ANGULAR 8

Jonatan Lucas Molina
jonatan.lucas@centic.es
@Jon_Lucas_

Index

1.- Introducción

Esta guía mostrará cómo construir y ejecutar una aplicación en Angular para su versión 8. Se revisarán los conceptos básicos para poder ejecutar una aplicación con Angular y al final de esta guía seremos capaces de hacer lo siguiente:

- Usar las directivas de Angular para mostrar y ocultar elementos.
- Crear componentes en Angular para trabajar con los datos.
- Usar *one-way data binding* para mostrar datos.
- Añadir campos editables para actualizar un modelo con *two-way data binding*.
- Enlazar métodos de los componentes para usar eventos.
- Mostrar datos con *pipes*.
- Crear servicios para recuperar datos del servidor.
- Usar *routing* para navegar por las diferentes vistas y sus componentes.
- Crear una aplicación multimodular.
- Crear formularios.
- Dar estilo a nuestra aplicación

1.1.- ¿Qué es una página Single-Page application (SPA)?

Se trata de una aplicación web o sitio web donde solamente se carga el contenido de dicha web en la primera petición al servidor: HTML, CSS y Javascript. Con esto se consigue dar una experiencia más fluida a los usuarios como una aplicación de escritorio.

Los retos a los que se enfrenta una aplicación SPA son:

- El enrutado: saber qué contenido mostrar dependiendo de la URL.
- Interfaz de usuario dinámica: posibilidad de actualizar la página cuando recibamos nuevos datos del servidor de manera fluida y rápida sin que dé la sensación de que se ha cargado una nueva página.
- Acceso a datos ya sean en loca, localStorage y sessionStorage, o peticiones a servidor.

1.2.- Pero.. ¿porqué Angular?

Angular es un *framework* desarrollado en *Typescript* de código abierto que construye aplicaciones en HTML y JavaScript. Fue desarrollado por Google. Este *framework* se utilizó para superar los obstáculos encontrados al trabajar con aplicaciones *Single Page*. El lanzamiento inicial de este *framework* fue en octubre de 2010.

Principales características de Angular:

- Velocidad y rendimiento.

    - El código de la app se puede convertir en código optimizado con las herramientas que Angular posee, ejemplo Ivy.
    - Se puede ejecutar bajo cualquier servidor que renderice una web, el más común y usado es node.js
    - División del código para que la web solo cargue el código de la vista que se esté visualizando en ese momento.
- Productividad.
    - Creación de componentes, módulos o servicios de forma rápida.
    - Angular CLI. Herramienta de línea de comandos que permite crear proyectos nuevos, elementos y realizar test.
    - IDEs. Integración con la mayoría de editores populares.

1.3.- Requisitos previos

1.3.1.- Typescript

Typescript es un lenguaje de programación libre y de código abierto desarrollado por Microsoft. Extiende la sintaxis de Javascript, esencialmente añade tipos estáticos y objetos basados en clases, esto quiere decir que pasamos de un lenguaje abierto a todo a un lenguaje más tipado y orientado a objetos.

Pero al tratarse de un superconjunto de Javascript podremos hacer funcionar cualquier código de Javascript en Typescript. Esto quiere decir que podremos crear proyectos al más puro estilo de Javascript o crear proyectos más tipados y orientados a objetos.

Tipos básicos

String	Number	Array
Tuple (similar al Array)	Enum	Any
Void	Boolean	Never (valores que nunca se producen)

String and Number

```
let color: string = "blue";
color = 'red';
```

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`;
```

```
let sentence: string = "Hello, my name is " + fullName + ".\n\n" +
    "I'll be " + (age + 1) + " years old next month.";
```

Array

```
let list: number[] = [1, 2, 3];
```

```
let list: Array<number> = [1, 2, 3];
```

Tuple

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

```
console.log(x[0].substring(1)); // OK
console.log(x[1].substring(1)); // Error, 'number' does not have 'substring'
```

```
x[3] = "world"; // Error, Property '3' does not exist on type '[string, number]'.

console.log(x[5].toString()); // Error, Property '5' does not exist on type '[string,
number]'.
```

Any

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

Clases

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");
```

```
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}

class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}

const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

Funciones

```
// Named function
function add(x, y) {
    return x + y;
}

// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

```
function add(x: number, y: number): number {
    return x + y;
}

let myAdd = function(x: number, y: number): number { return x + y; };
```

Parámetros opcionales

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}

let result1 = buildName("Bob");                  // works correctly now
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams");         // ah, just right
```

1.3.2.- Instalar Node.js y Angular 8

Para poder trabajar con Angular 8 debemos tener instalado:

- Node.js en versiones superiores a la 10.9.0. Se aconseja usar el gestor nvm, node version manager, para instalar la versión de node.js que necesitemos.
- Npm, node package manager, que se instala con la versión de Node.js

Instalar Angular CLI

```
npm install -g @angular/cli@8
```

Angular CLI

Angular CLI es una interfaz de línea de comandos que nos permitirá inicializar, crear, desarrollar una aplicación de Angular así como realizar pruebas y el despliegue de nuestra aplicación.

Los comandos más usados son:

add	Añade soporte para una librería externa	build	Compila nuestra aplicación para su uso en un servidor
config	Añade o modifica los valores de configuración de Angular, también se pueden ver en el angular.json	generate	Añade o modifica archivos
help		new	Crea un nuevo proyecto
serve	Construye y ejecuta la aplicación a nuestro servidor local	test	
update			

Crear un proyecto nuevo

```
ng new cursoAngular8
```

En la instalación de nuestro nuevo proyecto nos irá marcando las diferentes características que queremos añadir o no.

1.3.3.- Ejecutar una aplicación

```
ng serve
```

El comando anterior ejecutará el servidor y permitirá construir y ejecutar las aplicaciones en Angular y reconstruirlas cuando hagamos cambios para que se reflejen.

El servidor se lanzará bajo la dirección http://localhost:4200/ por defecto pero podremos cambiar este valor añadiendo al comando lo siguiente:

