Angular - Clase 5

Pipes, Directivas, Subject y Behaviour Subject

En esta clase veremos diversas utilidades de Angular que nos facilitan la vida a la hora de mostrar contenido dinámico en nuestras templates (pipes), modificar el DOM en runtime (directivas) y controlar el estado de nuestra aplicación o definir un sistema de estados al cual se puede reaccionar (subject y behavior subject).

Pipes

Los pipes de template son una forma de transformar los valores en tus plantillas antes de mostrarlos. Los pipes se utilizan para formatear datos de manera legible, como convertir una fecha en un formato específico, aplicar moneda o cambiar el texto a mayúsculas o minúsculas.

Angular proporciona una serie de pipes integrados, como **DatePipe**, **UpperCasePipe**, **LowerCasePipe**, **CurrencyPipe**, entre otros. También puedes crear tus propios pipes personalizados para adaptarse a tus necesidades específicas.

Veamos un ejemplo:

```javascript
<p>{{ fecha | date:'dd/MM/yyyy' }}</p>
```

En el template, al hacer una interpolación, usamos el símbolo de pipe o barra vertical “|” luego de la variable que queremos modificar. En el ejemplo tenemos una variable fecha seguida del pipe de angular “date” y en caso de querer pasar un argumento lo hacemos con los dos puntos, seguido del argumento. En este caso le estamos pasando el formato de fecha que queremos usar.

Puede que estés pensando que el uso de pipes es innecesario si simplemente puedo crear una función que haga lo mismo, lo cual es cierto, pero las pipes tienen un uso más interesante que solo modificar un valor.

Las pipes almacenan los valores de entrada en caché, por lo que en caso de repetirse un valor de entrada, en lugar de procesarse, el valor devuelto será el almacenado en caché. Esto implica una mejora de performance dado que hay código que no se ejecuta múltiples veces en caso de repetirse la entrada.

Otra facilidad que ofrece es en caso de tener algo específico que transformar, como una fecha, nos permite no repetir código innecesariamente cuando podemos implementarlo solo una vez y utilizarlo siempre que se lo requiera.

Vamos a crear una Pipe Personalizada

Ejecutamos el comando:

`ng generate pipe path/nombre`

Una vez generada la pipe veremos algo como lo siguiente:

```javascript
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'dummy'
})
export class DummyPipe implements PipeTransform {
  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }
}
```

Tenemos en primer lugar un decorador *@Pipe()* el cual marca la clase como “Pipe”. Luego implementamos una función transform la cual toma un valor el cual es el valor que recibe en el html y un array de argumentos.

La función luego retorna un valor el cual va a mostrarse en el template.

Vamos a modificarla para devolver un string en base a la edad de una persona para indicar si es menor o mayor de edad. Y un argumento el cual modificará el string para hacerlo con una palabra o con varias.

```JavaScript
export class EdadPipe implements PipeTransform {
  transform(edad: number, formatoLargo: boolean = true): string {
    let edadFinal = edad >= 18 ? 'Mayor' : 'Menor';
    if (formatoLargo) {
      edadFinal = `${edadFinal} de edad`;
    }
    return edadFinal;
  }
}
```

Lo primero que modificamos son los tipos de datos. El valor va a recibir un number, el formato es opcional dado que definimos un valor y recibe un boolean para indicar si queremos formato largo o corto.

Procesamos la edad para saber si es mayor o menor de edad (se usó un operador ternario).

Y al final dependiendo si queremos un formato largo, añadimos el resto de palabras, para finalmente retornar la variable.

Como podemos observar, siempre y cuando tengamos dos argumentos, podemos modificar la función a nuestra conveniencia para devolver lo que necesitemos.

También podemos encadenar pipes con el valor devuelto por la última pipe.

```JavaScript
<p>{{edad | edadPipe: false | uppercase}}</p>
// MAYOR
<p>{{edad | edadPipe| uppercase}}</p>
// MAYOR DE EDAD
<p>{{edad | edadPipe}}</p>
// Mayor de edad
```

Directivas

Las directivas son una funcionalidad de Angular que nos permite extender las funciones de las tags HTML. Las mismas se añaden como atributos HTML a las tags y desde allí se controla su comportamiento.

