8- Reactive : Formularios reactivos con Angular

El **doble enlace automático** entre elementos *html* y propiedades de objetos fue el primer gran éxito de **Angular**. Ese *doble-binding* facilita mucho el desarrollo de formularios. Pero esa magia tienen un coste en escalabilidad; impacta en el tiempo de ejecución y además dificulta la validación y el mantenimiento de formularios complejos.

La solución en Angular 7 pasa por desacoplar el modelo y la vista, introduciendo una capa que gestione ese doble enlace. Los servicios y directivas del módulo `ReactiveFormsModule` que viene en la librería `@angular/forms` permiten programar **formularios reactivos conducidos por el código**.

Partiendo de la aplicación tal cómo quedó en Vigilancia y seguridad en Angular. Al finalizar tendrás una aplicación con formularios *model driven* fáciles de mantener y validar.

> Código asociado a este artículo en *GitHub*: AcademiaBinaria/angular-board/8-reactive
>
> > Tienes una versión desplegada operativa para probar Angular Board

1 Desacople entre vista y modelo

La directiva `[(ngModel)]="model.property"` con su popular *banana in a box* establece el doble enlace entre el elemento de la vista al que se le aplica y una propiedad del modelo. Los cambios en la vista son trasladados automáticamente al modelo, y al revés; cualquier cambio en el modelo se refleja inmediatamente en la vista.

Se pueden establecer validaciones y configurar los eventos que disparan las actualizaciones; pero todo ello usando más y más atributos y directivas en la plantilla. Son los formularios *template driven* que degeneran en un *html* farragoso y difícil de mantener.

> Vamos a crear un formulario de registro de usuarios usando los formularios *model driven*. Para ello voy a crear el módulo *security* con el componente *register* y los engancharé a la ruta *security/register*.

El módulo de seguridad y su configuración lazy en el raíz:

```
ng g m security --routing true
ng g c security/register
```

```
{
  path: 'security',
  loadChildren: './security/security.module#SecurityModule'
},
```

El componente de registro y su ruta asignada:

```
const routes: Routes = [
  {
    path: 'register',
    component: RegisterComponent
  },
  {
    path: '**',
    redirectTo: 'register'
  }
];
```

```
<a routerLink="security/register" class="button">Register</a>
```

1.1 Form Builder

Entra en acción el `FormBuilder`, un servicio del que han de depender los componentes que quieran desacoplar el modelo de la vista. Se usa para construir un formulario creando un `FormGroup`, (un grupo de controles) que realiza un seguimiento del valor y estado de cambio y validez de los datos.

Para poder usarlo tenemos que importar el módulo de Angular en el que viene declarado, el `ReactiveFormModule`.

```
import { ReactiveFormsModule } from '@angular/forms';
@NgModule({
  declarations: [RegisterComponent],
  imports: [
    CommonModule,
    SecurityRoutingModule,
    ReactiveFormsModule
  ]
})
export class SecurityModule { }
```

Veamos un ejemplo mínimo de su declaración en `register.component.ts`.

```
export class RegisterComponent implements OnInit {
  public formGroup: FormGroup;

  constructor( private formBuilder: FormBuilder ) { }

  public ngOnInit() {
    this.buildForm();
  }
  private buildForm(){
    this.formGroup = this.formBuilder.group({});
```

```
  }
}
```

1.2 Form control

El formulario se define como un **grupo de controles**. Cada control tendrá un nombre y una configuración. Esa definición permite establecer un valor inicial al control.

```
private buildForm() {
  const dateLength = 10;
  const today = new Date().toISOString().substring(0, dateLength);
  const name = 'JOHN DOE';
  this.formGroup = this.formBuilder.group({
    registeredOn: today,
    name: name.toLowerCase(),
    email: 'john@angular.io',
    password: ''
  });
}
```

Como ves, es fácil asignar valores por defecto. Incluso es un buen momento para modificar o transformar datos previos para ajustarlos a cómo los verá el usuario; sin necesidad de cambiar los datos de base.

1.3 Form view

Mientras tanto en la vista html... Este trabajo previo y extra que tienes que hacer en el controlador se recompensa con una **mayor limpieza en la vista**. Lo único necesario será asignar por nombre el elemento html con el control *typescript* que lo gestionará.

> Para ello usaremos dos directivas que vienen dentro del módulo *reactivo* son `[formGroup]="objetoFormulario"` para el formulario en su conjunto, y `formControlName="nombreDelControl"` para cada control.

```
<form [formGroup]="formGroup">
  <label for="registeredOn">Registered On</label>
  <input name="registeredOn"
         formControlName="registeredOn"
         type="date" />
  <label for="name">Name</label>
  <input name="name"
         formControlName="name"
         type="text" />
  <label for="email">E-mail</label>
  <input name="email"
         formControlName="email"
         type="email" />
  <label for="password">Password</label>
  <input name="password"
```

```
        formControlName="password"
        type="password" />
</form>
```

2 Validación y estados

La validación es una pieza clave de la entrada de datos en cualquier aplicación. Es el primer **frente de defensa ante errores de usuarios**; ya sean involuntarios o deliberados.

