5 - Inject

> Servicios inyectables en Angular

La presentación, la lógica y el manejo de datos son tres capas de abstracción que usamos los programadores para mantener organizado nuestro código. En Angular 7, la presentación es cosa de los componentes. **La lógica y los datos tienen su lugar en servicios compartidos**.

Para que los componentes consuman los servicios de forma controlada tenemos proveedores *inyectables* en la librería `@angular/core` con los que realizar **la inyección de dependencias**.

Partiendo de la aplicación tal cómo quedó en Flujo de datos entre componentes Angular. Al finalizar tendrás una aplicación que comunica componentes entre páginas, reparte responsabilidades y gestiona claramente sus dependencias.

> Código asociado a este artículo en *GitHub*: AcademiaBinaria/angular-board/5-inject
>
> > Tienes una versión desplegada operativa para probar Angular Board

1. Inyección de dependencias

Como casi todo en Angular, **los servicios son clases TypeScript**. Su propósito es contener lógica de negocio, clases para acceso a datos o utilidades de infraestructura. Estas clases son perfectamente instanciables desde cualquier otro fichero que las importe. Pero Angular nos sugiere y facilita que usemos su sistema de inyección de dependencias.

Este sistema se basa en convenios y configuraciones que controlan la instancia concreta que será inyectada al objeto dependiente. Ahora verás cómo funciona la **Dependency Injection en Angular**.

Como demostración vamos a trabajar con un par de utilidades para conversión de unidades. Crearé un módulo y un componente en el que visualizarlo.

```
ng g m 5-inject/converter --routing true
ng g c 5-inject/converter/converter
```

1.1 Generación de servicios

La particularidad de las clases de servicios está en su decorador: `@Injectable()`. Esta función viene en el `@angular/core` e **indica que esta clase puede ser inyectada** dinámicamente a quien la demande. Aunque es muy sencillo crearlos a mano, el CLI nos ofrece su comando especializado para crear servicios. Estos son ejemplos de instrucciones para crear un *service*.

```
ng g s 5-inject/converter/converter
```

El resultado es el fichero `converter.service.ts` con su decorador que toma una *class* normal y produce algo *injectable*. Veamos una implementación mínima:

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class ConverterService {
  constructor() {}

  public fromKilometersToMiles = (kilometers) => kilometers * 0.62137;
}
```

Ahora tienes centralizado en este servicio la lógica de datos que tenemos hasta el momento. Los demás componentes la podrán utilizar.

1.2 Consumo de dependencias

Declarar y decorar la clase no es suficiente para poder reclamarla. Necesitas **registrarla como un proveedor en algún módulo**. Desde Angular 6 los servicios se auto-proveen en el módulo raíz mediante la configuración `providedIn: 'root'` de su decorador.

> Esto es útil y cómodo en una gran cantidad de casos. El módulo raíz es visible para toda la aplicación de forma que cualquier componente puede reclamar un servicio suyo sin problema. Excepto que el problema sea el tamaño. El módulo raíz se carga al arrancar y todas sus referencias van el *bundle* principal. Si queremos repartir el peso debemos llevar ciertos servicios al módulo funcional que los necesite.

Vamos a consumir este servicio en el `converter.component.ts`. Al consumo de los servicios inyectables se le conoce como *dependencia*. Cada componente o servicio puede **declarar en su constructor sus dependencias** hacia servicios inyectables. El convenio exige que se especifique el tipo esperado

```
export class ConverterComponent implements OnInit {
  public kilometers = 0;
  public miles: number;

  constructor(private converterService: ConverterService) {}

  public ngOnInit() { this.convert(); }

  public convert() {
    this.miles = this.converterService.fromKilometersToMiles(this.kilometers);
  }
}
```

> Agregar el modificador de alcance `private` o `public` en la declaración de argumentos hace que *TypeScript* genere una propiedad inicializada con el valor recibido. Es azúcar sintáctico para no tener

> que declarar la propiedad y asignarle el valor del argumento manualmente. En resumen, los constructores en TypeScrip admiten argumentos que transforman en propiedades. Mantenemos privado el `converterService` para evitar su uso desde la vista.

```
<h2> Distance Converter.</h2>
<h3> From Europe to USA </h3>
<form>
  <fieldset>
    <section>
      <label for="kilometers">Kilometers</label>
      <input name="kilometers"
             type="number"
             [(ngModel)]="kilometers"
             placeholder="0" />
    </section>
  </fieldset>
  <input value="Convert" type="button" (click)="convert()">
</form>
<section>
  <h4>{{ miles | number:'1.2-2' }} miles</h4>
</section>
```

2. Inversión del control

Un concepto íntimamente relacionado con la inyección de dependencias es el de **Inversion of Control**. El componente dependiente expresa sus necesidades, pero es el *framework* el que en última instancia decide lo que recibirá. Vemos entonces que **el invocado cede el control al invocador**.

Cuando proveemos un servicio en Angular, el comportamiento por defecto es el de proveer un *singleton* pero hay más opciones. Si se usa el objeto `provider` con `useClass` , `useValue` y `useFactory` podemos controlar el proceso de inyección.

Se crea un *singleton* por cada módulo en el que se provea un servicio. Normalmente si el servicio es para un sólo módulo funcional se provee en este y nada más. Si va a ser compartido gana la opción de auto proveerlo en el raíz, garantizando así su disponibilidad en cualquier otro módulo de la aplicación.

En un módulo cualquiera, siempre podríamos agregar un servicio a su array de *providers*.

