias validaciones.

Aplicando validaciones

Vamos a ver las validaciones en acción. Para ello debemos declararlas como un array en el segundo parámetro al crear el form control:

```javascript
usuarioCtrl = new UntypedFormControl('', [Validators.required]);
passwordCtrl = new UntypedFormControl('', [Validators.required]);
```

Y vamos a añadir un botón en el HTML

```javascript
<form [formGroup]="form">
  <mat-form-field>
    <mat-label>Nombre de Usuario</mat-label>
    <input matInput placeholder="Usuario" formControlName="usuario">
  </mat-form-field>

  <mat-form-field>
    <mat-label>Contraseña</mat-label>
    <input matInput placeholder="Contraseña" formControlName="pass">
  </mat-form-field>
  <button type="submit" [disabled]="form.invalid"></button>
</form>
```

Con esto, el botón estará deshabilitado a no ser que el formulario esté marcado como válido. ¿Cómo funciona esto?

Los controles poseen una serie de propiedades las cuales nos ayudan a definir lógicas de programación y nos permiten conocer el estado actual del mismo. Cada control posee una propiedad llamada valid y otra invalid.

A través de ellas, podemos conocer el estado actual del control en relación a las validaciones. Cuando una validación NO se cumple, la propiedad Valid se marca como false mientras que la propiedad invalid se marca como true.

Ésto se replica a nivel formGroup. Éste también posee dichas propiedades, solo que estas se marcan respectivamente como true o false dependiendo de si posee uno o más controles marcados como inválidos.

Es así que podemos conocer el estado de la validez de un control en específico o bien el formulario en general tal como vimos en la propiedad Disabled del botón.

Veamos un ejemplo distinto

JavaScript

```
  usuarioCtrl = new UntypedFormControl('', [Validators.required,
Validators.minLength(8), Validators.maxLength(16)]);
  passwordCtrl = new UntypedFormControl('', [Validators.required,
Validators.minLength(8), Validators.maxLength(16)]);
  mailCtrl = new UntypedFormControl('', [Validators.required,
Validators.email]);
  edadCtrl = new UntypedFormControl('', [Validators.required,
Validators.min(13)]);
  constructor(
    private uFormBuilder: UntypedFormBuilder,
  ) {
    this.form = this.uFormBuilder.group({
      usuario: this.usuarioCtrl,
      pass: this.passwordCtrl,
      mail: this.mailCtrl,
      edad: this.edadCtrl,
    });
  }
```

JavaScript

```
<form [formGroup]="form" class="form">
  <mat-form-field>
    <mat-label>Nombre de Usuario</mat-label>
    <input matInput placeholder="Usuario" formControlName="usuario">
  </mat-form-field>

  <mat-form-field>
    <mat-label>Contraseña</mat-label>
    <input matInput placeholder="Contraseña" formControlName="pass">
  </mat-form-field>

  <mat-form-field>
    <mat-label>Mail</mat-label>
    <input matInput placeholder="Mail" formControlName="mail">
  </mat-form-field>

  <mat-form-field>
    <mat-label>Edad</mat-label>
    <input matInput placeholder="edad" formControlName="edad"
type="number">
  </mat-form-field>

  <button type="submit" [disabled]="form.invalid">Crear Usuario</button>
</form>
```

JavaScript

```
.form {
  display: flex;
  flex-direction: column;
  gap: 10px;
  width: 400px;
  padding: 20px;
}
```

Hemos creado un formulario muy básico de registro el cual nos permite ingresar los datos del usuario validando varias cosas.

En el campo de usuario validamos que se ingrese algún valor y estos sean de mínimo 8 y máximo 16 caracteres. Lo mismo hacemos para la contraseña. En el campo de email simplemente validamos que tenga formato de email y finalmente en la edad solicitamos que sea como mínimo 13.

Si no se cumplen los requisitos del formulario, el botón de registro permanecerá inactivo.

Estados de un formulario

Como mencionamos anteriormente, un formulario posee muchos estados los cuales se pueden utilizar para identificar el estado del formulario en si. Estos pueden ser:

- Pristine (prístino): Un FormControl se considera prístino cuando no ha cambiado su valor desde que se creó o se restableció por última vez. Es el estado inicial de un control que aún no ha sido interactuado por el usuario.
- Dirty (modificado): Un FormControl se considera modificado cuando ha cambiado su valor desde que se creó o se restableció por última vez. Indica que el usuario ha interactuado con el control y ha realizado cambios.
- Touched (tocado): Un FormControl se considera tocado cuando ha sido enfocado y luego se ha desenfocado. Indica que el usuario ha interactuado con el control al tocarlo, pero no necesariamente ha realizado cambios.
- Untouched (no tocado): Un FormControl se considera no tocado cuando no ha sido enfocado y luego desenfocado. Es el estado inicial de un control que aún no ha sido interactuado por el usuario.

- Valid (válido): Un FormControl se considera válido cuando pasa todas las validaciones definidas en él. Es decir, cumple con todas las restricciones de validación establecidas, como longitud mínima, formato, entre otros.
- Invalid (inválido): Un FormControl se considera inválido cuando no cumple con al menos una de las validaciones definidas en él. Indica que el valor ingresado no satisface una o más restricciones de validación.
- Pending (pendiente): Un FormControl se considera pendiente cuando está en proceso de validación asíncrona. Esto ocurre cuando se realiza una validación asíncrona, como una petición HTTP, y aún no ha obtenido una respuesta.
- Disabled (deshabilitado): Un FormControl se considera deshabilitado cuando se desactiva explícitamente mediante la metodo disable() del control. Un control deshabilitado no se puede editar ni enviar como parte del formulario.

Aquí tenemos un ejemplo del uso de los estados

```JavaScript
  <mat-form-field>
    <mat-label>Nombre de Usuario</mat-label>
    <input matInput placeholder="Usuario" formControlName="usuario">
    <mat-error *ngIf="usuarioCtrl.touched && usuarioCtrl.invalid">El
campo debe contener entre 8 y 16 caracteres</mat-error>
  </mat-form-field>
```

Deshabilitando un campo

En caso de que necesitemos deshabilitar el uso de un campo de formulario, podemos hacerlo de la siguiente manera.

Al crear un campo:

```JavaScript
usuarioCtrl = new UntypedFormControl({value: '', disabled: true});
```

Si necesitamos deshabilitar o habilitar el campo luego de crearlo.

```JavaScript
this.usuarioCtrl.disable();
this.usuarioCtrl.enable();
```

Typed forms

Como podemos observar en los ejemplos anteriores, los nombres de las clases poseen como prefijo “Untyped”. Esto se debe a que en la versión 15 de angular se añadieron lo que se denomina “Typed forms” los cuales se convirtieron en el estándar.

Los “typed forms” son un nuevo tipo de formulario el cual permite que se le añada un tipo de dato a los formularios para que no utilicen como tipo de dato “any” o “unknown”.

El uso de dichos formularios es igual solo que se posee mayor flexibilidad a la hora de usarlos dado que poseemos tipos de datos cuando trabajamos con los valores.

Este tipo de formularios están más allá del alcance de este curso dado que es algo muy nuevo y la mayoría de proyectos anteriores siguen con la antigua manera de escribir formularios.