Componentes dinámicos, directivas y pipes con Angular

El sistema de componentes de Angular te permite reutilizar pequeñas unidades para montar grandes y complejas interfaces. Eso es bueno porque la capa de presentación se *come* la mayor parte del tiempo de desarrollo. Pero podemos ir más allá.

> Podemos crear componentes con estructura fija y contenido dinámico. Podemos dar funcionalidad extra mediante directivas y podemos crear pipes para transformar la presentación.

Y todo esto de una forma sencilla y familiar usando el CLI. No hay excusa para no reutilizar código en la capa de presentación. *Don't repeat Yourself!*.

Partiendo de la aplicación tal cómo quedó en Detección del cambio en Angular. Al finalizar tendrás una aplicación que muestra información sobre productos de manera flexible y al gusto del programador.

> Código asociado a este tutorial en *GitHub*: AcademiaBinaria/angular-boss

1 Plantillas de contenido dinámico

```
As a: customer,
  I want: to see a product card with price in euros
  so that: i can decide to purchase it or not

As a: seller,
  I want: to see a product card with stock
  so that: I can ask for more or not
```

En muchas ocasiones tenemos que presentar la misma información pero con ligeros cambios. Por ejemplo una vista a veces con unos botones a veces con un texto.

En otras ocasiones mostramos información muy diferente pero con la misma estructura. Por ejemplo páginas de búsqueda; pueden ser de productos, clientes, tareas... pero siempre con la parte de filtro y la del listado.

Algunas veces se puede resolver con un `*ngIf`. Pero hay ocasiones que donde la técnica del `ng-content` te puede ser mucho más útil. Como ejemplo voy a crear un componente base para mostrar una ficha de producto. Después usaré esa plantilla para *rellenarla* según las especificaciones de cada pantalla.

1.1 Un componente común

```
ng g c product-template --project=products --module=products.module.ts --export
```

> La visión del comprador y del vendedor es parecida. Mantenemos estructura, inyectamos contenido.

```
<article style="margin: 5px; padding: 5px; border: 2px; border-style: solid;">
  <header>
    <h2>
      {{ product.description }}
    </h2>
  </header>
  <main>
    <ng-content select="main"></ng-content>
  </main>
  <footer style="margin-top: 5px">
    <ng-content select=".actions"></ng-content>
  </footer>
</article>
```

La directiva `ngContent` permite crear *slots* para incrustar contenido a voluntad del consumidor. Cada *slot* se identifica mediante un `select="css-selector"`.

Los selectores pueden referirse a elementos *html*, clases *css* o identificadores. En la práctica es como dejar unos sitios en los que inyectar contenido. Y para referirse a esos sitios les damos un nombre selector.

1.2 Implementaciones distintas

A partir de aquí podría haber múltiples componentes que usasen la misma plantilla. Voy a mostrar uno de ejemplo.

```
ng g c catalog/product --project=shop
```

La etiqueta `<main>` y el atributo `class="actions"` son usados para seleccionar los slots en los que serán inyectados. Por lo demás el componente `ab-products-product-template` se comporta como cualquier otro pudiendo recibir argumentos...

`apps\shop\src\app\catalog\catalog.component.html`

```
<ab-products-product-template [product]="product">
  <main>
    <div>
      {{ product.brand }} - {{ product.category }}
    </div>
    <div>
      Price: {{ product.price }}
    </div>
  </main>
  <nav class="actions">
    <button (clic)="buy.next()"
            style="background-color:coral; padding: 5px"><strong>Buy me!</strong>
</button>
```

```
  </nav>
</ab-products-product-template>
```

La idea es que bajo **una misma estructura puedas crear distintas implementaciones** y así mantener una coherencia visual o simplemente no repetir código.

2 Atributos custom con Directivas

```
As a: seller,
  I want: to see a green mark on products with stock
  so that: I know I don't do need to refill

As a: seller,
  I want: to see a red mark on products with out stock
  so that: I know I need to refill
```

2.1 Generación de directivas

Primero la definición. Si los componentes nos permite crear nuevas etiquetas en *html*, las directivas nos permiten crear atributos. Se usan para dar **funcionalidad extra a los elementos** estándar o no de las aplicaciones.