Ya conocemos algunas directivas: *NgIf y *NgFor. Estas directivas se conocen como directivas estructurales, dado que modifican la estructura del DOM añadiendo o quitando tags HTML o bien modificando la estructura del árbol de elementos.

Las directivas que no ejecutan cambios en el árbol de elementos se conocen como Directivas de atributo. Estas directivas sirven para cosas como extender la funcionalidad de un elemento o por ejemplo modificar sus estilos.

Para crear una directiva vamos a ejecutar:

`ng generate directive path/nombre`

Veremos lo siguiente:

```JavaScript
import { Directive } from '@angular/core';
@Directive({
  selector: '[appShadow]'
})
export class ShadowDirective {
  constructor() { }
}
```

Aquí podremos realizar todas las acciones que necesitemos. Para ello en primer lugar debemos referenciar al elemento que es el que posee la directiva entre sus atributos.

En este ejemplo veremos una directiva que añade una sombra al elemento que referencia.

```JavaScript
import {Directive, ElementRef} from '@angular/core';
@Directive({
  selector: '[appShadow]'
})
export class ShadowDirective {
  constructor(
    private el: ElementRef
  ) {
    el.nativeElement.boxShadow = '5px 5px 8px 0px rgba(0,0,0,0.2)'
  }
}
```

Para aplicar esta directiva vamos a incluirla en un elemento.

```JavaScript
<button appShadow>Aceptar</button>
```

El nombre con el cual vamos a incluir esta directiva es igual al visto en la propiedad *selector* del decorador *@Directive* al inicio de la clase de la directiva.

Si extendemos esta directiva con más estilos, podemos crear un set de directivas el cual modifiquen los botones según un estándar de diseño.

Pero el poder de una directiva no se limita sólo a estilos. Vamos a crear una directiva que registre mediante un evento, cuando toquemos la tecla enter.

```JavaScript
import {Directive, EventEmitter, HostListener, Output} from
'@angular/core';
@Directive({
  selector: '[appEnterKey]'
})
export class EnterKeyDirective {
  @Output() enterKey = new EventEmitter<void>();
  constructor() { }

  @HostListener('keydown.enter')
  onEnter() {
    this.enterKey.emit();
  }
}
```

Para esto necesitamos hacer uso de un decorador nuevo: *@HostListener* el cual nos permite “escuchar” eventos del elemento que invocó la directiva. Acepta uno o varios argumentos, que indican el nombre del evento o los eventos a los que se desea escuchar.

Cuando el evento se dispara, el decorador ejecuta la función que se encuentra debajo del mismo. Dicha función emite un evento al cual nos suscribiremos en el HTML de la siguiente manera.

```JavaScript
<input type="text" appEnterKey (enterKey)="onEnterKey()">
```

Con esto, tendremos una directiva que nos permite reaccionar al momento en que el usuario presiona la tecla “Enter”. Podemos darle uso por ejemplo en el login luego de completar el usuario o la contraseña o para concretar una búsqueda en una barra de búsqueda.

Subject y BehaviorSubject

Estas son clases de RXJS que funcionan como observador y observable a la vez con las cuales podemos crear observables propios con el objetivo de suscribirnos a un evento específico o bien crear una “máquina de estados” con la cual podemos verificar el estado de una característica particular de nuestra app.

Subject

Los subject funcionan como un observable emitiendo un evento next con el valor que le proporcionamos y permitiéndonos suscribirnos al mismo.

```javascript
import { Subject } from 'rxjs';

// Crear un Subject
const subject = new Subject<number>();

// Suscribirse al Subject
subject.subscribe(value => console.log('Valor recibido:',
value));

// Emitir valores en el Subject
subject.next(1);
subject.next(2);
subject.next(3);
```

Como podemos ver, al crear un subject, este nos proporciona una función *next()* la cual podemos utilizar para emitir un evento next con su respectivo valor. Todos los suscriptores del subject, ejecutarán el evento next. En este caso veremos una serie de logs en la consola.