```
ng serve --port 4210
```

2.- Arquitectura de Angular

Angular es un framework escrita en Typescript que junto a ficheros HTML crean una aplicación.

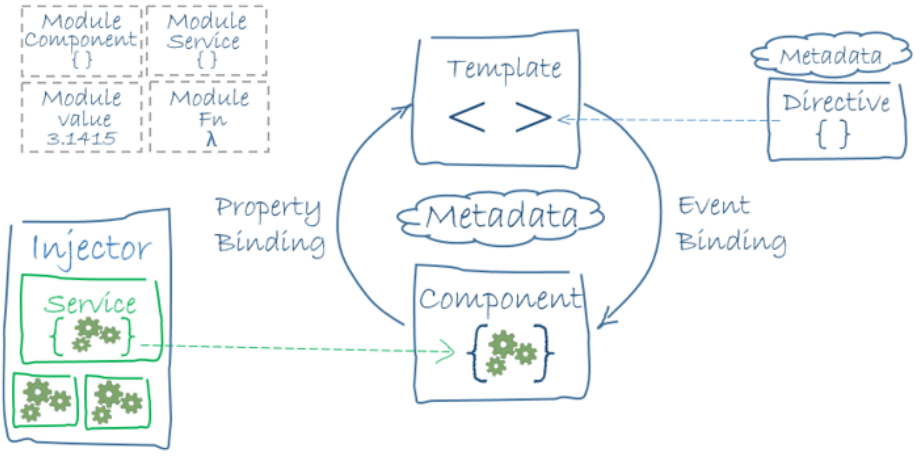
La construcción básica en Angular son los NgModules, los cuales proveen un contexto de compilación para los Components. NgModules recopilan el código en conjuntos funcionales; una app en Angular está definida como un conjunto de NgModules y siempre hay un módulo raíz que capacita el bootstrapping, o enlazamiento, y que contiene más módulos.

- Los Componentes definen vistas, las cuales son un conjunto de elementos visuales que Angular puede elegir y modificar de acuerdo a su lógica de programación y datos.
- Los Componentes usan servicios, services, los cuales proveen funcionalidades específicas no relacionadas directamente con la vista. Los servicios son inyectados en los componentes como dependencias, lo que hace que el código sea más eficiente.

Los componentes y servicios son simplemente clases, que proveen metadata que Angular les dice como usar.

- El metadata para una clase component se asocia con una plantilla, template, que es definida en una vista. Una plantilla combina HTML con directivas de Angular y la funcionalidad de binding que permite a Angular modificar el HTML antes de renderizarlo para mostrar la vista.
- El metadata para una clase service provee la información que Angular necesita para capacitar a los componentes de información a través de las inyecciones de dependencia, dependency injection.

Una app en Angular está compuesta por muchos componentes y vistas, por lo que para poder navegar entre ellas Angular provee un módulo llamado Router que ayuda a la navegación entre las mismas.

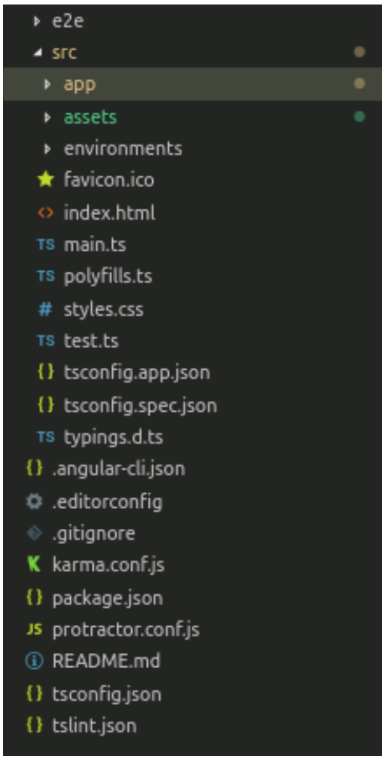


- Juntos, un componente y una plantilla definen una vista en Angular.
    - Un ayudante en la clase del componente que añade el metadata, para asociar la plantilla.

    - Directivas y binding en las plantillas de los componentes modifican las vistas basándose en la lógica de programación e información.
- El inyector de dependencias provee los services a los componentes, como por ejemplo el router service que permite navegar entre vistas.

2.1.- Estructura de un proyecto en Angular

2.1.1.- Estructura inicial



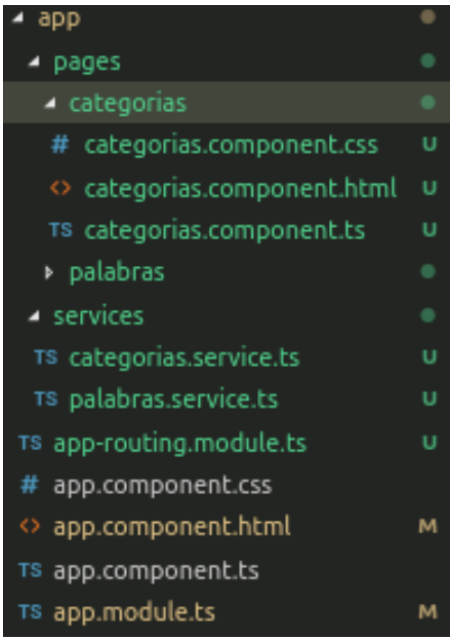
De los siguientes archivos, los más importantes son:

- Index.html: como toda aplicación web, se necesita este archivo para acceder a la aplicación. Pero en un proyecto en Angular no es necesario tocar nada de él salvo para algo de CSS como por ejemplo colocar un footer con imágenes.
- Main.ts: Este archivo rara vez se toca para una aplicación. Su función consiste en lanzar el módulo raíz, el app.module.ts, para poder ejecutar la aplicación.
- Styles.css: archivo principal de CSS de nuestra aplicación.
- Favicon.ico: la imágen de nuestra aplicación en el navegador web.

Y de carpetas las siguientes:

- App: contiene toda la aplicación.
- Assets: esta carpeta es para los archivos estáticos.

2.1.2.- Dentro de la carpeta app



- App.module.ts: módulo raíz de nuestra aplicación. Aquí se cargan los componentes, servicios y módulos de angular que vayamos a usar.
- App.component: es el componente principal, el primer componente que se carga y necesitamos para trabajar.
- App-routing.module.ts: archivo para crear las rutas internas de Angular para trabajar con la aplicación.
- Pages: carpeta creada para almacenar cada una de las vistas creadas para nuestro proyecto de Angular.
- Services: carpeta creada para almacenar nuestros archivos services.

2.2.- Módulos

Las aplicaciones en Angular son modulares y Angular tiene su propio sistema de modulación llamado NgModules. NgModules son contenedores para una cohesión de código dedicado a una app, un flujo de trabajo, o funciones relacionadas de capacidades. Pueden contener componentes, servicios y otros archivos de código necesario para nuestra app.

Cada app de Angular contiene al menos un NgModule, el módulo raíz, normalmente llamado AppModule en el archivo también llamado app.module.ts. La app se lanzará enlazando con este módulo raíz.

Si bien una pequeña aplicación puede tener solo un NgModule, la mayoría de apps tienen muchos más módulos. El módulo raíz se llama así porque puede contener muchos más NgModules secundarios.

2.2.1.- NgModule metadata

Un NgModule se define como una clase enlazada con @NgModule(). El @NgModule() es una función que toma un objeto de metadatos para describir las propiedades de un módulo. Las propiedades más importantes son:

- declarations: Los componentes, directivas y pipes que pertenecen a ese NgModule.
- exports: El subconjunto de declaraciones que deben ser visibles y utilizables en las plantillas de los componentes de otros NgModules.

- imports: Otros módulos cuyas clases exportadas son necesarias para las plantillas de los componentes de este NgModule.
- providers: Colección de los services que se necesitan en este NgModule.
- bootstrap: Solo el módulo raíz deberá almacenar esta propiedad.

Ejemplo de definición de root NgModule :

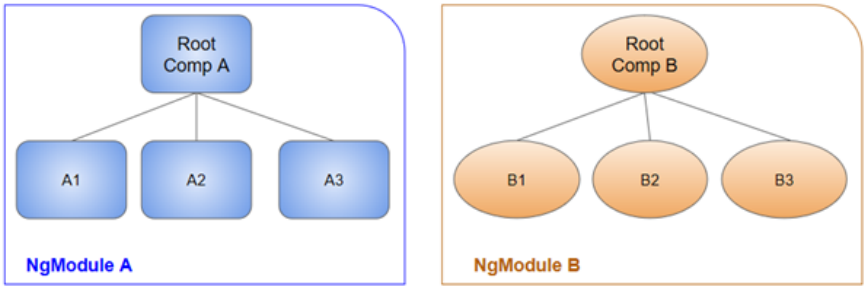
```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})