Dichas validaciones se solían realizar agregando atributos html tales como el archiconocido `required`. Pero todo eso ahora se traslada a la configuración de cada control, donde podrás establecer una o varias reglas de validación sin mancharte con html.

2.1 Validadores predefinidos y personalizados

De nuevo tienes distintas sobrecargas que te permiten resolver limpiamente casos sencillos de una sola validación, o usar baterías de reglas que vienen predefinidas como funciones en el objeto `Validators` del *framework*.

```
private buildForm() {
  const dateLength = 10;
  const today = new Date().toISOString().substring(0, dateLength);
  const name = 'JOHN DOE';
  const minPassLength = 4;
  this.formGroup = this.formBuilder.group({
    registeredOn: today,
    name: [name.toLowerCase(), Validators.required],
    email: ['john@angular.io', [
      Validators.required, Validators.email
    ]],
    password: ['', [
      Validators.required, Validators.minLength(minPassLength)
    ]]
  });
}
```

A estas validaciones integradas se puede añadir otras creadas por el programador. Incluso con ejecución asíncrona para validaciones realizadas en el servidor.

Por ejemplo podemos agregar una validación específica a las contraseñas

```
password: ['', [
  Validators.required,
  Validators.minLength(minPassLength),
  this.validatePassword
]]
```

Lo único que se necesita es una función que recibe como argumento el control a validar. El resultado debe ser un `null` si todo va bien. Y cualquier otra cosa en caso de fallo. Te muestro una propuesta para que puedas crear tus propios validadores.

```
private validatePassword(control: AbstractControl) {
  const password = control.value;
  let error = null;
  if (!password.includes('$')) {
    error = { ...error, dollar: 'needs a dollar symbol' };
  }
  if (!parseFloat(password[0])) {
    error = { ...error, number: 'must start with a number' };
  }
  return error;
}
```

2.2 Estados de cambio y validación

Una vez establecidas las reglas, es hora de aplicarlas y avisar al usuario en caso de que se incumplan. Los formularios y controles reactivos están gestionados por **máquinas de estados** que determinan en todo momento la situación de cada control y del formulario en si mismo.

2.2.1 Estados de validación

Al establecer una o más reglas para uno o más controles activamos el sistema de chequeo y control del estado de cada control y del formulario en su conjunto.

La máquina de estados de validación contempla los siguientes estados mutuamente excluyentes:

- **VALID**: el control ha pasado todos los chequeos
- **INVALID**: el control ha fallado al menos en una regla.
- **PENDING**: el control está en medio de un proceso de validación
- **DISABLED**: el control está desactivado y exento de validación

Cuando un control incumple con alguna regla de validación, estas se reflejan en su propiedad `errors` que será un objeto con una propiedad por cada regla insatisfecha y un valor o mensaje de ayuda guardado en dicha propiedad.

2.2.2 Estados de modificación

Los controles, y el formulario, se someten a otra máquina de estados que monitoriza el valor del control y sus cambios.

La máquina de estados de cambio contempla entre otros los siguientes:

- **PRINSTINE**: el valor del control no ha sido cambiado por el usuario
- **DIRTY**: el usuario ha modificado el valor del control.
- **TOUCHED**: el usuario ha tocado el control lanzando un evento `blur` al salir.
- **UNTOUCHED**: el usuario no ha tocado y salido del control lanzando ningún evento `blur`.

Como en el caso de los estados de validación, el formulario también se somete a estos estados en función de cómo estén sus controles.

Veamos su aplicación primero en el caso general del formulario. Uno de los usos más inmediatos es deshabilitar el botón de envío cuando la validación de algún control falla.

```
<button (click)="register()"
    [disabled]="formGroup.invalid">Register me!</button>
```

> Por cierto, este sistema de gestión de los controles del formulario oculta la parte más valiosa (el valor que se pretende almacenar) en la propiedad `value` del formulario. Contendrá un objeto con las mismas propiedades usadas durante la definición del formulario, cada una con el valor actual del control asociado.

```
public register() {
  const user = this.formGroup.value;
  console.log(user);
}
```

La validación particular para cada control permite informar al usuario del fallo concreto. Es una **buena práctica de usabilidad** el esperar a que edite un control antes de mostrarle el fallo. Y también es muy habitual usar la misma estrategia para cada control.

