UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

ESTUDIO SOBRE ANGULAR 2 Y SUPERIOR

Piero Rospigliosi Beltrán

2020

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN

INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

"ESTUDIO SOBRE ANGULAR 2 Y SUPERIOR"

Autor: **Piero Rospigliosi Beltrán**

Director: **Carlos Delgado Hita**

Tribunal:

Presidente: .....

Vocal 1º: ..... Vocal 2º: .....

Calificación: .....

Fecha: ..... de ..... de .....

En agradecimiento
a todos mis
compañeros y
profesores
quienes siempre
estuvieron
dispuestos a
prestar ayuda.

ÍNDICE RESUMIDO

ÍNDICE DETALLADO

ÍNDICE DE FIGURAS

1.INTRODUCCIÓN

El desarrollo software avanza hacia formas que permiten programar a más alto nivel, tener mayor capacidad de abstracción y mayor capacidad para centrarse en la lógica de negocio de cada aplicación a desarrollar.

Esto se debe al aumento creciente de la necesidad de crear aplicaciones cada vez más robustas y minimizar los recursos humanos necesarios el desarrollo, la implementación, la operación, el mantenimiento y reutilización del código.

Esta evolución y aumento de necesidades no es ajeno al desarrollo web en la parte del cliente, comúnmente denominado *front-end*. Una aplicación web está basada en la arquitectura cliente/servidor. En la cual un cliente, a través de una interfaz realiza peticiones al servidor. El servidor, al recibir la petición, se encarga de procesar la información y devolver el recurso solicitado al cliente, quien se encargará de presentar la información obtenida en el formato adecuado.

Debido a que el número de servidores es limitado y el número de clientes es creciente, interesa trasladar cada vez más responsabilidades a la capa del cliente, ya que estos serán procesados en el ordenador del cliente aligerando la carga de trabajo del servidor.

El tipo de responsabilidades que se han ido trasladando al cliente tienen que ver desde, el renderizado de la información, validaciones en formularios, seguridad, entre otros.

En este sentido Angular y TypeScript (lenguaje de programación utilizado por Angular) ofrecen una serie de ventajas que serán explicadas en detalle en los siguientes capítulos de este trabajo.

TypeScript que es una extensión de JavaScript, tiene como particularidad el tipado, la orientación a objetos y su alineamiento con estándares nuevos como ECMAScript 6 (ES6).

Angular nos ofrece características tan importantes como los controladores, servicios, directivas para organizar el proyecto o el Data Binding que nos permite reflejar automáticamente cambios en nuestro modelo en el DOM de nuestra aplicación, tarea que tradicionalmente se hacía de forma manual.

2. OBJETIVOS DEL PROYECTO

El objetivo del documento es repasar las características principales de Angular, unos de los frameworks front-end más utilizados por los programadores en el mundo, y evidenciar como estas características son el resultado natural de la evolución del desarrollo web a lo largo de la historia.

Mediante el repaso de las tecnologías previas al framework y el repaso de sus propias características se pretende dar a conocer como Angular respalda el ciclo de vida de software, brindando herramientas para que el código de programación a desarrollar sea más fácil de comprender, desarrollar, mantener e implementar. Siendo la meta última minimizar el coste de vida de la aplicación y maximizar la productividad del programador.

Para ello, se llevará a cabo el desarrollo de una aplicación muy básica, la cual nos servirá para tomar ejemplos y referencias de los elementos que se explican en el documento en los diferentes apartados.

3. CONCEPTOS PREVIOS

3.1 JavaScript

Es un lenguaje de programación que permite crear acciones en las páginas web. Es un lenguaje de programación nativo de los navegadores quienes son los encargados de interpretar estos códigos. Lo cual quiere decir que cada navegador tiene su propio intérprete del código JavaScript y se vuelve necesario una normalización mediante un estándar.

Es un lenguaje de programación de scripts (secuencias de comandos) y se define como orientado a objetos, basado en prototipos, débilmente tipado, dinámico, imperativo e interpretado.

```
<script type="text/javascript">
  document.write("Este es un ejemplo de una sentencia JavasScript")
</script>
```

Figura 1. Ejemplo básico JavaScript

3.2 ECMAScript

Es el estándar que define como debe ser el lenguaje JavaScript publicado por ECMA International. Dicho de otra manera, es la especificación que deben seguir los creadores de software para crear intérpretes para JavaScript.

Javascript es un lenguaje interpretado lo cual significa que ejecuta las instrucciones directamente (sin una compilación previa) a través de un intérprete. Existen múltiples intérpretes los cuales suelen utilizarse en los diferentes navegadores web o en servidores web como Node.js.

Por lo que, ECMAScript surge para establecer las reglas de juego que deben cumplir los diferentes intérpretes a la hora de establecer como debe funcionar y debe ser interpretado el lenguaje JavaScript.

Este estándar ha ido evolucionado con el tiempo y añadiendo nuevas novedades pero es el año en el año 2015 donde se lanza ECMAScript 6, el cual introduce novedades importantes que conducen al JavaScript más moderno de hoy en día.

ECMAScript 6 introduce cambios importantes como:

- El uso de clases, las cuales permiten utilizar estructuras predefinidas como molde para crear objetos. Las clases son un conjunto de atributos y métodos, que en JavaScript clásico equivaldrían a variables y funciones.

  La aparición de las clases nos brinda un valor importantísimo porque nos permite reutilizar con mayor facilidad nuestras estructuras de datos y por consiguiente un código más limpio.

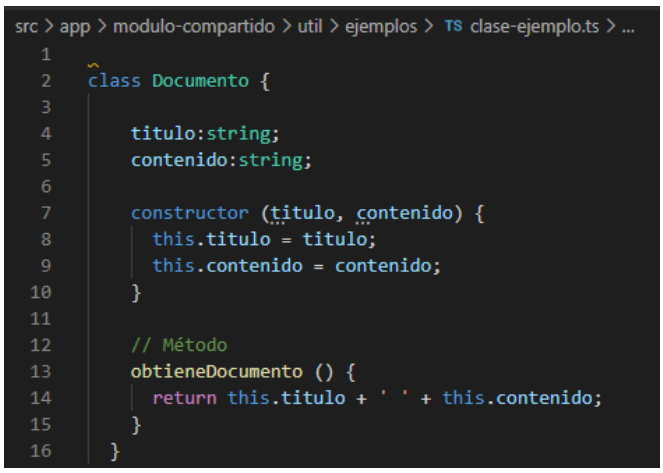


Figura 2. Ejemplo clase ECMAScript

- El concepto de módulos, los cuales permiten definir archivos que contienen directivas de importación y exportación a nivel superior.

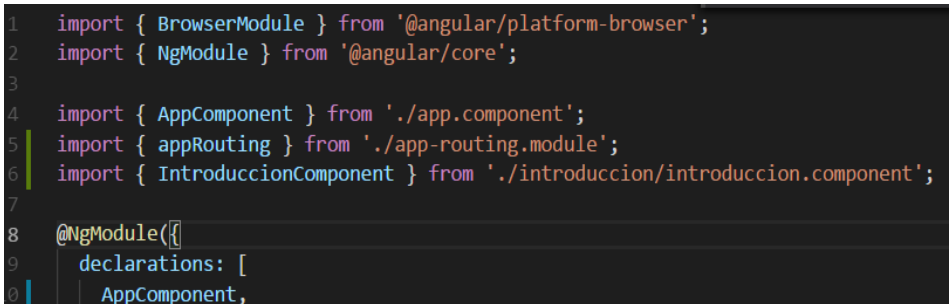


Figura 3. Declaración de un módulo

- Distintos tipos de variables con el uso de las declaraciones const y let. La primera nos permite crear constantes, y la segunda variables locales en segmentos de código. Esto también facilita crear aplicaciones más robustas, ya que podemos usar estas declaraciones para asegurar el alcance de estas y evitar errores humanos controlando su posible o no modificación.

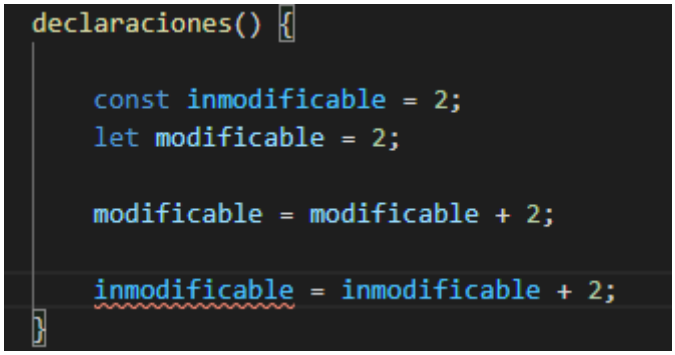


Figura 4. Ejemplo declaraciones const y let

- Otras características, que nos permiten diseñar mejor nuestra lógica de negocio como el poder tener valores por defecto en nuestras funciones, el uso de lambdas o poder utilizar el paradigma de la programación funcional. Un ejemplo de estos son las funciones de orden superior, map, filter y reduce.