export class AppModule { }
```

Ejemplo de un subconjunto de NgModule:

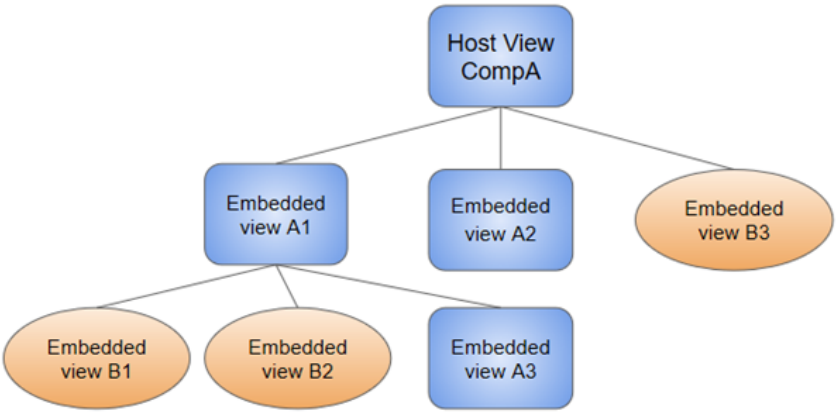
```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ NameService ],
  declarations: [ NameComponent ],
  exports:      [ NameComponent ]
})

export class NameModule { }
```

2.2.2.- NgModule y componentes

NgModule proporciona un contexto de compilación para sus componentes. Un NgModule raíz siempre tiene un componente raíz que se crea durante el arranque, pero cualquier NgModule puede incluir cualquier número de componentes adicionales, que se pueden cargar a través del enrutador o crear a través de la plantilla. Los componentes que pertenecen a un NgModule comparten un contexto de compilación.



Un componente y su plantilla juntos definen una vista de Angular. Un componente puede contener una vista principal, con la cual se permite definir áreas más complejas de una vista que pueden ser creadas, modificadas y destruidas como una unidad. Una vista jerarquizada puede combinar varias vistas definidas en componentes que pertenecen a diferentes NgModules.



Cuando se crea un componente, se asocia directamente con una vista única. Esta vista puede ser raíz de una serie de vistas de otros componentes. Esos componentes pueden estar en el mismo NgModule, o pueden ser importadas desde otros módulos.

2.3.- Componentes

Cada componente controla una vista. Definen la lógica de la aplicación dentro de una clase. Esta clase interactúa con la vista a través de una API de propiedades y métodos.

```
export class HeroListComponent implements OnInit {
  //Variables
  heroes: Hero[];
  selectedHero: Hero;

  constructor(private service: HeroService) { }

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  selectHero(hero: Hero) {
     this.selectedHero = hero;
  }
}
```

Angular crea, actualiza y destruye componentes a medida que el usuario se mueve a través de la aplicación.

Para crear un componente Angular CLI posee un comando que generará una carpeta con los archivos básicos necesarios para una vista en Angular y lo introducirá en el @NgModule principal.

```
ng generate component name_component
```

Tras este comando se generará una carpeta con el nombre para el componente los siguientes archivos:

- Un fichero HTML para la plantilla.
- Un fichero TS para el componente con el código básico de un componente.
- Un fichero CSS para el diseño.

2.3.1.- Metadatos de un componente

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ],
  styleUrls: [‘./hero-common.css’ ]
})

export class HeroListComponent implements OnInit {
/* . . . */
}
```

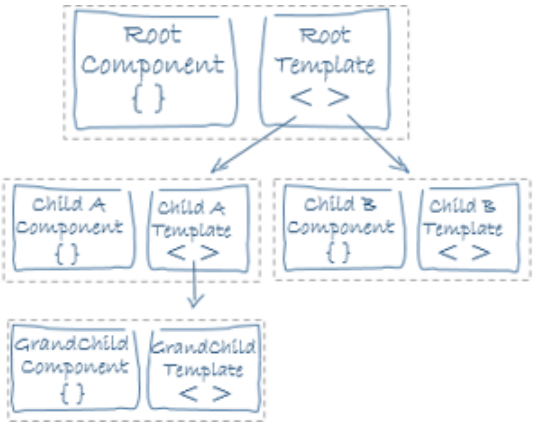
En el código de ejemplo se puede ver como un componente no es más que una clase, sin ningún tipo de anotación o sintaxis especial. Pero no se puede trabajar con él como un componente hasta que no creamos el decorador @Component()

Los metadatos de un componente le indican a Angular dónde obtener los principales bloques de construcción que se necesitan para crear y presentar el componente y su vista. En particular, asocia una plantilla con el componente. Juntos describen y muestran una vista de Angular. Además los metadatos también pueden configurar como se puede hacer referencia al componente en HTML y qué servicios requiere.

Algunas de las opciones más útiles son:

- selector: Un selector CSS que le dice a Angular que cree e inserte una instancia de este componente donde encuentre la etiqueta correspondiente en la plantilla HTML.
- templateUrl: Carga la plantilla que se usará para este componente. Alternativamente se puede generar el CSS directamente a través de la opción template. Pero se recomienda la opción de un archivo diferente para una mejor estructuración.
- providers: Introducimos los servicios necesarios para el componente.
- sytleUrls: Carga los ficheros CSS en nuestra plantilla.

2.3.2.- Plantillas y vistas



Una plantilla es una forma de HTML que le dice a Angular como representar el componente.