Un ejemplo común de uso de un subject es detectar un cambio de estado de nuestra plataforma.

Supongamos que tenemos una página de visualización del perfil de un usuario. Ejecutamos un cambio pero para que este se refleje debemos recargar al usuario. Para ello vamos a suscribirnos a un subject que se encargue de recargar al usuario siempre que se lo requiera.

JavaScript

```
export class UserComponent {
  subjectRecarga = new Subject<object>();
  constructor(
  ) {
    this.subjectRecarga.subscribe({
      next: user => {
        this.cargarUsuario()
      }
    })
  }

  cargarUsuario() {
    // codigo para cargar los datos del usuario
  }

  modificarUsuario() {
    this.userService.modificarUsuario().subscribe({
      next: user => {
        this.subjectRecarga.next(user);
      }
    })
  }
}
```

Vamos linea por linea:

En primer lugar vemos la declaración del subject. Podemos indicar el tipo de dato con un genérico, si no lo hacemos por defecto es de tipo ***unknown***.

En el constructor nos suscribimos al subject para recargar al usuario tal como lo haríamos con un observable.

Luego tenemos la propia función que recarga los datos de usuario.

Finalmente tenemos una función que modifica los datos del usuario, llamando a la función *modificarUsuario()* de un hipotético servicio de usuarios. Devuelve un observable al que nos suscribimos y en el evento next de dicha modificación disparamos un evento next del subject indicando una modificación exitosa por lo que se debe recargar los datos de usuario.

BehaviorSubject

Los behavior subject son un subtipo de la clase Subject el cual posee una diferencia clave el cual es que puede almacenar el último valor emitido.

Esto nos ayuda en casos en los cuales el último valor emitido es relevante. Pero es más fácil de entender con un ejemplo:

Supongamos que tenemos una app de ventas y un icono de un carrito el cual posee el número de elementos en el carrito. Dependiendo del número de elementos en el mismo ese número puede aumentar o disminuir.

Nos suscribimos al BehaviorSubject del servicio de carrito el cual nos provee el número a mostrar.

JavaScript

```
export class CarritoCantidadComponent {

  numeroDeElementos = 0;

  constructor(
    private carritoService: CarritoService
  ) {
    carritoService.numeroDeElementos.subscribe({
      next: num => {
        this.numeroDeElementos = num;
      }
    });
  }
}
```

Un BehaviorSubject emite un evento next inmediatamente después de suscribirnos al mismo con el último valor que se emitió, por lo que siempre tendremos disponible el número de elementos en el carrito.

El servicio se vería así:

```JavaScript
export class CarritoService {

  public numeroDeElementos = new BehaviorSubject<number>(0);
  private carrito: Producto[] = [];

  constructor() {}

  sumarItem(item: Producto) {
    this.carrito.push(item);
    this.numeroDeElementos.next(this.carrito.length);
  }
}
```

Tanto subject como BehaviorSubject nos sirven en ocasiones de que tenemos que reaccionar a algún evento. Son utilidades que nos van a servir para escribir código más limpio y evitar crear lógicas más complejas cuando podemos usar un sistema de observables.

Código limpio

Normalmente no se expone un subject o behaviorSubject como tal dado que eso le da acceso a otras partes del código a los eventos next, lo cual implica que hay código que debería estar exclusivamente en el servicio desperdigado por la aplicación. Es por esto que lo que se hace para evitar este problema es exportarlo como un observable:

```javascript
private numero = new BehaviorSubject<number>(0);
public numero$ = this.numero.asObservable();
```

De esta manera tenemos el BehaviorSubject como privado, teniendo acceso al mismo solo dentro del servicio, y exponemos el mismo BehaviorSubject al resto de la aplicación pero como un observable puro, el cual NO tiene acceso a las funciones Next, Error o Complete. Esto obliga a mantener el código pertinente al BehaviorSubject en el servicio y que el resto de la aplicación solo pueda suscribirse al mismo mediante el observable expuesto o bien llamar a una función del servicio que modifique el BehaviorSubject.