```
@NgModule({
  declarations: [...],
  imports: [...],
  providers: [ ConverterService ]
})
```

Pero siempre será **una instancia única por módulo**. Si un *singleton* no es lo adecuado, entonces puedes proveer el mismo servicio en distintos módulos. De esa forma se creará una instancia distinta para cada uno.

Si se provee la misma clase en dos o más módulos se genera una instancia en cada uno de ellos. Los componentes recibirán la instancia del módulo jerárquicamente más cercano.

> Incluso es posible usar el array `providers:[]` en la decoración de un componente o de otro servicio. Haciendo así aún más granular la elección de instancia.

Veamos un ejemplo extendiendo el problema del conversor de unidades de forma que se pueda escoger una **estrategia** de conversión en base a una cultura concreta. Para empezar necesitamos una interfaz, un servicio base que la implemente y un componente que lo consuma.

```
ng g interface 5-inject/converter/culture-converter
ng g service 5-inject/converter/culture-converter
ng g component 5-inject/converter/culture-converter
```

```
export interface CultureConverter implements CultureConverter {
  sourceCulture: string;
  targetCulture: string;
  convertDistance: (source: number) => number;
  convertTemperature: (source: number) => number;
}
```

```
export class CultureConverterService implements CultureConverter {
  sourceCulture: string;
  targetCulture: string;
  convertDistance: (source: number) => number;
  convertTemperature: (source: number) => number;

  constructor() {}
}
```

```
export class CultureConverterComponent implements OnInit {
  public source: string;
  public target: string;
  public sourceUnits = 0;
  public targetUnits: number;

  constructor(private cultureConverterService:CultureConverterService){ }

  public ngOnInit() {
    this.source = this.cultureConverterService.sourceCulture;
    this.target = this.cultureConverterService.targetCulture;
    this.convert();
  }
  public convert() {
    this.targetUnits =
```

```
this.cultureConverterService.convertDistance(this.sourceUnits);
  }
}
```

```
<h2> Culture Converter.</h2>
<h3> From {{ source }} to {{ target }} </h3>
<form>
  <fieldset>
    <section>
      <label for="sourceUnits">Distance</label>
      <input name="sourceUnits"
             type="number"
             [(ngModel)]="sourceUnits"
             placeholder="0" />
    </section>
  </fieldset>
  <input value="Convert" type="button" (click)="convert()">
</form>
<section>
  <h4>Distance {{ targetUnits | number:'1.2-2' }} </h4>
</section>
```

2.2 Implementaciones

El `CultureConverterComponent` depende de `CultureConverterService` el cual implementa de forma abstracta la interfaz `CultureConverter`. Pero eso no es para nada funcional. Vamos a crear dos implementaciones específicas para Europa y USA. Estas clases concretas se apoyarán en el anteriormente creado `ConverterService` que necesita algo más de código.

```
export class ConverterService {
  constructor() {}

  public fromKilometersToMiles = (kilometers) =>kilometers * 0.62137;

  public fromMilesToKilometers = (miles) => miles * 1.609;

  public fromCelsiusToFarenheit = (celsius) => celsius * (9 / 5) + 32;

  public fromFarenheitToCelsius = (farenheit) => (farenheit - 32) * (5 / 9);
}
```

Y aquí están las dos servicios concretos.

```
@Injectable()
export class EuropeConverterService {
  sourceCulture = 'USA';
```

```
  targetCulture = 'Europe';
  constructor(private converterService: ConverterService) {}
  convertDistance = this.converterService.fromMilesToKilometers;
  convertTemperature = this.converterService.fromFarenheitToCelsius;
}
```

```
@Injectable()
export class UsaConverterService implements CultureConverter {
  sourceCulture = 'Europe';
  targetCulture = 'USA';
  constructor(private converterService: ConverterService) {}
  convertDistance = this.converterService.fromKilometersToMiles;
  convertTemperature = this.converterService.fromCelsiusToFarenheit;
}
```

2.3 Provisión manual

Por ejemplo si queremos utilizar la implementación concreta de USA lo indicamos en el módulo que lo consuma.

```
{
  providers: [
    {
      provide: CultureConverterService,
      useClass: UsaConverterService
    }
  ]
}
```

El componente reclama una instancia de `CultureConverterService` y le damos otra con la misma interfaz. De esta forma podríamos tener módulos distintos, cada uno con su propia estrategia de conversión.

2.4 Factoría

Una situación muy común es poder elegir dinámicamente la implementación concreta. Para ello necesitamos una función factoría que con alguna lógica escoja la estrategia concreta.

```
const cultureFactory = (converterService: ConverterService) => {
  if (environment.unitsCulture === 'metric') {
    return new EuropeConverterService(converterService);
  } else {
    return new UsaConverterService(converterService);
  }
};
{
  providers: [
```

```
    {
      provide: CultureConverterService,
      useFactory: cultureFactory,
      deps: [ConverterService]
    }
  ]
}
```

De esta forma la aplicación se comportará distinto en función de una variable de entorno.

Ya tenemos la aplicación mucho mejor estructurada, pero el almacén de datos es mejorable. Se mantienen los datos *hard-coded*, muy incómodo para actualizar; y en memoria, poco fiable y volátil. Lo más habitual es guardar y recuperar la información en un servidor *http*. Eso lo veremos en el próximo tema.