Las vistas suelen estar ordenadas jerárquicamente, lo que le permite modificar o mostrar y ocultar secciones o páginas completas como una unidad.

La plantilla inmediatamente asociada a un componente define la vista principal de ese componente. El componente también puede definir una jerarquía de vistas, que vistas anidadas, alojadas por otros componentes.

Una vista puede incluir vistas de componentes en el mismo NgModule, pero también puede incluir vistas de componentes que están definidos en diferentes NgModules.

2.3.3.- Sintaxis de una plantilla

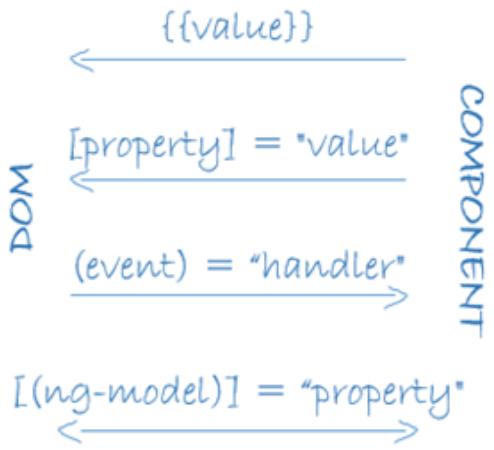
Una plantilla es como un archivo de HTML, a excepción que este contiene sintaxis de Angular, con el cual se podrá alterar el HTML a través de la lógica de la app. La plantilla puede usar data binding para coordinar la app y el DOM, los pipes transforman la información antes de ser mostrada y las directivas aplican la lógica a lo que se muestra.

```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

<app-hero-detail *ngIf="selectedHero" [hero]="selectedHero"></app-hero-detail>
```

Esta plantilla usa elementos comunes de HTML como <h2> y <p> combinados con elementos de Angular como *ngFor, {{hero.name}}, (click), [hero] y <app-hero-detail>. La sintaxis de Angular dirá como mostrar los elementos de HTML, usando la lógica de la app y los datos.

- *ngFor dice a Angular como interactuar con una lista de elementos.
- {{hero.name}}, (click) y [hero] enlaza los datos a mostrar con el DOM.
- <app-hero-detail> es un elemento que representa una plantilla diferente con su componente correspondiente.

2.3.4.- Data binding



Sin un framework, seríamos responsables de enlazar la información en el HTML y los componentes y viceversa, lo que supondría una tarea pesada.

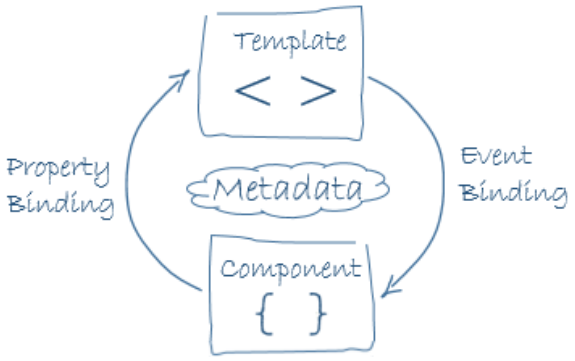
Angular soporta two-way data binding, un mecanismo que coordina las partes de una plantilla con la lógica de un componente y viceversa para trabajar con los datos de una forma más sencilla y ágil.

```
<li>{{hero.name}}</li>

**getVal() devuelve el valor de 4**
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}.</p>

<app-hero-detail [hero]="selectedHero"></app-hero-detail>

<img [src]="itemImageUrl2">

<button [class.special]="isSpecial" [style.color]="isSpecial ? 'red': 'green'" >Special</button>

<li (click)="selectHero(hero)"></li>
```

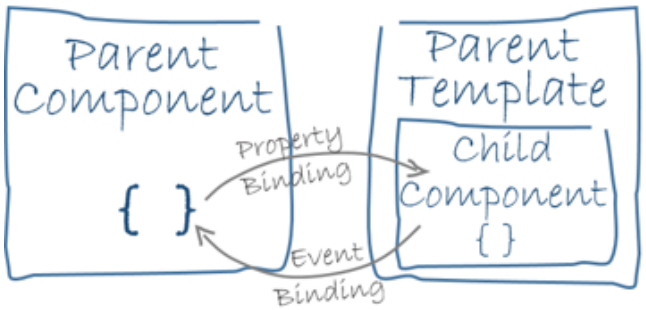
- {{hero.name}} muestra la información del componente relacionada con el objeto hero.name.
- [hero] enlaza el valor de la propiedad selectedHero del componente HeroListComponent al objeto hero del componente HeroDetailComponent.
- (click) enlace de evento. Llama al método del componente cuando el usuario hace clic en el nombre de un elemento.

```
<input [(ngModel)]="hero.name">
```

- Two-way data binding, enlaza los datos asociados a ese elemento de tal manera que si se actualizan en el componente también lo hacen en la plantillas.



El enlace de datos juega un papel importante en la comunicación entre una plantilla y su componente, y también es importante para la comunicación entre los componentes principal y secundario.



Bindeo de propiedad o interpolación

Normalmente podemos elegir entre ambas. Por ejemplo:

```
<p><img src="{{heroImageUrl}}"> is the <i>interpolated</i> image.</p>
<p><img [src]="heroImageUrl"> is the <i>property bound</i> image.</p>

<p><span>"{{title}}" is the <i>interpolated</i> title.</span></p>
<p>"<span [innerHTML]="title"></span>" is the <i>property bound</i> title.</p>
```

Interpolar {{}} es una alternativa al bindeo de propiedad []. No hay una regla que defina cuándo usar uno u otro con valores string. Pero sí que debemos usar el bindeo de propiedad para enlazar con valores no-string.

2.3.5.- Pipes

Las “tuberías” de Angular permiten transformar el valor de visualización de un elemento en la plantilla de HTML.

Angular define varias pipes, las mas conocidas son las de fecha y de moneda. Para una lista completa visitar la lista de API de pipes. También se pueden definir nuevas tuberías.

Para especificar dicha transformación se usará el operador de pipe:

```
{{ interpolated_value | pipe_name }}
```

Ejemplos de uso de pipe:

```
<!-- Default format: output 'Jun 15, 2015'-->
 <p>Today is {{today | date}}</p>

<!-- fullDate format: output 'Monday, June 15, 2015'-->
<p>The date is {{today | date:'fullDate'}}</p>

 <!-- shortTime format: output '9:43 AM'-->
 <p>The time is {{today | date:'shortTime'}}</p>
```

2.3.6.- Directivas

Las plantillas de Angular son dinámicas. Cuando Angular las procesa, transforma el DOM de acuerdo con las instrucciones dadas por las directivas. Una directiva es una clase con un decorador @Directive().