  - map: El resultado nos devuelve una colección de elementos que son el resultado de la expresión aplicada al elemento.

    Por ejemplo:

    ```
    [1,2,3].map(n => n+1).
    ```

    En este ejemplo el array se iteraría, siendo n cada uno de los valores, y se aplicaría la expresión que en este caso es sumar 1, de esta forma el resultado esperado sería [2,3,4]

- filter: El resultado tendrá la misma forma que el argumento de entrada, sin embargo solo se quedará con los elementos que cumplan la condición dada.

  Por ejemplo:

```
[1,2,3,4].filter(n=> n < 3).
```

  Este caso le estamos diciendo que itere todos los valores y se quede solo con aquellos menores que 3, por lo que el resultado esperado sería [1,2]

- reduce: Nos permite aplicar una función contra un acumulador y los elementos de un array (de izquierda a derecha) para reducirlo a un solo valor.

  Por ejemplo:

```
 [1,2,3,4].reduce((acumulador, valorActual) => acumulador + valorActual).
```

  En este caso cada valor se sumará al actual valor acumulado hasta dicho momento. Actuando como una sumatoria donde el resultado sería 10.

  Siendo los pasos:

```
0+1 => 1
1+2 => 3
3+3=> 6
6+4=> 10
```

  Como vemos cada nuevo valor se suma al anterior valor acumulado en dicho momento.

Este tipo de operadores nos permite abstraer de cómo se hace algo y crear código más claro que permita y centrarse en la lógica de negocio o en la funcionalidad a desarrollar.

3.3 TypeScript

TypeScript es un superconjunto de JavaScript, concretamente del estándar ECMAScript 6. Esto quiere decir que incluye todas las funcionalidades de ECMAScript 6 pero añade funcionalidades extra propias.

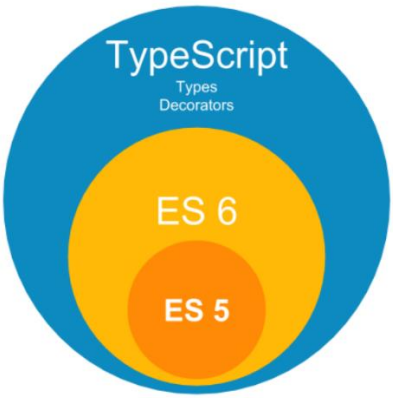


Figura 5. Composición de TypeScript

Existen una serie de características extras importantes que TypeScript ofrece con respecto al estándar ECMAScript 6 y que hacen los desarrolladores de Angular prefieran utilizando en lugar de limitarse al estándar.

Entre las funcionalidades extras más importantes encontramos:

- Es un lenguaje compilado: Lo cual es una ventaja porque permite encontrar errores en tiempo de compilación antes de abordar la ejecución. El compilador de Typescript

transforma el código a código Javascript tradicional el cual puede ser interpretado por todos los navegadores.

- Variables tipadas: Permite a los programadores utilizar herramientas y prácticas de desarrollo altamente productivas: verificación estática, refactorización de código, finalización de declaraciones y navegación basada en símbolos.

Por ejemplo:

Se define un array de tipo number

```
numeros:Array<number>
```

Luego se añade dos elementos:

```
numeros.push(1);
numeros.push(2);
```

Hasta el momento se tiene al array [1,2]. Ahora se intenta añadir un número, pero en forma de string.

```
numeros.push("3");
```

Esto en JavaScript clásico, como por ejemplo con el estándar ECMAScript 5, no daría ningún problema. Sin embargo, en Typescript veríamos un error en tiempo de compilación.

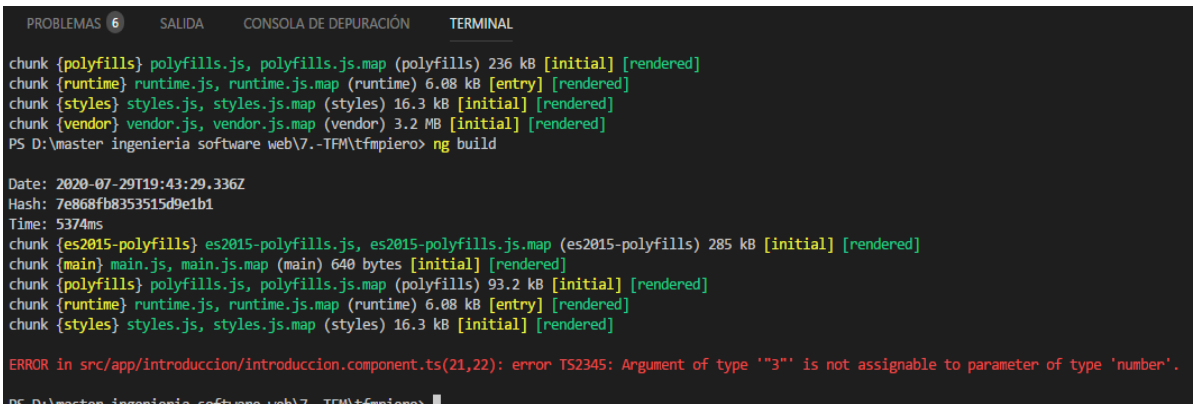


Figura 6. Error de compilación con TypeScript

Esto ofrece una gran ventaja, porque evita encontrar el error en tiempo de ejecución. Incluso si se utiliza un editor adecuado, se podrá observar el error al instante, sin necesidad de compilar el código.

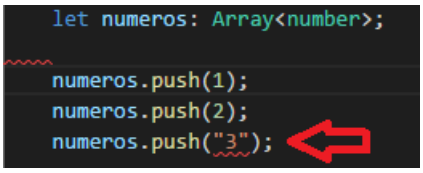


Figura 7. Aviso de error en Visual Studio Code

- Modificadores de acceso *public, private y protected*: Los cuales permiten definir el alcance de un miembro de una clase. Nuevamente, esto permite detectar errores en tiempos de compilación minimizando el error humano.

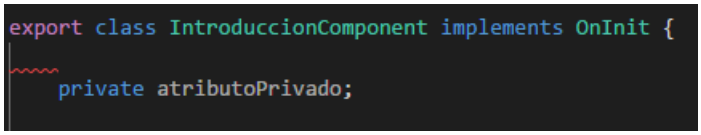


Figura 8. Ejemplo atributo con declaración de acceso

- Decoradores: Permite implementar el patrón decorador, el cual sirve para añadir funcionalidad extra de forma dinámica, evitando crear nuevas clases que heredan de nuestra clase a expandir.

  Por ejemplo:

  Dado la clase llamada ECMAScript la cual tiene una función que imprime el argumento pasado por parámetro.

```
imprimeArgumento(parametro: string) {
   console.log(`Parametro recibido es ${parametro}`);
}
```

Figura 9. Función imprimeArgumento()

Se llama a la función:

```
ngOnInit() {
    this.imprimeArgumento('Soy un parametro muy útil');
}
```

Figura 10. Invocación de la función imprimeArgumento()

El resultado de esta función tal como se espera es la impresión en consola del parámetro recibido.

Este es el component Ecmascript inicial

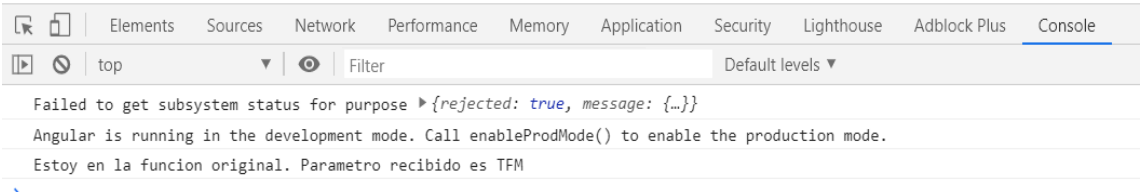


Figura 11. Resultado de la función en la consola del navegador Google Chrome

Sin embargo, puede requerirse que cuando se llame a ciertas funciones que consideramos importantes, se imprima información relativa a su ejecución, como la clase de donde se invoca y los argumentos recibidos.

Para resolver este nuevo problema una opción sería modificar las funciones deseadas y añadir la funcionalidad. Otra opción sería recurrir a la herencia y sobrescribir el método imprimeArgumento() mediante una nueva clase.

Las dos opciones anteriores tienen el mismo objetivo: agregar una nueva funcionalidad a una función ya existente. Sin embargo, si se en cuenta los principios de SOLID (Acrónimo mnemónico introducido por Robert C. Martin a comienzos de la década del 2000 que representa cinco principios básicos de la programación orientada a objetos y el diseño.) y particularmente, el principio de responsabilidad única (Single Responsability Principle, SRP), que nos indica que una clase o módulo debe tener una única responsabilidad u objetivo, estas opciones quedan

descartadas. Porque si la función tiene un objetivo, que es imprimir argumentos, cualquier otra acción secundaria rompería el principio mencionado. Si se opta por no cumplir el principio mencionado se obtendría un fragmento de código y por tanto una aplicación con peor legibilidad, mantenimiento y escalabilidad.

