4 - Flow

> Flujo de datos entre componentes Angular

Los desarrollos profesionales son complicados pero **con Angular tenemos soluciones de comunicación simples para pantallas complejas**. Mediante el desarrollo de componentes atómicos y reutilizables Angular 7 favorece la implementación de buenas prácticas.

Crear y comunicar muchos componentes puede llevarnos a código difícil de seguir. La librería `@angular/forms` ofrece *tuberías de comunicación* para **mantener el flujo de datos bajo control**.

Partiendo de la aplicación tal cómo quedó en Formularios, tablas y modelos de datos en Angular. Al finalizar tendrás una aplicación que reparte la responsabilidad de recoger y presentar datos en componentes.

> Código asociado a este artículo en *GitHub*: AcademiaBinaria/angular-board/

1. Comunicación entre componentes

Las aplicaciones web en las que destaca Angular suelen ser complejas y con mucha variedad funcional en diversas páginas. A menudo esas páginas están repletas de formularios, informes y botones. La solución viene de mano del viejo principio *divide y vencerás*: **La componentización**

1.1. Necesidad de comunicación

El *framework* permite y recomienda repartir el trabajo en múltiples componentes de responsabilidad única. También es práctica común el crear páginas específicas para situaciones concretas aunque relacionadas. Por supuesto que estructuras como el menú de navegación o secciones de estado general necesitan conocer datos provenientes de las páginas. Nada está completamente asilado. Esto nos enfrenta la problema de **comunicar componentes**.

1.2. Escenarios

Las situaciones que te encontrarás caerán en alguna de estas tres categorías para las que hay soluciones específicas.

Comunicar componentes acoplados

Solemos empezar creando un componente por página. Pero es normal que esa página se complique y la solución a la complejidad es la **división en componentes y reparto de responsabilidades**. Dado que están en una misma página existe cierto acoplamiento entre ellos y eso nos facilitará la comunicación.

Comunicar componentes en páginas distintas

Cuando los componentes se carga en rutas distintas ya no hay forma de comunicarlos directamente. Pero lo resolveremos fácilmente **usando las capacidades del router.**

Comunicar componentes entre estructuras dinámicas

La situación más compleja se da cuando queremos comunicar componentes o servicios desacoplados pero sin cambio de página. En este caso hará falta un **mediador observable**.

2. El patrón Contendor / Presentadores

En **arquitectura de software** cuando encontramos una solución a un problema recurrente le ponemos un nombre y tratamos de utilizarlo siempre que podemos. Obviamente es una elección del programador y siempre tiene un coste que debe valorar. En este caso la ventaja es clara: **reparto de responsabilidades**.

2.1 El patrón

En este caso **el patrón contendor/presentadores** estipula que haya un único componente responsable de obtener, mutar y guardar el estado. Será el componente contenedor. Los presentadores serán responsables de.. ejem, presentar la información y los elementos de interacción con el usuario. Las ventajas derivadas son: mayor facilidad para el *testeo* y mayores posibilidades de reutilización de presentadores.

> A este patrón a veces se le conoce como parent/children por la jerarquía html que genera.

Veamos una implementación sencilla. Haremos una interfaz mínima para simular el manejo de un coche. Habrá pedales de aceleración y freno, y un cuadro dónde se refleje la velocidad. Para todo ello vamos a usa el *Angular CLI* y crear un módulo y sus componentes base.

```
ng g m 4-flow/car
ng g c 4-flow/car/car
ng g c 4-flow/car/car/display
ng g c 4-flow/car/car/pedals
```

Agregamos una ruta en el enrutador con su enlace en el menú.

```
{
  path: 'car',
  loadChildren: './car/car.module#CarModule'
}
```

```
<a routerLink="car" class="button">
  <span> 4 - Car</span>
</a>
```

2.2 El contendor

En el componente contenedor tendremos **una vista muy sencilla y un controlador más complejo**. La vista será la composición de los componentes presentadores, pero el controlador tendrá que obtener datos, aplicarles lógica de negocio y guardarlos cuando corresponda.