Un componente es técnicamente una directiva. Además existen otros dos tipos de directivas: estructurales y de atributo.

Al igual que para los componentes, los metadatos de una directiva asocian la clase con un selector que se usa para insertarlo en HTML. En las plantillas las directivas suelen aparecer dentro de una etiqueta como atributo de un elemento.

Directivas estructurales

Las directivas estructurales modifican el diseño al agregar, eliminar y reemplazar elementos en el DOM. La plantilla del ejemplo utiliza dos directivas integradas para agregar la lógica de la aplicación a cómo se representa la vista.

```
<li *ngFor="let hero of heroes"></li>

<app-hero-detail *ngIf="selectedHero"></app-hero-detail>
```

- *ngFor es un iterativo; le dice a Angular que muestre e un <li> por cada héroe de la lista.
- *ngIf es un condicional: incluye el componente HeroDetail solo si exite un héroe seleccionado.

Directivas de atributos

Las directivas de atributos alteran la apariencia o el comportamiento de un elemento existente. En las plantillas, se ven como atributo HTML normales.

La directiva ngModel que implementa enlace de datos bidireccional, es un claro ejemplo. NgModel modifica el comportamiento de un elemento existente configurando su propiedad de valor de visualización y respondiendo a los eventos que generen algún cambio.

```
<input [(ngModel)]="hero.name">
```

Otras directivas de atributo pueden ser:

- ngSwitch: altera la estructura del diseño.
- ngStyle o ngClass: modifican los elementos y componentes del DOM.

2.4.- Servicios e inyector de dependencias

Un servicio, service, es una clase amplia que abarca cualquier valor, función o característica que necesita una app. Un servicio está bien definido y con un propósito claro. Debería hacer algo específico y hacerlo bien.

Angular distingue los componentes de los servicios para aumentar la modularidad y la reutilización. Al separar la funcionalidad relacionada con la vista de un componente de otros tipos de procesamiento, puede hacer que sus clases de componentes sean ágiles y eficientes.

Un componente delega ciertas tareas a los servicios, como recuperar datos del servidor, validar la entrada del usuario o iniciar sesión directamente en la consola. Al definir dichas tareas de procesamiento en una clase de servicio inyectable, las pone a disposición de cualquier componente.

Angular ayuda a los componentes a facilitar la incorporación de la lógica de los servicios a través de la inyección de dependencia.

Como en el caso de los componente, Angular CLI tiene un comando por el que se generará un archivo básico de service con el nombre del parámetro que le pasemos:

```
ng generate service name_service
```

2.4.1.- Ejemplos de servicios

El siguiente ejemplo es un servicio que muestra los mensajes de la consola del navegador.

```
export class Logger {

  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }

}
```

Los servicios pueden depender de otros servicios.

```
export class HeroService {
  private heroes: Hero[] = [];

  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  getHeroes() {
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {
      this.logger.log(`Fetched ${heroes.length} heroes.`);
      this.heroes.push(...heroes); // fill cache
    });
    return this.heroes; }}
```

2.4.2.- Inyección de dependencias (ID)

ID está conectado al marco de Angular y se usa en todas partes para proporcionar nuevos enlaces de componentes con sus servicios y otras cosas que necesitan.

Para definir una clase como un servicio en Angular, se usa el decorador @Injectable() para proporcionar los metadatos que permiten a Angular inyectarlo en un componente como una dependencia.

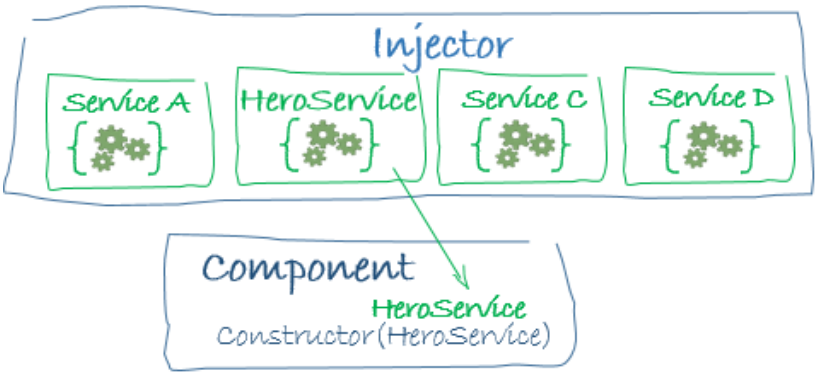
- El inyector es el mecanismo principal. Angular crea un inyector para toda la app durante el proceso de arranque, e inyectores adicionales según sea necesario. No se tienen que crear a mano en ningún momento.
- Un inyector crea dependencias y mantiene un contenedor de instancias de dependencia que reutiliza si es posible.
- Un provider es un objeto que le dice a un inyector cómo obtener o crear una dependencia.

Cuando Angular crea una instancia de una clase de componente, determina qué servicios u otras dependencias necesita ese componente al observar los tipos de parámetros del constructor.

Cuando Angular descubre que un componente depende de un servicio, primero verifica si el inyector tiene alguna instancia existente de ese servicio. Si una instancia de servicio solicitada aún no existe, el inyector hace que el provider agregue al inyector el servicio antes de ser devuelto.

Cuando todos los servicios solicitados se han resuelto y devuelto, Angular puede llamar al constructor del componente con esos servicios como argumentos.

El ejemplo de inyección se ve algo así:



2.4.3.- Proporcionando servicios

Debe registrar al menos un provider de cualquier servicio que se vaya a usar. El proveedor puede ser parte de los propios metadatos del servicio, haciendo que ese servicio esté disponible en todas partes, o puede registrar proveedores con módulos o componentes específicos. Se registran proveedores en los metadatos de los servicios (en el decorador @Injectable()), o en los metadatos de @NgModule() o @Component().

- Por defecto, el comando de Angular CLI ng generate service registra un proveedor con el inyector raíz para su servicio al incluir metadatos del proveedor en el decorador @Injectable().

```
@Injectable({
 providedIn: 'root',
})
```

Cuando se proporciona el servicio en el nivel del proveedor raíz, Angular crea una instancia única y compartida de dicho servicio y la inyecta en cualquier clase que lo solicite. El registro del proveedor en los metadatos también le permite a Angular optimizar una app eliminando el servicio de la aplicación compilada si no se utiliza.

- Cuando se registra un proveedor con un NgModule específico, la misma instancia de un servicio está disponible para todos los componentes en ese NgModule.

```
@NgModule({
  providers: [
  BackendService,
  Logger
 ],
 ...
})
```

- Cuando se registra un proveedor a nivel de componente, obtiene una nueva instancia del servicio con cada nueva instancia de ese componente.