Es aquí donde se encuentra una ventaja al uso de decoradores ya que permite esconder todo el código secundario nuevo y evitar añadirlo a la función. De esta forma, el siguiente desarrollador a cambiar la funcionalidad principal, solo tendrá que preocuparse de eso, olvidándose de las acciones secundarias.

La manera en que funciona el decorador es que primero se entrará por la función del decorador y podemos realizar acciones con los metadatos recibidos de la función decorada.
Se crea del decorador el cual como podemos observar tiene una forma predefinida en los argumentos que nos facilita TypeScript.

```
export  function imprimeInformacionFuncion(target: Object, propertyKey: string, descriptor: any) {

    const metodoOriginal = descriptor.value;
    console.log('Clase: ', target.constructor.prototype);
    console.log('Método: ', propertyKey);
    console.log('Property Descriptor: ', descriptor);

    descriptor.value = function (...args: any[]) {
        console.log('Argumentos de la funcion', args);
        console.log('Se va invocar la función ' + propertyKey);
        metodoOriginal.apply(this, args);
    }


    return descriptor;
}
```

Figura 12. Función asociada al decorador

Se ve que se imprime en consola la información de los metadatos de la función. Y luego invocamos a la función original para que se continúe haciendo la funcionalidad principal.
Ahora se puede decorar el método con el nuevo decorador.

```
ngOnInit() {
    this.imprimeArgumento('TFM');
}

@imprimeInformacionFuncion
imprimeArgumento(parametro: string) {
    console.log(`Estoy en la funcion original. Parametro recibido es ${parametro}`);
}
```

Figura 13.Ejemplo función decorada

En el resultado en la consola ha impreso los metadatos para luego invocar la función original.

```
Angular is running in the development mode. Call enableProdMode() to enable the production mode.                core.js:16829
Clase:  ▸ {ngOnInit: ƒ, imprimeArgumento: ƒ, constructor: ƒ}                                                    ecmascript.component.ts:27
Método:  imprimeArgumento                                                                                       ecmascript.component.ts:28
Property Descriptor:  ▸ {writable: true, enumerable: true, configurable: true, value: ƒ}                       ecmascript.component.ts:29
Argumentos de la funcion ▸ ["TFM"]                                                                              ecmascript.component.ts:32
Se va invocar la función imprimeArgumento                                                                       ecmascript.component.ts:33
Estoy en la funcion original. Parametro recibido es TFM                                                         ecmascript.component.ts:19
```

Figura 14. Resultado final decorador

De esta forma se ha añadido código nuevo y secundario de forma elegante y sin ensuciar la funcionalidad principal.

- Editores para trabajar con TypeScript: Es una experiencia más agradable y eficiente trabajar con un editor como Visual Studio Code, que permite tener autocompletado o detección errores al momento de programar.

  Por ejemplo, aquí se ve como nos indica que no existe la variable hola.

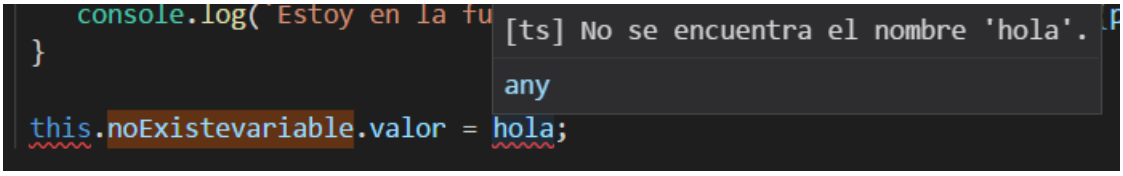


Figura 15. Ejemplo en Visual Studio Code

Y aquí sugiere mediante un autocompletar las posibles funciones a utilizar para la variable string parámetro

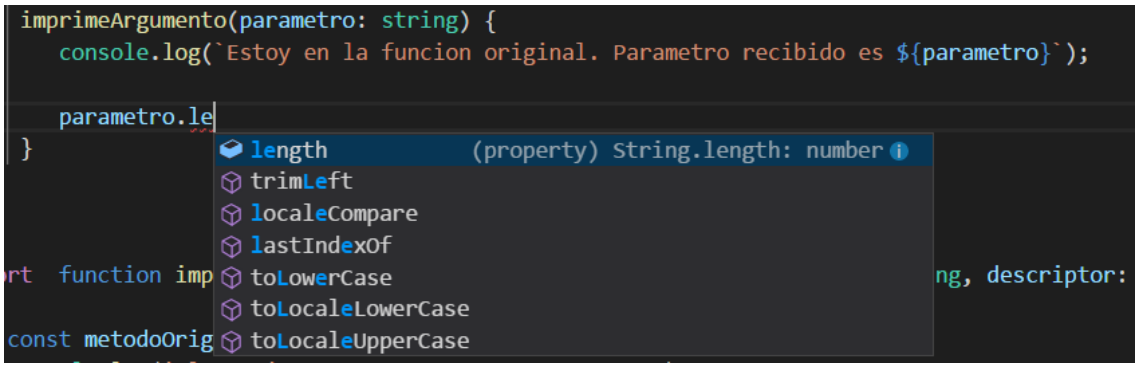


Figura 16. Sugerencia en Visual Studio Code

- El apoyo de grandes empresas y proyectos. Empresas como Microsoft, el creador de TypeScript, o Google, quien mantiene el framework Angular, el cual es objeto de estudio en este trabajo.

4. ANGULAR

Es un framework para crear aplicaciones cliente una sola página (SPA) eficientes y sofisticadas utilizando HTML y Typescript.

Desarrollado en TypSscript, por Google, nos permite disponer de herramientas para que el desarrollo y las pruebas sean más fáciles.

Angular es la evolución de AngularJS, aunque incompatible con su predecesor. Esto se debe a que es un framework totalmente nuevo que surge como solución a las carencias que mostraba AngularJS con respecto a problemas de rendimiento, escalabilidad, modularidad y también el soporte en móviles.

4.1 Arquitectura

La arquitectura de una aplicación Angular se basa en dos componentes fundamentales los cuales son los módulos y los componentes. Los bloques de construcción básicos son los módulos o también llamados NgModules, los cuales contienen a los componentes, quienes junto a sus plantillas definen una vista.

Los módulos proporcionan un contexto de compilación para los componentes y permiten descomponer nuestras funcionalidades en bloques individuales que exponen interfaces de comunicación bien definidas. Una aplicación Angular está definida por un conjunto de módulos y siempre tiene al menos un módulo raíz que permite el arranque y carga todos los demás módulos.

En ingeniería del software este tipo de arquitectura se denomina arquitectura basada en componentes, el cual es un enfoque que se centra en la descomposición del diseño en componentes funcionales o lógicos.

Una aplicación Angular se basa en la unión de distintas piezas de código los cuales son diseñados para cumplir con un propósito particular y que agrupan todo lo relacionado con alguna funcionalidad concreta de forma aislada haciendo explícito una interfaz para ofrecer sus servicios. Y dado que los módulos son quienes representan estas piezas de código, éstos son los que vendrían a representar el papel de componentes desde el punto de vista de una arquitectura basada en componentes.

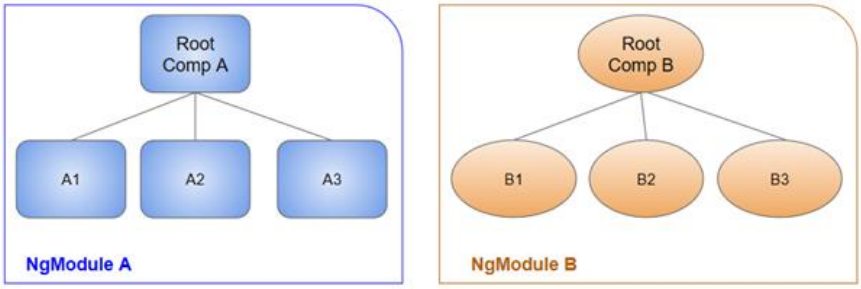


Figura 17. Ejemplos de composición de módulos

Los principios fundamentales en los que se basa la arquitectura basada en componentes son:

- Reusabilidad: Los componentes son diseñados para poder ser utilizados en distintos escenarios, ya que agrupan todo lo necesario para poder desempeñar su tarea específica.
- Libre de contexto: Debe poder operar en diferentes ambientes y contextos por lo que información específica como el estado de los datos deben ser pasados al componente y no estar incluidos en ellos.
- Extensibles: Puede expandir su funcionalidad o crear una nueva.
- Encapsulado: Exponen su funcionalidad mediante interfaces, evitando revelar detalles internos del componente.
- Independientes: Los componentes deben ser diseñados para no tener dependencia de otros componentes. Por lo que deben poder ser instalados sin afectar a otros componentes.

De esto se desprende que Angular, mediante el uso de esta arquitectura, intenta hacer suyo los principios mencionados permitiéndonos la creación de código más limpio, robusto y mantenible.