La forma de crearlas es mediante el CLI. En este caso con el comando `directive`.

```
ng g directive out-of-stock --project=products --export
```

EL resultado es una clase con su propio decorador `@Directive` en el que asignamos su selector. Es decir el nombre del atributo que podremos usar. El convenio es utilizar el prefijo y el propio nombre de directiva en *camel case*.

```
@Directive({
  selector: '[abProductsOutOfStock]'
})
export class OutOfStockDirective {
  private minimalStock = 10;

  @Input()
  set abProductsOutOfStock(stock: number) {
    const color = stock <= this.minimalStock ? 'MistyRose' : 'Aquamarine';
    this.el.nativeElement.style.backgroundColor = color;
  }

  constructor(private el: ElementRef) {}
}
```

En la implementación ya puedes hacer lo que te parezca. Es muy habitual que las directivas se usen para manipular el elemento nativo al que se aplican. Por eso casi siempre solicitarás la inyección de un puntero a dicho elemento: el de la dependencia de tipo `ElementRef` que viene en el *core* de Angular.

A partir de ahí ya puedes hacer lo necesario manipulando el *html* tanto en el constructor como en respuesta a cambios o durante la entrada de atributos.

2.2 Consumo de directivas

Usarlas es todavía más sencillo. Una vez exportadas e importadas como cualquier otro artificio Angular, ya puedes asignarlas como atributo a las etiquetas de tus plantillas.

`apps\shop\src\app\catalog\product\product.component.html`

```
  <div [abProductsOutOfStock]="product.stock">
    Stock: {{ product.stock }}
  </div>
```

Algunas, como este caso, reciben valores. Otras simplemente actúan una vez asignadas. De nuevo la idea es que **todas las manipulaciones de bajo nivel se *escondan* en las directivas** y que los controladores de nuestros componentes queden limpios y desacoplados de la capa de presentación.

3 Funciones de transformación con Pipes

```
As a: customer,
  I want: to see a product price also in dollars
  so that: I can compare prices
As a: customer,
  I want: to see a product price also in pounds
  so that: I can compare prices
```

En este caso lo que queremos es transformar un dato de nuestro modelo antes de presentarlo en pantalla. Para ello seguro que has usado algún `pipe` de Angular desde el primer día que empezaste a programar.

3.1 Generación de pipes

Pero ahora vamos a generar nuestros propios *pipes*. Una vez más usaremos el CLI con su comando específico para este caso:

```
ng g pipe exRate --project=products --export
```

Al igual que los componentes y que las directivas, los *pipes* tienen un nombre como clase y un nombre en su decorador para ser usados desde fuera.

```
@Pipe({
  name: 'exRate'
})
export class ExRatePipe implements PipeTransform {
  private readonly euroDollars = 1.13;
  private readonly ratesApi = 'https://api.exchangeratesapi.io/latest?symbols=';

  constructor(private httpClient: HttpClient) {}

  public transform(euros: number, symbol: string): number | Observable<number> {
    if (!symbol) {
      return euros * this.euroDollars;
    } else {
      return this.getOnlineRates$(symbol).pipe(map(rate => euros * rate));
    }
  }
}
```

La clase tienen que implementar la interfaz `PipeTransform` que obliga a tener un método público `transform(value: any):any`. En este método definimos nuestra función de transformación.

La única salvedad es incidir en que sea lo más ligera posible pues se ejecuta en cada uso y repintado. Por ejemplo yo aquí necesito un servicio asíncrono, y para no penalizar demasiado el rendimiento, le aplico el operador `shareReplay` para que actúe como una caché.

```
private getOnlineRates$(symbol: string) {
  const ratesUrl = this.ratesApi + symbol;
  return this.httpClient.get<any>(ratesUrl).pipe(
    shareReplay(1),
    refCount(),
    map(resp => resp.rates[symbol])
  );
}
```

3.2 Consumo de pipes

La parte del consumo es la más simple. Se trata de invocar a la transformación pasándole los atributos necesarios y volver a transformar o utilizar directamente el resultado.

`apps\shop\src\app\catalog\product\product.component.html`

```
  <div>
    ${{ product.price | exRate | number:'1.0-0'}}
    {{ product.price | exRate:'GBP' | async | number:'1.0-0'}} £
  </div>
```

Y con esto ya tenemos tres nuevas maneras de reutilizar código y dar nuevas capacidades a la parte de presentación. Pero en este tutorial de formación avanzada en Angular volvemos de nuevo a las profundidades del modelo y la gestión de datos empezando con el Flujo reactivo unidireccional con Angular y RxJs.

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