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
```

2.5.- HTTP

La mayoría de las aplicaciones creadas con Angular 2 se comunican a través de los services con un back-end que les proporciona la información necesaria a través del protocolo HTTP. Los navegadores modernos admiten dos APIs para realizar las peticiones HTTP: la interfaz XMLHttpRequest y fetch() API.

En Angular encontramos un objeto para simplificar las peticiones llamado HttpClient en el paquete de @angular/common/http que se basa en la interfaz XMLHttpRequest expuesta por los navegadores. Pero este objeto HttpClient incluye características de capacidad de prueba, objetos de solicitud y respuesta, interpretación de solicitud y respuesta, Observable api y manejo de errores.

2.5.1.- Setup

Lo primero que se debe hacer para poder utilizar HttpClient es importar el modulo de HttpClientModule. Como se trata de un objeto que se usará bastante se colocará en el AppModule raíz.

```
import { NgModule }         from '@angular/core';
import { BrowserModule }    from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})

export class AppModule {}
```

Ahora ya podremos utilizarlo en cualquier service que lo necesitemos.

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

@Injectable()
export class ConfigService {

  constructor(private http: HttpClient) { }

}
```

2.5.2.- JSON data

La mejor forma de recuperar información de un servidor será en formato JSON, por lo que el servidor al que se estén realizando las peticiones deberá estar configurado para que devuelva los datos en este formato.

Un ejemplo de petición sería:

Config.service.ts

```
configUrl = 'url_server';

getConfig() {
  return this.http.get(this.configUrl);
}
```

Config.component.ts

```
showConfig() {
  this.configService.getConfig()
    .subscribe((data: Config) => this.config = {
        heroesUrl: data['heroesUrl'],
        textfile:  data['textfile']
    },
    (error) => console.log(error)
    );
}
```

Un método de HttpClient no comienza su solicitud HTTP hasta que llama al método .subscribe().

2.5.3.- Tipo de peticiones

Además de obtener datos del servidor, *HttpClient* admite solicitudes de PUT, POST y DELETE.

2.5.3.1.- Añadir Headers

Muchos servidores requieren encabezados adicionales para operaciones de guardado. Un ejemplo sería como el siguiente donde enviamos un token de autorización y el tipo de contenido.

```
import { HttpHeaders } from '@angular/common/http';

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type':  'application/json',
    'Authorization': 'my-auth-token'
  })
};
```

2.5.3.2.- Peticiones POST

```
/** POST: add a new hero to the database */
addHero (hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('addHero', hero))
    );
}
```

El método de *.post()* es similar al de *.get()* pero necesita dos parámetros más:

- Los datos a enviar en la petición POST en el cuerpo de la solicitud.
- El objeto *httpOptions,* opciones de método que se almacenan en el *headers*.

2.5.3.3.- Peticiones DELETE

Este método elimina un objeto del servidor cuando le pasamos el id del mismo.

```
/** DELETE: delete the hero from the server */
deleteHero (id: number): Observable<{}> {
  const url = `${this.heroesUrl}/${id}`; // DELETE api/heroes/42
  return this.http.delete(url, httpOptions)
    .pipe(
      catchError(this.handleError('deleteHero'))
    );
}
```

2.5.3.4.- Peticiones PUT

Una aplicación enviará una solicitud PUT para actualizar completamente un recurso.

```
/** PUT: update the hero on the server. Returns the updated hero upon success. */
updateHero (hero: Hero): Observable<Hero> {
  return this.http.put<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('updateHero', hero))
    );
}
```

2.6.- Routing

El router de Angular permite la navegación de una vista a la siguiente a medida que los usuarios realizan tareas de la aplicación. Puede interpretar una URL del navegador como una instrucción para navegar a una vista generada por el cliente. Puede pasar parámetros opcionales. El router registra la actividad en el diario de historial del navegador para que los botones de avance y retroceso también funcionen.

Antes de comenzar a introducir rutas en nuestra aplicación debemos configurar la app para que el router de Angular pueda trabajar y crear un fichero donde almacenar dichas rutas.

En el archivo index.html, justo debajo del comienzo de la etiqueta <head> se debe colocar:

```
<base href=”/”>
```

Esta etiqueta le indicará al router cómo redactar las URLs de navegación.

En Angular, la mejor forma de cargar y configurar las rutas es en un archivo aparte, a nivel del app.module.ts que se dedique a esta función solamente y sea cargado por el AppModule. Para crear este archivo usaremos el siguiente comando de Angular CLI:

```
ng generate module app-routing --flat --module=app
```

Este comando genera un archivo app-routing.module.ts con la clase AppRoutingModule. El archivo generado será algo parecido al siguiente código:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})

export class AppRoutingModule { }
```

Pero este no es el código que se necesita verdaderamente para generar las rutas de nuestra app. Sustituiremos parte del código y quedará como sigue:

```
import { NgModule }             from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

@NgModule({
  exports: [ RouterModule ]
})

export class AppRoutingModule {}
```

2.6.1.- Añadiendo rutas

Las rutas le indican al router qué vistas mostrar cuando un usuario hace clic en un enlace. Cada dirección tiene dos propiedades mínimo:

- path: una cadena que representa la url a visitar.
- component: el componente que el router debe cargar al navegar a esa ruta.

Siguiendo con el ejemplo anterior ahora nuestro archivo quedaría así:

```
import { NgModule }             from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { HeroesComponent } from ‘./heroes/heroes.component’;
import { HeroDetailComponent  } from ‘./heroDetail/heroDetail.component’;
import { PageNotFoundComponet } from ‘./page/page.component’;

const   routes:   Routes =  [
   {   path: ‘heroes’, component:  HeroesComponent  },
   {   path: ‘heroes/:id’, component:  HeroDetailComponent   },
   {   path: ‘’, redirecTo:  ‘/heroes’, pathMatch:  ‘full’  },
   {   path: ‘**’, component:  PageNotFoundComponent  }
];

@NgModule({
  exports: [ RouterModule ],
  imports: [ RouterModule.forRoot(routes) ]
})
export class AppRoutingModule {}
```

El método .forRoot() se llama así porque configura el router en la raíz de la app. Este método proporciona los providers y las directivas necesarias para el enrutamiento y realiza la navegación inicial según la url actual.

2.6.2.- Añadir RouterOutlet

En la vista de app.component.html debemos introducir la etiqueta router-outlet que se encarga de mostrar las vistas de los componentes cargados en el router. Aunque lo normal es dejar solo este archivo HTML para esta etiqueta se puede trabajar en este archivo como se crea conveniente para el proyecto.

```
<h1>{{title}}</h1>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

2.6.3.- Navegación entre vistas

Para navegar entre diferentes vistas Angular posee varias formas de realizar esta tarea:

- Con el atributo routerLink en una etiqueta en HTML.
- Con la clase router desde el componente a través de código.