La documentación oficial de Angular nos incluye una gráfica donde se nos explica la relación de los elementos que componen su arquitectura y que serán explicados con más detalles en los siguientes apartados.

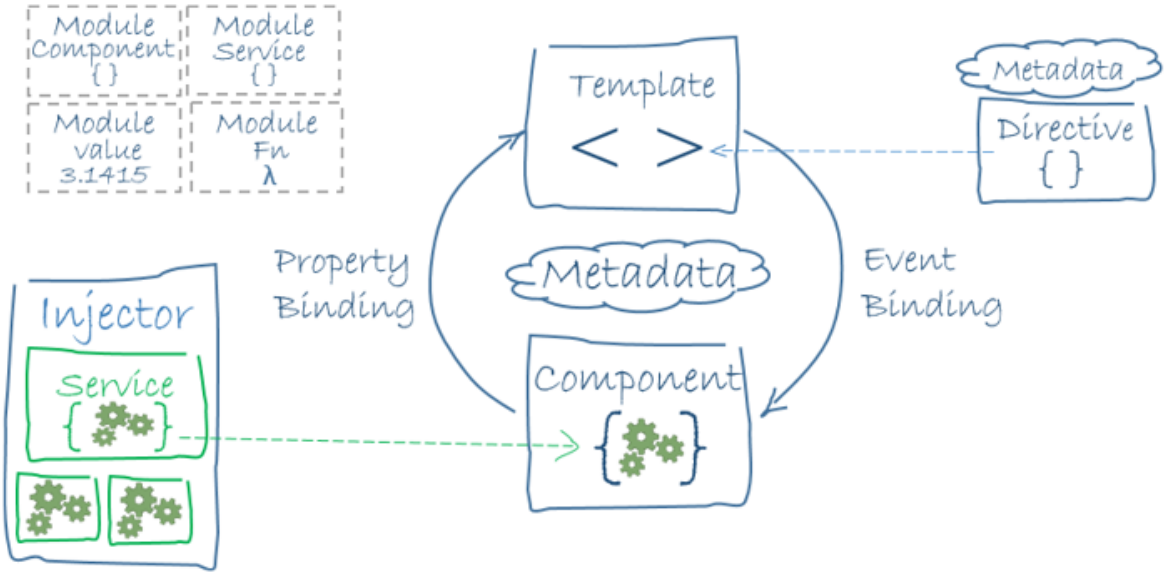


Figura 18. Arquitectura de Angular

En Angular los módulos, componentes y servicios son clases que usan decoradores. Estos decoradores marcan su tipo y proporcionan metadatos que le dicen a Angular cómo usarlos.

4.2 Módulos

Como se mencionó anteriormente los módulos o NgModules vendrían a representar esas piezas reutilizables que agrupan a modo de contenedor un bloque cohesivo de código dedicado a un dominio de aplicación, un flujo de trabajo o un conjunto de capacidades estrechamente relacionadas que suelen estar estrechamente ligadas al desarrollo de una funcionalidad.

Pueden contener componentes, proveedores de servicios y otros archivos de código cuyo alcance está definido por el módulo que los contiene. Un módulo puede importar a otro módulo y con ello importar su funcionalidad. Un módulo a su vez, puede exportar su funcionalidad para que la utilicen otros módulos.

Cada aplicación Angular tiene al menos una clase módulo, el módulo raíz, el cual convencionalmente se denomina AppModule y reside en el archivo app.module.ts.

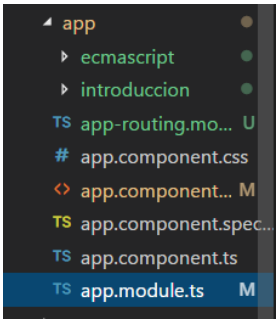


Figura 19. Módulo raiz

Este módulo será cargado desde un archivo principal, normalmente main.ts, el cual será el punto de entrada a la aplicación.

Figura 20. Fichero principal

El módulo raíz puede contener todos los módulos secundarios que necesite, los cuales a su vez pueden contener otros módulos representando una jerarquía de cualquier profundidad.

4.2.1 Composición de un Modulo

4.2.1.1 El decorador @NgModule

Un módulo se define mediante una clase decorada con @NgModule. Este decorador es una función que indica al framework que la clase decorada se trata de un módulo e informa de las propiedades que lo describen.

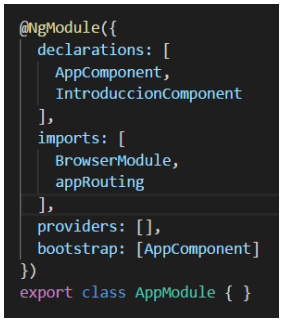


Figura 21. Composición del módulo raíz

Las propiedades que componen un módulo son:

4.2.1.2 El array declarations

Se trata de los componentes, directivas, y pipes que pertenecen a este módulo.

4.2.1.3 El array exports

Es el subconjunto del array declarations que deseamos que sean visibles y utilizables por otros módulos que importen este módulo.

4.2.1.4 El array imports

En este apartado otros elementos o módulos van a ser necesarios para que el módulo actual funcione correctamente.

4.2.1.5 El array providers

Aquí declaramos todos aquellos servicios que pasan a ser parte de la colección global de servicios. Es decir, se vuelven accesibles desde cualquier parte de la aplicación.

4.2.1.6 El array Bootstrap

Esta propiedad solo debe ser llamada por el módulo raíz. Aquí se llama al componente principal el cual aloja a todas las demás vistas de la aplicación.

4.3 Componentes

Un componente controla una zona de espacio de la pantalla a través de las vistas. Un componente es una clase estándar decorada con @Component. Esta clase define la lógica de una vista y tiene asociado una plantilla que contiene el código HTML junto a directivas de Angular que afectan su comportamiento.

Figura 22. Decorador Component

4.3.1 Componentes y Data Binding

Los componentes son dinámicos. Esto quiere que Angular los va creando, actualizando o destruyendo conforme el usuario se mueve por la aplicación. Al encargarse Angular de la actualización del DOM quita esta responsabilidad a los desarrolladores y permite crear un código más limpio al enfocarse más en la lógica de negocio.

Sin Angular y sin el Data Binding, los programadores serían los responsables de introducir nuevos valores de datos en el código HTML y de convertir las respuestas de los usuarios en acciones y actualizaciones de valores. Dejar esto en mano de los programadores puede resultar tedioso, propenso a errores, y más difícil de leer y por tanto de mantener. Data Binding nos permite delegar estas responsabilidades al framework.

Data Binding nos permite mostrar contenido dinámico en lugar de estático. Es la comunicación entre nuestro código HTML y nuestra lógica de programación ubicado en nuestro archivo .ts.

Es un enlace de datos bidireccional, un mecanismo para coordinar las partes de una plantilla con las partes de un componente. Lo cual quiere decir que el DOM puede modificar la clase y la clase puede modificar al DOM logrando que cualquier cambio en el modelo de datos de nuestra clase se refleje inmediatamente en nuestra plantilla HTML, y viceversa.

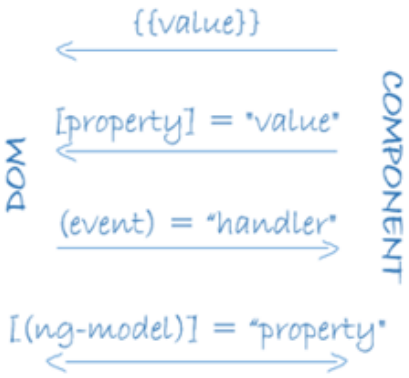


Figura 23. Funcionamiento de Data Binding

Por ejemplo:

En Angular, para construir los elementos de una tabla nos basta con recorrer los elementos de un array.

```
<div class="table-responsive">
    <table class="table">
        <thead>
            <tr>
                <th scope="col">Valor</th>
                <th scope="col">Label</th>
            </tr>
        </thead>
        <tbody>
            <tr *ngFor="let objeto of listadoObjetos">
                <td>{{objeto.value}}</td>
                <td>{{objeto.label}}</td>
            </tr>
        </tbody>
```

Figura 24. Ejemplo construcción de una tabla con Angular

Siendo la variable “listadoObjetos” un array de objetos con 5 valores que se pueden ver a continuación

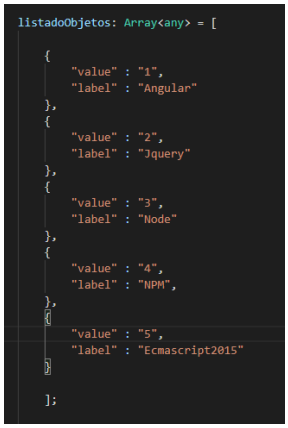


Figura 25. Ejemplo variable "listadoObjetos"

Angular nos construye los elemento del DOM con la etiqueta <tr> de manera dinámica gracias a la directiva *ngFor:

Figura 26. Vista de tabla HTML creada

Si quisiéramos actualizar los valores de dicha tabla, bastaría con actualizar la variable “listadoObjetos”. Lo cual nos ofrece ventajas comparativas con respecto a tecnologías JavaScript anteriores como JQuery dónde se tendría que eliminar , actualizar o agregar un <tr> de forma manual. Esto permite que los programadores se centren en la funcionalidad principal, que en este caso consiste en actualizar la variable que representa a los datos.