Lo que no queremos es llevar de vuelta la lógica a la vista; así que lo recomendado es crear una **función auxiliar para mostrar los errores** de validación.

```
public getError(controlName: string): string {
  let error = '';
  const control = this.formGroup.get(controlName);
  if (control.touched && control.errors != null) {
    error = JSON.stringify(control.errors);
  }
  return error;
}
```

En la vista colocaremos adecuadamente los mensajes para facilitarle la corrección al usuario.

```
<span>{{ getError('name')}}</span>
<span>{{ getError('email')}}</span>
<span>{{ getError('password')}}</span>
```

Ya tenemos formularios reactivos conducidos por los datos que te permitirán construir pantallas complejas manteniendo el control en el modelo y dejando la vista despejada. Como resumen podemos decir que vamos a programar más en TypeScript que en Html. La ventaja del desacople es que podremos controlar lo que

enviamos y recibimos de la vista. Así se pueden aplicar formatos, validaciones y transformaciones entre lo que presentamos y lo que enviamos hacia los servicios.

3. Un gestor de credenciales

Vamos a provechar el conocimiento sobre *Interceptores* y *Observables* para montar un pequeño sistema de gestión de credenciales. La idea es detectar respuestas a llamadas no autenticadas y redirigir al usuario a nuestra pantalla de registro.

Si el usuario se registra correctamente recibiremos un *token* que lo identifica. Lo que haremos será guardarlo y usarlo en el resto de las llamadas.

3.1 Detección y redirección de intrusos

Empezaré por crea un servicio para el interceptor de fallos de seguridad y un componente que invocará deliberadamente a un API protegida.

```
ng g s security/auth-interceptor
ng g c security/secret
```

Al componente *secret* le asignaremos la ruta *security/secret*

```
  <a routerLink="security/register" class="button">Register</a>
```

```
const routes: Routes = [
  {
    path: 'register',
    component: RegisterComponent
  },
  {
    path: 'secret',
    component: SecretComponent
  },
  {
    path: '**',
    redirectTo: 'secret'
  }
];
```

En cuanto al interceptor, haremos como ya hemos visto en el tema 7-watch. Lo primero será proveerlo invirtiendo el control que nos cede el `HttpClient`.

```
@NgModule({
  declarations: [RegisterComponent, SecretComponent],
```

```
  imports: [CommonModule, SecurityRoutingModule, ReactiveFormsModule,
HttpClientModule],
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptorService,
      multi: true
    }
  ]
})
export class SecurityModule {}
```

Luego codificaré la implementación de la interfaz `HttpInterceptor`. En este caso me interesan las respuestas con **error 401**. Emplearé el `Router` de Angular para obligar al usuario a visitar la página de registro cuando esto ocurra.

```
export class AuthInterceptorService implements HttpInterceptor {
  constructor(private router: Router) {}

  public intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
    return next.handle(req).pipe(catchError(this.handleError.bind(this)));
  }
  private handleError(err) {
    const unauthorized_code = 401;
    if (err instanceof HttpErrorResponse) {
      if (err.status === unauthorized_code) {
        this.router.navigate(['security/register']);
      }
    }
    return throwError(err);
  }
}
```

3.2 Almacenamiento y uso del token

Antes de nada volveremos al tema de los observables y su uso como **intermediarios entre objetos desacoplados**. Para ello crearé un servicio para almacenar y distribuir el *token* de identificación de usuarios.

```
ng g s security/token_store
```

```
export class TokenStoreService {
  private token = '';
  private token$ = new BehaviorSubject<string>('');

  constructor() {}
```

```
  public select$ = () => this.token$.asObservable();
  public dispatch(token) {
    this.token = token;
    this.token$.next(this.token);
  }
}
```

De vuelta en el componente `RegisterComponent`. Tenemos que enviar las credenciales al API para su aceptación. Si todo va bien, nos devolverán un *token* que identifica al usuario. Es momento de usar al `TokenStore` para transmitir la noticia por toda la aplicación.

```
public register() {
  const url = 'https://api-base.herokuapp.com/api/pub/credentials/registration';
  const user = this.formGroup.value;
  this.httpClient.post<any>(url, user)
    .subscribe(res => this.tokenStore.dispatch(res.token));
}
```

Por último volvemos al `AuthInterceptorService` en el que nos suscribiremos a los cambios acerca del *token*.

```
private token = '';
constructor(private router: Router, private tokenStore: TokenStoreService) {
  this.tokenStore.select$()
    .subscribe(token => (this.token = token));
}
```

Y lo usaremos en las cabeceras de todas las llamadas.

```
public intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
  const authHeader = { Authorization: 'bearer ' + this.token };
  const authReq = req.clone({ setHeaders: authHeader });
  return next.handle(authReq)
    .pipe(catchError(this.handleError.bind(this)));
}
```

Con este conocimiento ya casi finalizas tu introducción a Angular. Próximamente veremos una parte de su ecosistema como es *Material Design con Angular* y aprenderás más cosas para programar con Angular 7.

> Aprender, programar, disfrutar, repetir. -- *Saludos, Alberto Basalo*