2.6.4.- Extraer parámetros de la url

Si una de las url creadas en el archivo de rutas posee algún parámetro como la siguiente:

```
{ path:  ‘heroes/:id’, component:  HeroDetailComponent  }
```

La forma de recuperar ese valor en la vista de HeroDetailComponent será:

```
import { ActivatedRoute } from '@angular/router';

…

constructor(…, private activatedRoute: ActivatedRoute ) {
  activatedRoute.params.subscribe( params => {
      this.id = +params['id'];
  });
  …
}
…
```

Los parámetros de router son siempre string. Para convertirlos a número se coloca el operador + delante para hacer la conversión.

3.- Relación entre componentes

Repasaremos las opciones más importantes que tienen los componentes de Angular para comunicarse entre ellos.

3.1.- Comunicación padre-hijo a través de la etiqueta *@input*

Tenemos nuestra clase 'Padre':

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

import { ArrayObjects } from './classObject';

@Component({
  selector: 'app-parent',
  template: `
    <h2>{{master}} controls {{arrayObjects.length}} numbers</h2>
    <app-child *ngFor="let element of arrayObjects"
      [objectChild]="element"
      [master]="master">
    </app-child>
  `
})
export class ParentComponent {
  arrayObjects = ArrayObjects;
  master = 'Master';
}
```

Y nuestra clase 'Hijo':

```
import { Component, Input } from '@angular/core';

import { Object } from './classObject';

@Component({
  selector: 'app-child',
  template: `
    <h3>{{objectChild.name}} says:</h3>
    <p>I, {{objectChild.name}}, am at your service, {{masterName}}.</p>
  `
})
export class ChildComponent {
  @Input() objectChild: Object;
  @Input('master') masterName: string;
}
```

Como se ha podido comprobar la forma que tienen estos dos componentes es a través de la etiqueta @Input pero ¿qué quiere decir esto? Cuando colocamos esta etiqueta a un componente se

mantiene a la escucha de alguna variable que cumpla con los requisitos puestos, en este caso el primero es de una clase Object y el segundo es un string.

Una de las maneras de pasar al componente la información necesaria para recogerla en los @Input es colocando dicha información como atributo de la etiqueta del componente, como se vé en la etiqueta de ChildComponent dentro del HTML del padre.

3.2.- Comunicación entre componentes con *ngOnChanges()*

Tenemos nuestra clase 'Padre':

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

@Component({
  selector: 'app-version-parent',
  template: `
    <h2>Source code version</h2>
    <button (click)="newMinor()">New minor version</button>
    <button (click)="newMajor()">New major version</button>
    <app-version-child [major]="major" [minor]="minor"></app-version-child>
  `
})
export class VersionParentComponent {
  major = 1;
  minor = 23;

  newMinor() {
    this.minor++;
  }

  newMajor() {
    this.major++;
    this.minor = 0;
  }
}
```

Y tenemos nuestra clase 'Hijo':

```
import { Component, Input, OnChanges, SimpleChange } from '@angular/core';

@Component({
  selector: 'app-version-child',
  template: `
    <h3>Version {{major}}.{{minor}}</h3>
    <h4>Change log:</h4>
    <ul>
      <li *ngFor="let change of changeLog">{{change}}</li>
    </ul>
  `
})
export class VersionChildComponent implements OnChanges {
  @Input() major: number;
  @Input() minor: number;
  changeLog: string[] = [];
```

```
  ngOnChanges(changes: {[propKey: string]: SimpleChange}) {
    let log: string[] = [];
    for (let propName in changes) {
      let changedProp = changes[propName];
      let to = JSON.stringify(changedProp.currentValue);
      if (changedProp.isFirstChange()) {
        log.push(`Initial value of ${propName} set to ${to}`);
      } else {
        let from = JSON.stringify(changedProp.previousValue);
        log.push(`${propName} changed from ${from} to ${to}`);
      }
    }
    this.changeLog.push(log.join(', '));
  }
}
```

En este ejemplo se mantiene la misma forma de comunicación entre padre-hijo a través de la etiqueta @Input pero ahora añadimos la función *ngOnChanges()*. Esta función se implementa al crear la clase igual que el ngOnInit() y actúa cada vez que se detecten cambios en las directivas o atributos de dicho componente.

3.3.- Comunicación Hijo-Padre a través de eventos

Tenemos nuestra clase 'Padre':

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

@Component({
  selector: 'app-vote-taker',
  template: `
    <h2>Should mankind colonize the Universe?</h2>
    <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>
    <app-voter *ngFor="let voter of voters"
      [name]="voter"
      (voted)="onVoted($event)">
    </app-voter>
  `
})
export class VoteTakerComponent {
  agreed = 0;
  disagreed = 0;
  voters = ['Narco', 'Celeritas', 'Bombasto'];

  onVoted(agreed: boolean) {
    agreed ? this.agreed++ : this.disagreed++;
  }
}
```

Y tenemos nuestra clase ‘Hijo’:

```
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'app-voter',
  template: `
    <h4>{{name}}</h4>
    <button (click)="vote(true)"  [disabled]="didVote">Agree</button>
    <button (click)="vote(false)" [disabled]="didVote">Disagree</button>
  `
})
export class VoterComponent {
  @Input()  name: string;
  @Output() voted = new EventEmitter<boolean>();
  didVote = false;

  vote(agreed: boolean) {
    this.voted.emit(agreed);
    this.didVote = true;
  }
}
```

En este ejemplo el padre ‘escucha’ el evento del hijo a través del Objeto EventEmitter. Este objeto lanza el evento, en la etiqueta de dicho componente debemos colocar un atributo con el mismo nombre para capturarlo e igualarlo a alguna función del componente padre para que la lance al escuchar el evento. Para esta funcionalidad se debe de usar el objeto *@Output()* de Angular.

3.4.- Acceso componente Padre a variables del componente Hijo

Tenemos al componente ‘Padre’:

```
import { Component }                from '@angular/core';
import { CountdownTimerComponent }  from './countdown-timer.component';

@Component({
  selector: 'app-countdown-parent-lv',
  template: `
  <h3>Countdown to Liftoff (via local variable)</h3>
  <button (click)="timer.start()">Start</button>
  <button (click)="timer.stop()">Stop</button>
  <div class="seconds">{{timer.seconds}}</div>
  <app-countdown-timer #timer></app-countdown-timer>
  `,
  styleUrls: ['../assets/demo.css']
})
export class CountdownLocalVarParentComponent { }
```

Y tenemos al componente ‘Hijo’:

```
import { Component, OnDestroy, OnInit } from '@angular/core';