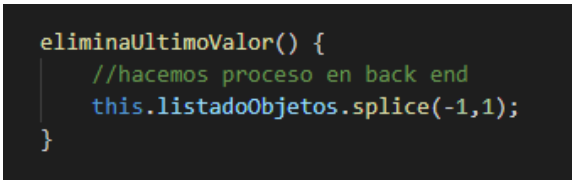


Figura 27. Función que actualiza atributo

4.4 Servicios

Los servicios son las clases que se utilizan para manejar los datos. Abarcan cualquier valor, función o característica que necesita una aplicación. Tiene típicamente un propósito limitado y bien definido y aportan a los desarrolladores la capacidad de reutilización del código y mantener los componentes lo más desacoplados posible.

Esto gracias a que los servicios en Angular trabajan mediante el patrón de diseño inyección de dependencias. El funcionamiento consiste en que Angular instanciará nuestros servicios y los dejará disponibles para que un módulo o componente los inyecte y estén disponibles para todo el módulo o componente.

Es decir, un mismo servicio puede ser inyectado en un módulo y estar disponible en todo el módulo, sin necesidad de importarlo en todas las clases que se vaya a utilizar y sin necesidad de instanciar dicha clase pues Angular lo habrá hecho por nosotros.

La función principal de un servicio en Angular es interactuar con la parte del servidor, el backend, de la aplicación en busca de los datos. Un componente interactúa con el backend a través de un servicio. Este servicio además puede ser reutilizado en todos los sitios donde se inyecte.

Un servicio es una clase normal el cual lleva el decorador @Injectable, el cual proporciona los metadatos necesarios para que Angular pueda inyectar dicha clase.

```
@Injectable()
export class ComponenteService {
```

Figura 28. Decorador @Injectable

4.4.1 Servicios en módulos

Como se mencionó un servicio puede estar inyectado en un módulo. De manera que todos los elementos del módulo tengan acceso al servicio. De esta forma todos los componentes de un módulo concreto pueden acceder al servicio que se ha registrado en el array providers correspondiente.

```
@NgModule({
    imports: [componenteRouting, ModuloCompartidoModule],
    exports: [],
    declarations: [ComponenteComponent, ListadoComparacionComponent],
    providers: [ComponenteService],
})
export class ComponenteModule { }
```

Figura 29. Ejemplo servicios en módulo

4.4.2 Servicios en componentes

Un servicio puede ser declarado directamente en un componente concreto. Lo cual significará que dicho servicio solo estará disponible para ese componente en concreto, o por sus componentes hijos.

Cuando se inyecta un servicio directamente en un componente, este servicio se creará solo en el momento que se cree el componente y su ciclo de vida dependerá del ciclo de vida del componente.

```
@Component({
    selector: 'app-componente',
    templateUrl: 'componente.component.html',
    providers: [ ComponenteService ]
})

export class ComponenteComponent implements OnInit {
```

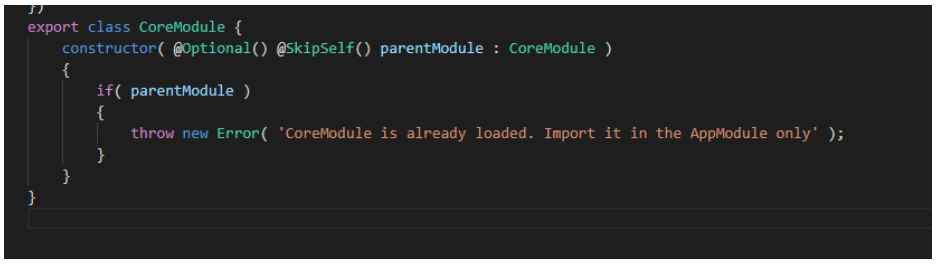
Figura 30. Ejemplo servicio importado directamente en componente

4.4.3 Servicios y el Patrón de diseño Singleton

El patrón de diseño Singleton consiste en restringir la creación de un objeto a una única instancia. Angular nos permite aprovechar la ventaja de este enfoque y utilizarlo para nuestros servicios. Es decir, podemos hacer que un servicio determinado sea un Singleton.

La ventaja de que un servicio sea un Singleton viene del hecho que podemos asegurar que los datos se manipulen en él no estarán sufriendo cambios no deseados desde algún otro componente. Por ejemplo, si se está transfiriendo datos desde un componente hacia otro componente a través de un servicio, es de vital importancia que ninguno de los dos componentes implicados pueda instanciar nuevamente al servicio que sirve como conductor.

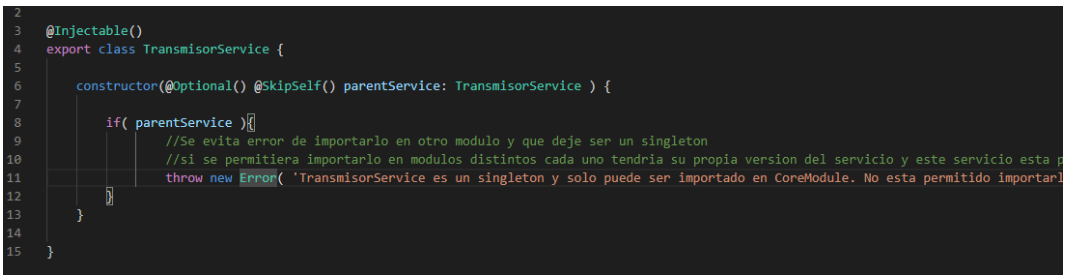
La forma más común de hacer esto es crear un módulo llamado CoreModule el cual se inyectará en el módulo raíz y quién contendrá nuestros servicios singleton. Además, CoreModule utilizará las anotaciones @Optional y @SkipSelf para comprobar que no se instancie en ningún otro lado.



```
export class CoreModule {
    constructor( @Optional() @SkipSelf() parentModule : CoreModule )
    {
        if( parentModule )
        {
            throw new Error( 'CoreModule is already loaded. Import it in the AppModule only' );
        }
    }
}
```

Figura 31. Módulo CoreModule

Una forma más concreta de hacerlo es agregar directamente en el servicio las anotaciones que aseguran que no se instancie más de una vez una clase.



```
@Injectable()
export class TransmisorService {

    constructor(@Optional() @SkipSelf() parentService: TransmisorService ) {

        if( parentService ){
            //Se evita error de importarlo en otro modulo y que deje ser un singleton
            //si se permitiera importarlo en modulos distintos cada uno tendria su propia version del servicio y este servicio esta p
            throw new Error( 'TransmisorService es un singleton y solo puede ser importado en CoreModule. No esta permitido importarl
        }
    }

}
```

Figura 32. Asegurar Singleton directamente en servicio

4.5 Directivas

Las directivas son una de las piezas clave que nos brinda Angular para manipular el DOM de manera más sencilla y por tanto tener código más mantenible.

Se puede decir que las directivas son pequeñas instrucciones de código que se aplican sobre el DOM. En cierto sentido, los componentes son un tipo de directiva, ya que cuando creamos un componente lo que estamos diciendo es que se construya el HTML a partir de nuestra clase .ts.

4.5.1 Directivas Estructurales

Las directivas estructurales son aquellas que se cambian el diseño del DOM, permitiendo añadir o eliminar sus elementos. Estas directivas son fácilmente reconocibles debido que están antecedidas por un asterisco (*) seguido del nombre de la directiva.

En Angular tenemos directivas estructurales predefinidas y directivas estructurales personalizadas.

4.5.1.1 Directivas Estructurales Predefinidas

Aquellas que nos proporciona Angular. Para poder usarlas hace falta importar el módulo CommonModule de @angular/core en el módulo deseado. Una buena práctica es importarlo dentro de un módulo compartido el cual contendrá este tipo de módulos predefinidos.

```
import { CommonModule, DatePipe }        from '@angular/common';
```

Figura 33. Importación módulo directivas

```
@NgModule({
    imports:      [
        CommonModule,
        RouterModule,
    ],
    declarations: [
        LocalDatePipe,
        LocalTimePipe,
        BooleanPipe,
        ObjectToArrayPipe,
    ],
    exports:      [
        CommonModule,
        ObjectToArrayPipe,
    ],
    providers:    [ DatePipe, LocalTimePipe],
    entryComponents : [  ]
})
export class ModuloCompartidoModule {}
```

Figura 34. Creación módulo compartido

4.5.1.1.1 Directiva Estructural *ngFor

Una directiva predefinida es la directiva *ngFor el cual nos permite crear elementos del DOM en bucle recorriendo un array.