> No es habitual asignarle un sufijo al nombre del componente para indicar que es el contendor. Suele ser suficiente el verlo en la raíz de la jerarquía de carpetas.

```
<app-display [model]="car.name"
             [currentSpeed]="car.currentSpeed"
             [topSpeed]="car.maxSpeed"
             [units]="'Km/h'">
</app-display>
<app-pedals (brake)="onBrake($event)"
            [disableBrake]="disableBrake"
            (throttle)="onThrottle($event)"
            [disableThrottle]="disableThrottle">
</app-pedals>
```

Vemos que usa los componentes presentadores `Display` y `Pedals` enviándoles información y suscribiéndose a sus eventos. Concretaremos esta funcionalidad más adelante.

```
public car: CarModel;
public disableBrake: boolean;
public disableThrottle: boolean;

constructor() {}

public ngOnInit() {
  this.car = { name: 'Roadster', maxSpeed: 120, currentSpeed: 0 };
  this.checkLimits();
}
private checkLimits() {
  this.disableBrake = false;
  this.disableThrottle = false;
  if (this.car.currentSpeed <= 0) {
    this.car.currentSpeed = 0;
    this.disableBrake = true;
  } else if (this.car.currentSpeed >= this.car.maxSpeed) {
    this.car.currentSpeed = this.car.maxSpeed;
    this.disableThrottle = true;
  }
}
public onBrake(drive: number) {
  this.car.currentSpeed -= this.getDelta(drive);
  this.checkLimits();
}
public onThrottle(drive: number) {
  this.car.currentSpeed += this.getDelta(drive);
  this.checkLimits();
}
private getDelta = (drive: number) =>
  drive + (this.car.maxSpeed - this.car.currentSpeed) / 10;
```

Lo dicho, *la clase controladora del componente contendor retiene el grueso de la funcionalidad*. En este caso inicializar una instancia de un coche y mantener sus velocidad en los límites lógicos respondiendo a las acciones del usuario conductor.

2.3 Envío hacia el presentador con @Input()

Esta comunicación *hacia abajo* envía la información **desde el contenedor hacia el presentador**. Es similar a como una plantilla recibe la información desde el controlador.

@Input()

Para que una vista muestre datos tiene que usar directivas como `{{ model }}` asociada a una propiedad pública de la clase componente. Se supone que dicha clase es la responsable de su valor. Pero también puede **recibirlo desde el exterior**. La novedad es hacer que lo reciba vía *html*.

```
<h2> {{ model }} </h2>
<h3> Top speed: {{ topSpeed | number:'1.0-0' }}</h3>
<div class="card">
  <div class="section">
    {{ currentSpeed | number:'1.2-2' }} {{ units }}
  </div>
  <progress [value]="currentSpeed"
            [ngClass]="getSpeedClass()"
            [max]="topSpeed">
  </progress>
</div>
```

Empieza por decorar con `@Input()` la propiedad que quieres usar desde fuera. Por ejemplo un código como este del archivo `display.component.ts`.

```
export class DisplayComponent implements OnInit {
  @Input() public model: string;
  @Input() public currentSpeed: number;
  @Input() public topSpeed: number;
  @Input() public units: string;
  constructor() {}
  ngOnInit() {}
  public getSpeedClass = () =>
    this.currentSpeed < this.getThreshold() ? 'primary' : 'secondary';
  private getThreshold = () => this.topSpeed * 0.8;
}
```

Ahora puedes enviarle datos a este componente desde el *html* de su consumidor. Por ejemplo desde `car.component.html` le puedo enviar una variable o cualquier expresión evaluable. Recordemos como usa `[propiedad]="expresion"` en el elemento presentador.

```
<app-display [model]="car.name"
             [currentSpeed]="car.currentSpeed"
             [topSpeed]="car.maxSpeed"
             [units]="'Km/h'">
</app-display>
```

En la clase controladora del presentador quedan responsabilidades reducidas a temas específicos como determinar las clases *css* apropiadas o transformar los datos para su presentación.

Estoy usando al componente de nivel inferior como un presentador; mientras que el contenedor superior actúa como controlador. Este mismo patrón puede y debe repetirse hasta **descomponer las vistas en estructuras simples** que nos eviten repeticiones absurdas en código.

De esta forma es fácil crear componentes reutilizables; y queda muy limpio el **envío de datos hacia abajo**. Pero, ¿y hacia arriba?.

2.4. Respuesta del presentador con @Output()

Los componentes de nivel inferior no sólo se dedican a presentar datos, también presentan controles. Con ellos el usuario podrá crear, modificar o eliminar los datos que quiera. Aunque no directamente; para hacerlo **comunican el cambio requerido al contenedor de nivel superior**.

@Output()

Por ejemplo, el componente `PedalsComponent` permite acelerar y frenar. Bueno, realmente permite que el usuario diga que lo quiere hacer; los cambios se harán más arriba. Veamos lo básico del `pedals.component.html` antes de nada:

```
<h3> Pedals: </h3>
<form>
  <input value="brake"
    class="secondary"
    type="button"
    [disabled]="disableBrake"
    (click)="brake.emit(1)"/>
  <input value="throttle"
    class="tertiary"
    type="button"
    [disabled]="disableThrottle"
    (click)="throttle.emit(1)"/>
</form>
```

Claramente son un par de botones que con el evento `(click)` responden a acciones del usuario. En este caso se manifiesta una intención de acelerar o frenar el coche. Pero el método del controlador no actúa directamente sobre los datos.