Su sintaxis es *ngFor="let v of valores", donde valores es el array a recorrer y v es la variable local donde se almacena el valor iterado,

```
<tbody>
    <tr *ngFor="let objeto of listadoObjetos">
        <td>{{objeto.value}}</td>
        <td>{{objeto.label}}</td>
    </tr>
```

Figura 35. Ejemplo directiva *ngFor

En el ejemplo, se entiende que se crearán tantos <tr> como valores tenga el array "listadoObjeto"

```
<tr *ngFor="let value of values">
    <td>{{value}}</td>
</tr>
```

Figura 36. Sintaxis directiva *ngFor

La importancia de esta directiva reside en que, si queremos modificar los elementos del DOM nuevamente, basta con cambiar la variable asociada. Angular se encargará del resto.

4.5.1.1.2 Directiva Estructural *ngIf

La directiva *ngIf nos permite crear o eliminar un elemento del DOM según el valor de la variable comprobada. Esta directiva no solo esconde o muestra el elemento, sino que lo construye o destruye lo que nos brinda un mayor rendimiento en el cliente.

Su sintaxis es *ngIf="condicion" donde "condicion" es el la expresión booleana a comprobar. Esta directiva se aplica a todo el elemento y a sus hijos.

```
<button *ngIf="listadoObjetos.length > 0"
```

Figura 37. Ejemplo directiva *ngIf

De la imagen anterior se puede deduce que el botón solo se mostrará si la expresión "listadoObjectos.length > 0" resulta true.

4.5.1.2 Directivas Estructurales Personalizadas

Angular nos da la posibilidad de crear nuestras propias directivas estructurales de manipulación del DOM. De forma que se pueda crear un comportamiento personalizado al elemento del DOM al que se le aplique.

Para ello debemos crear una clase decorada con @Directive donde el parámetro selector será el nombre que se utilizará para identificar a la directiva cuando se aplique en las plantillas deseadas.

```
@Directive({
    selector: '[appVolver]'
})

export class VolverDirective implements OnInit {
```

Figura 38. Ejemplo directiva estructural personalizada

También es obligatorio importar las clases Input, TempleteRef y ViewContainer de @angular/core

```
import { Directive, ElementRef, HostListener, Input, OnInit, OnChanges, SimpleChanges } from '@angular/core';
```

Figura 39. Importación de clases obligatorias para crear directivas estructurales

Además, para poder manipular desde la clase el elemento DOM desde la clase hace falta declarar en el constructor a ElementRef.

```
constructor(private el: ElementRef, private router: Router) {}
```

Figura 40. Constructor en directiva estructural

Para asociar eventos a una función, la cual realizará las acciones deseadas, se utiliza el decorador @HostListener, al cual se le pasa por parámetro el evento asociado

```
@HostListener('click') onMouseEnter() {

    if(this.urlAVolver) {
        this.router.navigate( [ this.urlAVolver ] );
    } else {
        if(history) { // nunca deberia ser indefinido
            history.back();
        }
    }
}
```

Figura 41. Función asociada a evento en directiva estructural

Para este ejemplo se puede observar que los elementos que tengan asociado la directiva appVolver, realizarán las acciones de la función onMouseEnter() para el evento click.

```
<nav class="botonera-nav">
    <button  type="button" class="btn btn-warning" appVolver="/">
</nav>
```

Figura 42. Ejemplo de uso de directiva estructural personalizada

4.5.2 Directivas de Atributos

Son aquellas directivas que permiten cambiar la apariencia o el comportamiento de un elemento del DOM a diferencia de las directivas estructurales que permiten agregar o eliminar elementos del DOM.

Las directivas de atributo se utilizan como atributos de elementos.

Un ejemplo es la directiva NgStyle, la cual es una directiva de atributo predefinida que nos permite cambiar varios estilos del elemento al mismo tiempo.

4.6 Angular y la Programación Reactiva

La programación reactiva es un paradigma de programación asíncrona que se ocupa de los flujos de datos (streams) y la propagación del cambio. Una manera sencilla de explicar este concepto sería comparándolo con la programación tradicional secuencial en JavaScript.

Cuando se programa de forma tradicional las instrucciones se ejecutan una detrás de otra y el resultado no se altera si cambian las variables involucradas de forma posterior.

En el siguiente ejemplo, aunque se cambie el valor de la variable “a” de forma posterior, no se afectará a la variable suma, pues se ha cambiado después de que se ha realizado el cálculo.

```
let a = 1;
let b = 3;
let suma = a + b; // resultado seria 4
a = 7; // Asignamos otro valor a la variable a
```

Figura 43. Ejemplo programación tradicional

Sin embargo, mediante la programación reactiva la variable suma estaría constantemente atento a los valores involucrados en la operación, en este caso los valores a y b, y si se cambian también cambiará su resultado.

En la programación reactiva se pueden crear flujos de datos (streams) asíncronos de cualquier cosa, como por ejemplo los valores que toma una variable a lo largo del tiempo o los clics sobre un botón.

La ventaja de esto es que nos ofrece sistemas que son capaces de consumirlos de distintos modos, aunque centrándose principalmente en los siguientes tipo de eventos:

- La respuesta o aparición de un evento dentro del stream.
- La aparición de un error en el stream.
- La finalización del stream.

Angular nos ofrece diversas implementaciones para aprovechar este paradigma de programación.

4.7 Observables

Los observables son una implementación de la programación reactiva. Permiten producir eventos y consumirlos de diversos modos.

Se basa en el patrón de diseño software “Observer” donde un sujeto mantiene una lista de dependientes, llamados observadores, y les notifica automáticamente los cambios de estado.

Los observables son declarativos, porque no se ejecutan hasta que un consumidor se suscribe. A su vez, el consumidor recibe notificaciones hasta que se completa la función o hasta que se da de baja. Los observables brindan soporte para pasar mensajes entre las partes de la aplicación y permiten el manejo de eventos, programación asíncrona y manejo de valores múltiples.

Un observable puede entregar múltiples valores de cualquier tipo: literales, mensajes o eventos, según el contexto. Con los observables el código desarrollado solo necesita preocuparse de suscribirse para consumir valores y cuando termine, si fuese el caso, cancelar la suscripción.

4.7.1 RxJS

Es una librería Javascript que contiene la implementación de los observables y que utiliza Angular. Esta librería será necesaria hasta que sea parte el estándar ECMAScript y los navegadores lo admitan de forma nativa.

4.7.2 Observables en Angular con RxJS

Como se mencionó anteriormente Angular hace uso de los observables de la librería RxJS.

Las operaciones asíncronas más comunes en las que usan los observables son:

- Datos de salida de un componente secundario a un componente principal.
- En el módulo HTTP para manejar solicitudes y respuestas AJAX.
- En los módulos Router y Forms para escuchar y responder a los eventos de entrada del usuario.

4.7.2.1 Transmitiendo datos entre componentes

A través de la clase EventEmitter que se usa al publicar valores de un componente a través del decorador @Output. EventEmitter extiende a RxJS Subject y proporciona un método emit() para que puedan enviar valores mediante el metodo next() a cualquier observador suscrito.

4.7.2.2 Observables en el módulo HTTP.

En Angular, HttpClient devuelve observables de llamadas al método HTTP. Esto ofrece varias ventajas con respecto a peticiones Http basadas en promesas, entre las cuales encontramos:

- Los observables, no cambian la respuesta del servidor. En su lugar, puede utilizar una serie de operadores para transformar valores según sea necesario.
- Las solicitudes HTTP se pueden cancelar mediante el método unsubscribe()
- Las solicitudes fallidas se puede reintentar fácilmente.

4.7.2.3 AsyncPipe

Una tubería asíncrona o AsyncPipe, se suscribe a un observable o promesa y devuelve el último valor que ha emitido. Cuando se emite un nuevo valor, la tubería marca el componente a verificar para ver si hay cambios. Cuando el componente se destruye, la canalización asíncrona cancela la suscripción automáticamente para evitar pérdidas posibles de memoria.

4.7.2.4 Router

Router.events proporciona eventos como observables. Puede usar el operador filter() de RxJS para buscar eventos de interés y suscribirse a ellos para tomar decisiones basadas en la secuencia de eventos en el proceso de navegación.

ActivedRoute es un servicio de enrutador inyectado que utiliza observables para obtener información sobre una ruta y parámetros.

4.7.2.5 Reactive Forms

Tienen propiedades que usan observables para monitorear los valores de control de los formularios. Las propiedades FormControl, valuesChanges y statusChanges contienen observables que generan eventos de cambio. La suscripción a una propiedad de control de formularios es una forma de activar la lógica de la aplicación dentro de la clase de componente.

4.8 Angular Router

Con Angular construimos aplicaciones de una sóla página (SPA, Single Page Application). Esto consiste en mostrar todas las vistas o pantallas en una única página, sin realizar sucesivas cargas de otras direcciones provenientes del servidor.

Es decir, el usuario puede cambiar lo que ve, mostrando u ocultando partes de la pantalla que corresponden a componentes particulares, en lugar de solicitarlos al servidor.

Para poder manejar la navegación de una vista a otra se utiliza a Angular Router, un enrutador del lado cliente. El cual permite la navegación interpretando la URL del navegador como una instrucción para cambiar la vista.

4.8.1 Configurar Angular Router

Para crear el enrutamiento en Angular hace falta crear un archivo convencionalmente llamado app-routing.module.ts, el cual contendrá las rutas deseadas en forma de array.

```
const routes: Routes = [
    { path: 'introduccion', component: IntroduccionComponent },
    { path : 'ecmascript', loadChildren: './ecmascript/ecmascript.module#EcmascriptModule' },
    { path : 'componente', loadChildren: './componente/componente.module#ComponenteModule' },
    { path : 'directiva', loadChildren: './directiva/directiva.module#DirectivaModule' },
    { path: '**', pathMatch:'full', redirectTo: '/' }
];

export const appRouting = RouterModule.forRoot(routes);
```

Figura 44. Ejemplo configuración Angular Router

En el ejemplo anterior se observan dos formas de definir las rutas. Definiendo directamente una ruta a un componente concreto y definiendo una ruta a un módulo en concreto que tendrá su propia configuración de enrutamiento.
Definir una ruta a un módulo concreto trae la ventaja adicional de que dicho módulo solo será cargado si se solicita. Este mecanismo se denomina carga perezosa o Lazy Loading.

El siguiente paso es importar la constante creada en nuestro módulo raíz.

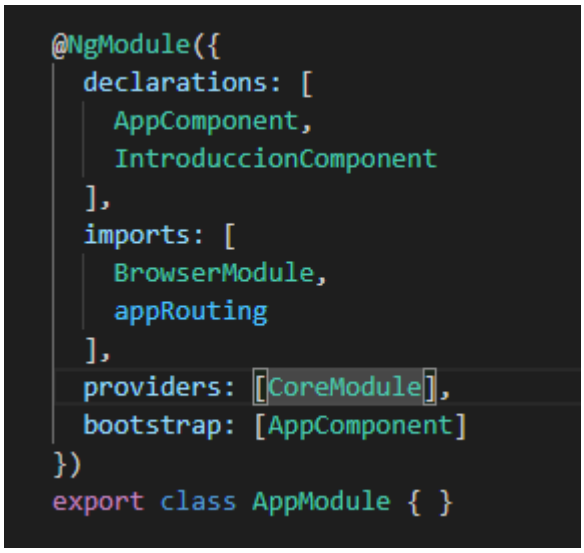


Figura 45. Declaración de Angular Router en el módulo raíz

Además debemos añadir el elemento <base href=”/> a la página principal de la aplicación, nuestro fichero index.html, para indicar al enrutador cómo componer las URLs de navegación.

Por último, para finalizar un enrutamiento básico debemos agregar la directiva routerOutlet a nuestro componente principal, pues así se le dice a Angular que todos los componentes se mostrarán donde se ha colocado la directiva mencionada.

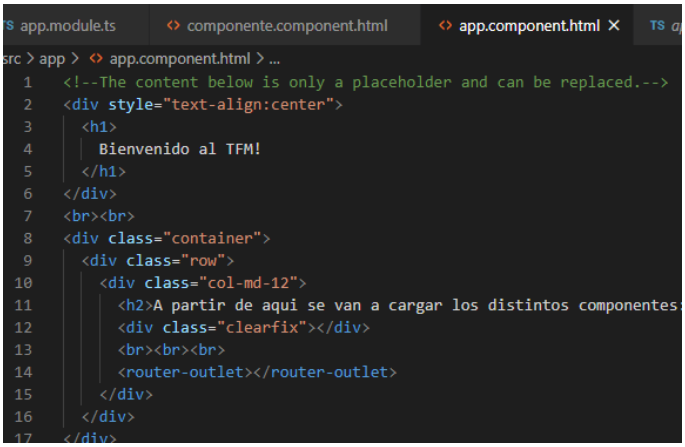


Figura 46. Declaración de directiva routerOutlet

4.9 Angular y Principios de diseño

Como se ha visto a lo largo del documento Angular es un framework que nos facilita la aplicación de distintos principios de diseño. El objetivo último de aplicar estos principios es crear aplicaciones que:

-Toleren el cambio

-Sean fáciles de comprender

-Sean escalables

A continuación, se explicará cómo Angular por su propia estructura, facilita la aplicación de principios de diseño importantes.

4.9.1 Angular y el Principio de responsabilidad única

El principio de responsabilidad única de SOLID, suele entenderse como que una función debe tener una, y solo una, razón para cambiar. Este principio es usado por los desarrolladores para refactorizar grandes funciones en funciones más pequeñas que tengan una única responsabilidad.

Sin embargo, el ámbito de este principio abarca va más allá que una función. En realidad, se refiere a que un módulo debería tener una, y sola una, razón para cambiar.

Como se vio en apartados anteriores, los módulos son uno de los componentes fundamentales de la arquitectura de Angular. Se puede decir que una aplicación Angular, es un conjunto de módulos.

Que Angular tenga esta estructura facilita el cumplimiento de este principio, ya que cada módulo puede representar un flujo de trabajo determinado ligado al desarrollo de una funcionalidad concreta. En Angular podemos crear cuantos módulos se crean convenientes y el desarrollo de cada módulo puede ser asignado a un grupo independiente de desarrolladores.

Esto nos aporta una serie de ventajas como evitar conflictos en el equipo de desarrollo como pueden ser las fusiones del sistema de control de versiones, duplicaciones accidentales de código o que alguna modificación en un módulo afecte a una funcionalidad distinta a la que se deseaba cambiar.

4.9.2 Angular y el Principio de abierto-cerrado

El principio de abierto-cerrado es otro de los principios de SOLID. Establece que, para que las aplicaciones sean fáciles de cambiar, deben diseñarse de forma que se permita que el comportamiento de dichos sistemas pueda cambiarse añadiendo código, en lugar de cambiar el código existente.

Que Angular este formado por módulos y estos a su vez formados de componentes permite descomponer hasta el detalle deseado una aplicación. Al ser una aplicación compuesta por pequeños elementos, es más sencillo lograr que los cambios de funcionalidad se traduzcan en elementos nuevos.

4.9.3 Angular y el Principio de sustitución de Liskov

El principio de sustitución de Liskov, es otro de los principios SOLID. Este principio afirma que, para crear aplicaciones a partir de partes intercambiables estas partes deben adherirse a un contrato que les permitan ser sustituidas por otras.

Aunque conceptualmente este principio trata sobre la herencia y la capacidad de sustitución de sus clases hijas. Se puede relacionar de forma filosófica, con la manera en que Angular está diseñado para permitir la fácil sustitución de módulos enteros con Angular Router o dentro de un módulo en concreto mediante sus mecanismos de exportación e importación de todos módulos.

5. RESUMEN Y CONCLUSIÓN

5.1 Resumen

Este documento se centra en el estudio teórico del framework Angular. Para ello se hace un repaso de la evolución de las distintas tecnologías predecesoras hasta llegar al propio framework y ofrecer una visión detallada de sus características.

Mientras estas se detallan, se evidencia como éstas permiten a los desarrolladores minimizar los recursos humanos y crear código de programación más fácil de comprender, desarrollar, mantener e implementar.

5.2 Conclusiones

El framework Angular es el resultado de la evolución natural del desarrollo de software web hacia formas que maximizan la productividad de los programadores.

Angular nos permite crear una aplicación con una mejor arquitectura y con ello minimizar los recursos humanos necesarios y crear código de programación más fácil de comprender, desarrollar, mantener e implementar.

Esto se debe a una serie de características clave que se detallan a continuación.

Arquitectura basada en componentes

La arquitectura de Angular es un enfoque que se centra en la descomposición del diseño en componentes funcionales o lógicos. Dichos componentes funcionales son nuestros módulos, los cuales son grupos cohesionados de código que agrupan una lógica concreta.

El objetivo de tener una arquitectura basada en componentes es conseguir módulos reusables, libres de contexto, extensibles, encapsulado e independientes.

Data Binding en Componentes

Los componentes son los elementos que definen un espacio de la pantalla. Tradicionalmente cualquier modificación a los elementos del DOM podía hacerse de formas. La primera es cargando toda la página nuevamente del servidor. La segunda es manipulando los elementos manualmente con tecnologías como jQuery, comúnmente dentro un callback() tras una respuesta a una petición Ajax.

Con el Data Binding de Angular, se puede delegar la responsabilidad de actualizar los elementos del DOM al framework, pues dichos elementos se construyen a partir de atributos de la clase del componente y permanecen unidos en un enlace bidireccional que permite sincronizar el DOM con la clase asociada.

Programación Reactiva

Angular posibilita el uso de librerías JavaScript, que permiten utilizar el paradigma de la programación reactiva. El cual es un paradigma de programación asíncrona que se ocupa de los flujos de datos y la propagación del cambio.

Angular nos ofrece mecanismos para consumir los flujos de datos de distintos modos, centrándose en los eventos para cambios, errores y finalización de los flujos de datos.

Siendo los observables de Angular, una implementación de la programación reactiva basado en el patrón de diseño “Observer” y uno de los mecanismos más importantes del framework para el manejo de eventos.