> Si lo hiciera sería más difícil gestionar los cambios e imposibilitaría el uso de inmutables o técnicas más avanzadas de programación que se verán más adelante...

En su lugar, lo que hace es **emitir un evento** confiando que alguien lo reciba y actúe en consecuencia. Por ejemplo la emisión de la instrucción de frenado se realiza mediante la propiedad `brake` decorada con `@Output() public brake new EventEmitter<number>();`. Dicha propiedad será una instancia de un emisor de eventos que mediante el método `.next()` que emite la señal hacia arriba.

```
export class PedalsComponent implements OnInit {
  @Input() public disableBrake: boolean;
  @Input() public disableThrottle: boolean;
  @Output() public brake = new EventEmitter<number>();
  @Output() public throttle = new EventEmitter<number>();
  constructor() {}
  ngOnInit() {}
}
```

Mientras tanto, **en el contenedor la vista se subscribe al evento** `(brake)` como si este fuese un evento nativo y llama a los métodos que manipulan los datos de verdad.

```
<app-pedals (brake)="onBrake($event)"
            [disableBrake]="disableBrake"
            (throttle)="onThrottle($event)"
            [disableThrottle]="disableThrottle">
</app-pedals>
```

Las propiedades *output* también pueden enviar argumentos que serán recibidos mediante el identificador `$event` propio del framework. Se declaran especificando el tipo del argumento en el genérico del constructor de `EventEmitter<any>`.

En el controlador ya podemos operar con los datos. El método `onBrake(drive: number)` accede y modifica el valor de la velocidad y lo notifica automáticamente hacia abajo.

De esta manera se cierra el círculo. Los componentes de bajo nivel pueden **recibir datos para ser presentados o emitir eventos para modificarlos**. El componente de nivel superior es el **único responsable de obtener y actuar** sobre los datos.

3. Comunicaciones entre páginas o estructuras

3.1 Comunicación entre distintas páginas

En las aplicaciones hay **comunicaciones de estado más allá de la página actual**. La comunicación entre páginas es responsabilidad del `@angular/router`. Una vez activada una ruta, el sistema carga un componente en el `<router-outlet>` correspondiente. No hay forma de comunicarse hacia *(arriba) o [abajo]* con algo desconocido. De una página a otra tampoco es problema pues la comunicación va mediante los parámetros de la *url*.

Ya hemos usado esta comunicación anteriormente en el tema 2-spa el componente `AuthorComponent` es capaz de recibir por parámetros una identificación de un autor. Esa información es el resultado de una acción

del usuario en la pantalla `/about/authors` programada en el componente `AuthorsComponent`. Por tanto es una comunicación entre componentes, en la que ambos son *controladores hermanos*.

> Desde luego habrá que mejorar el acceso y control de los datos que por ahora es muy rudimentario. Lo haremos en próximos pasos. Primero mediante Servicios inyectables en Angular y después usando Comunicaciones HTTP en Angular

2.2 Comunicación entre estructuras desacopladas

Estando en la misma ruta, no siempre se podrán conocer los componentes, y por tanto no se podrán usar sus `[propiedades] y (eventos)`

2.2.1 El layout principal y los componentes por ruta páginas.

Una situación habitual es **comunicar la vista de negocio activa con elementos generales** de la página. Por ejemplo podrías querer mostrar la velocidad máxima alcanzada en la barra del menú o un un mensaje emergente cada vez que se alcance la velocidad límite. En este caso, el `<router-outlet>` es una barrera que impide usar el patrón contendor-presentador pues no se puede predecir el contenido dinámico que carga el `RouterOutlet`.

2.2.2 Múltiples niveles de presentadores.

Cuando las pantallas se hacen realmente complejas empiezan a surgir **árboles de componentes** de muchos niveles de profundidad. En estas situaciones mantener un único controlador a nivel raíz es poco práctico. Enviar hacia abajo las `[propiedades]` es tedioso, pero peor aún es hacer burbujear los `(eventos)` por varias capas de presentadores.

La solución en ambos casos pasa por permitir que *algunos componentes presentadores tengan su propio control de datos*. Este tipo de comunicaciones técnicamente se resuelve mediante *Observables* y merece un capítulo especial que se verá más adelante en esta serie. Incluso en situaciones complejas habrá que optar por patrones avanzados de gestión de estado como pueda ser *Redux*.