Inyección de dependencias

Angular hace uso de este patrón de diseño, mediante el cual los servicios están instanciados y disponibles para los módulos y componentes que lo necesiten sin necesidad de que sean estos los que instancien dichos servicios.

Además, en Angular existe un árbol de inyección de dependencias jerárquico, lo cual quiere decir que, si un módulo o componente no tiene una dependencia definida, se buscará en los nodos superiores del árbol. Lo cual facilita crear dependencias que estén

disponibles para toda la aplicación. Por ejemplo, si se quiere que un servicio este disponible para todos los módulos, bastaría con importarlo en el módulo raíz.

Uso de TypeScript frente a JavaScript

Angular aprovecha todas las ventajas que ofrece TypeScript frente a JavaScript, explicadas en apartados anteriores, como el uso de variables tipadas, compilación, modificadores de acceso, decoradores entre otros.

6.ANEXO

A continuación, se muestran algunas capturas de la aplicación creada mientras se realizaba este documento.

En la siguiente imagen se puede apreciar la pantalla inicial de la aplicación. Se ha señalado la separación entre el módulo raíz y los demás módulos que se cargan justo debajo para mayor claridad. Haciendo clic en los distintos apartados se cargarán los módulos.

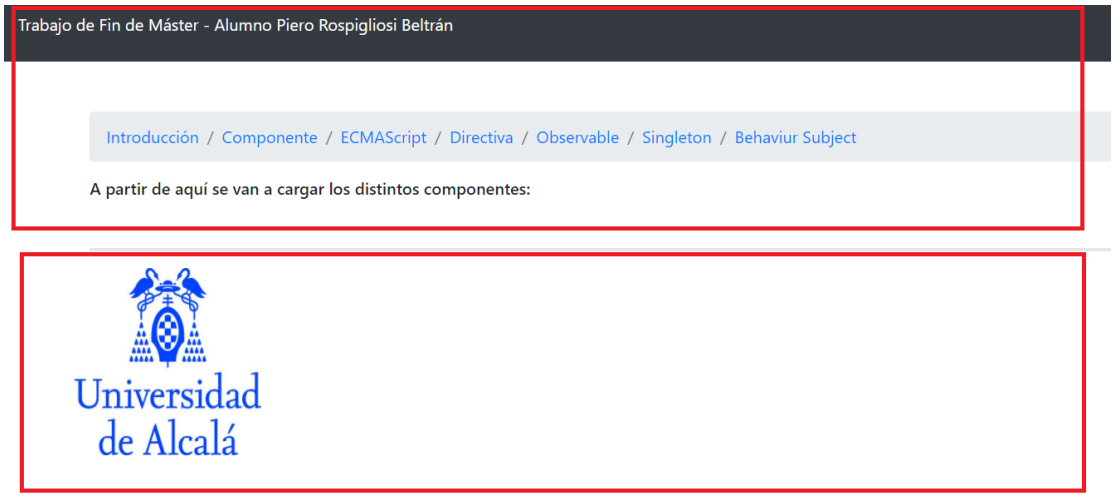


Figura 47. Pantalla introducción de la aplicación

En la siguiente imagen se puede observar la navegación hacia un ejemplo de cargar información mediante un observable

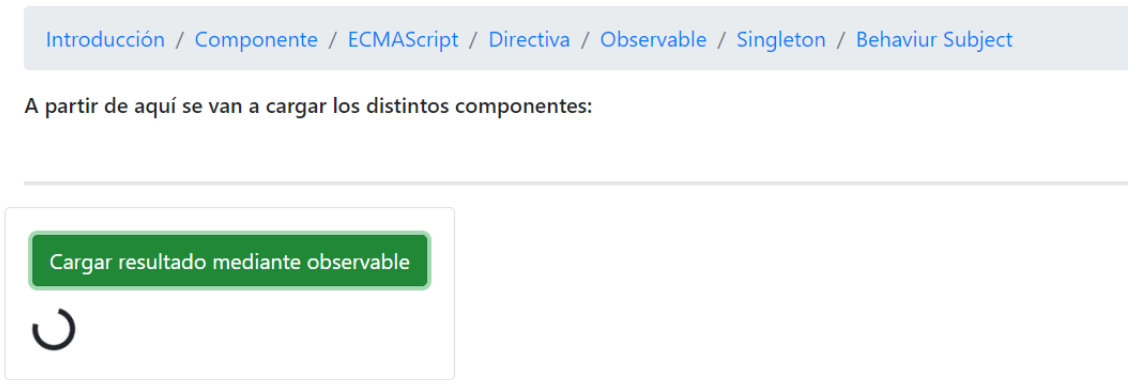


Figura 48. Módulo observable en la aplicación

El módulo “BehaviorSubjectModule” nos muestra como un valor emitido puede ser transmitido a todos los suscriptores y recordar el último valor.

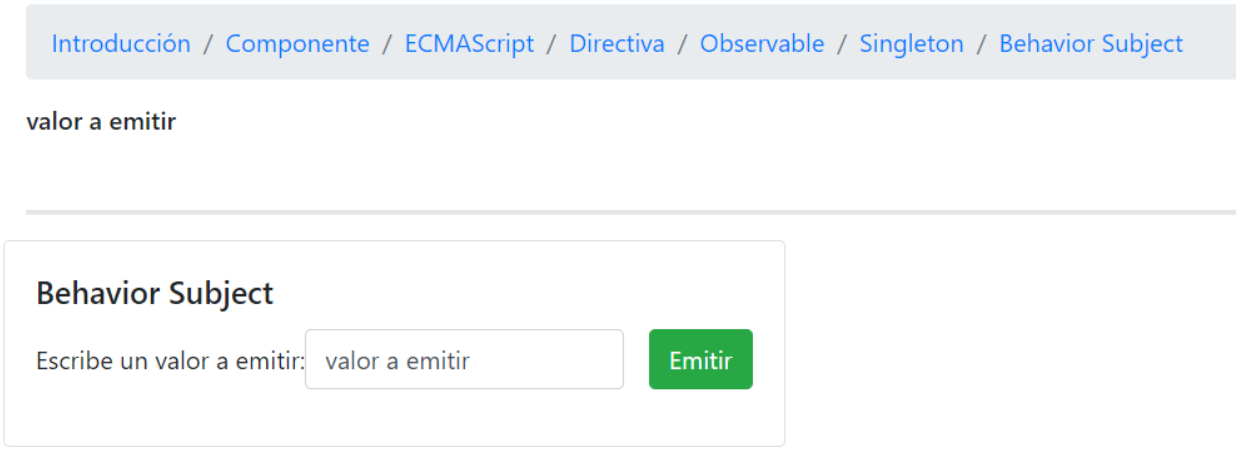


Figura 49. Módulo Behavior Subject en la aplicación

El módulo Singleton nos muestra como transmitir información entre componentes aprovechando el patrón de diseño Singleton.

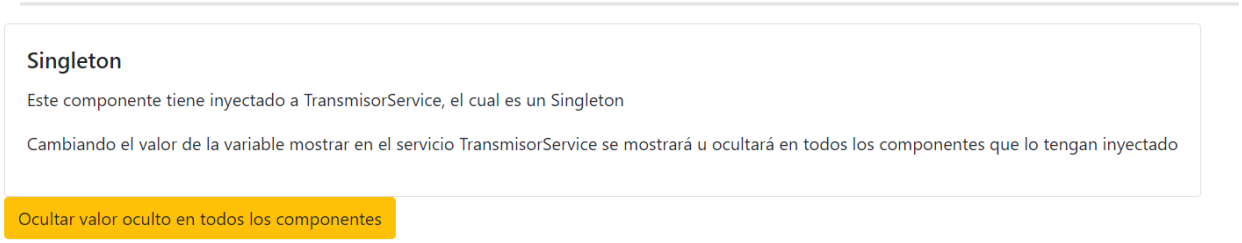


Figura 50.Módulo Singleton en la aplicación

7. BIBLIOGRAFÍA

[1] Documentación oficial de Angular: https://angular.io/docs

[2] Robert C. Martin. (2018). Arquitectura Limpia. Guía para especialistas en la estructura y el diseño software.

[3] Robert C. Martin. (2009). Código Limpio. Manual de estilo para el desarrollo ágil de software.

[4] ECMAScript 6 - ECMAScript 2015: https://www.w3schools.com/js/js_es6.asp

[5] Documentación oficial de TypeScript: https://www.typescriptlang.org/docs/

[6] DIEGO BOSCAN. Decoradores en TypeScript http://diegoboscan.com/decoradores-en-typescript/

[7] Fernando Herrera, "Typescript: tu completa guía y manual de mano" https://www.udemy.com/typescript-guia-completa/learn/v4/overview

[8] Fernando Herrera, "Angular: De cero a experto creando aplicaciones" https://www.udemy.com/angular-2-fernando-herrera/learn/v4/overview

[9] Braulio Diez. Introducción a la programación funcional en JavaScript: https://lemoncode.net/lemoncode-blog/2017/9/5/introduccion-programacion-funcional-javascript