@Component({
  selector: 'app-countdown-timer',
  template: '<p>{{message}}</p>'
})
export class CountdownTimerComponent implements OnInit, OnDestroy {

  intervalId = 0;
  message = '';
  seconds = 11;

  clearTimer() { clearInterval(this.intervalId); }

  ngOnInit()    { this.start(); }
  ngOnDestroy() { this.clearTimer(); }

  start() { this.countDown(); }
  stop()  {
    this.clearTimer();
    this.message = `Holding at T-${this.seconds} seconds`;
  }

  private countDown() {
    this.clearTimer();
    this.intervalId = window.setInterval(() => {
      this.seconds -= 1;
      if (this.seconds === 0) {
        this.message = 'Blast off!';
      } else {
        if (this.seconds < 0) { this.seconds = 10; } // reset
        this.message = `T-${this.seconds} seconds and counting`;
      }
    }, 1000);
  }
}
```

De esta manera estamos vinculando una variable en el ‘Padre’ (#timer) para poder utilizar aquellas funciones o variables públicas del ‘Hijo’.

3.5.- Comunicación entre componentes a través de Services

Con este método podremos comunicar dos componentes sin la necesidad de que sean ‘Padre-Hijo’. Primero crearemos un Service que servirá de comunicación entre ambos componentes.

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

```
@Injectable()
export class MissionService {

  // Observable string sources
  private missionAnnouncedSource = new Subject<string>();
  private missionConfirmedSource = new Subject<string>();

  // Observable string streams
  missionAnnounced$ = this.missionAnnouncedSource.asObservable();
  missionConfirmed$ = this.missionConfirmedSource.asObservable();

  // Service message commands
  announceMission(mission: string) {
    this.missionAnnouncedSource.next(mission);
  }

  confirmMission(astronaut: string) {
    this.missionConfirmedSource.next(astronaut);
  }
}
```

Padre:

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

import { MissionService }     from './mission.service';

@Component({
  selector: 'app-mission-control',
  template: `
    <h2>Mission Control</h2>
    <button (click)="announce()">Announce mission</button>
    <app-astronaut *ngFor="let astronaut of astronauts"
      [astronaut]="astronaut">
    </app-astronaut>
    <h3>History</h3>
    <ul>
      <li *ngFor="let event of history">{{event}}</li>
    </ul>
  `,
  providers: [MissionService]
})
export class MissionControlComponent {
  astronauts = ['Lovell', 'Swigert', 'Haise'];
  history: string[] = [];
  missions = ['Fly to the moon!',
              'Fly to mars!',
              'Fly to Vegas!'];
  nextMission = 0;

  constructor(private missionService: MissionService) {
    missionService.missionConfirmed$.subscribe(
      astronaut => {
        this.history.push(`${astronaut} confirmed the mission`);
```

```
    });
  }

  announce() {
    let mission = this.missions[this.nextMission++];
    this.missionService.announceMission(mission);
    this.history.push(`Mission "${mission}" announced`);
    if (this.nextMission >= this.missions.length) { this.nextMission = 0; }
  }
}
```

Hijo:

```
import { Component, Input, OnDestroy } from '@angular/core';

import { MissionService } from './mission.service';
import { Subscription }   from 'rxjs';

@Component({
  selector: 'app-astronaut',
  template: `
    <p>
      {{astronaut}}: <strong>{{mission}}</strong>
      <button
        (click)="confirm()"
        [disabled]="!announced || confirmed">
        Confirm
      </button>
    </p>
  `
})
export class AstronautComponent implements OnDestroy {
  @Input() astronaut: string;
  mission = '<no mission announced>';
  confirmed = false;
  announced = false;
  subscription: Subscription;

  constructor(private missionService: MissionService) {
    this.subscription = missionService.missionAnnounced$.subscribe(
      mission => {
        this.mission = mission;
        this.announced = true;
        this.confirmed = false;
    });
  }

  confirm() {
    this.confirmed = true;
    this.missionService.confirmMission(this.astronaut);
  }

  ngOnDestroy() {
    // prevent memory leak when component destroyed
    this.subscription.unsubscribe(); }}
```

3.5.1.- ¡¡Importante!!

En el ejemplo anterior, seguimos utilizando la lógica componente Padre-Hijo pues en un componente tenemos el selector del otro componente por lo que ambos se están renderizando y visualizando en la misma vista en el navegador.

Esto es importante entenderlo pues por ese motivo al usar el objeto Subject en el Service me va a mostrar la variable modificada. Para poder mostrar una variable en dos componentes que no se renderizan en la misma vista se debe de cambiar ese objeto por BehaviorSubject de la misma librería. El cambio sería tan solo de esa parte, todo lo demás sigue la misma lógica de uso.

El código a cambiar del ejemplo anterior sería:

```
private missionAnnouncedSource = new Subject<string>();
```

Por:

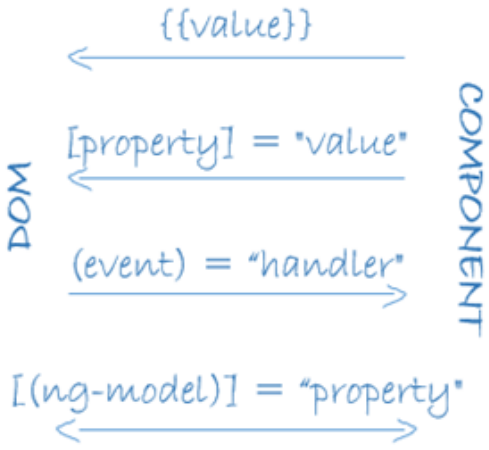
```
private missionAnnouncedSource = new BehaviorSubject(‘’);
```

4.- Conceptos avanzados de componentes

4.1.- Sintáxis binding

Data-Binding es una mecánica para coordinar que vé el usuario, especialmente con los valores de la aplicación. Angular posee diferentes tipos de data-binding, se podrían agrupar en tres categorías:

- *Source-to-view*. Desde el componente a la vista.
- *View-to-source*. Desde la vista al componente.
- *Two-way*. Doble binding.



Categoría	Tipo	Sintáxis
Source-to-view	Interpolacion Propiedades Atributos 1. Clases Estilos	{{expression}} [target]=”expression” bind-target=”expression”
View-to-source	Eventos	(target)=”statement” on-target=”statement”
Two-way		[(target)]=”expression” bindon-target=”expression”

Ejemplos de uso

Tipo	Target	Ejemplo
Propiedad	Elemento Componente Directiva	<img [src]="heroImageUrl"> <app-hero-detail [hero]="currentHero"></app-hero-detail> <div [ngClass]="{'special': isSpecial}">
Evento	Elemento Componente Directiva	<button (click)="onSave()">Save</button> <app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail> <div (myClick)="clicked=$event" clickable>click me
Two-way	Eventos Propiedades	<input [(ngModel)]="name">
Atributos	Atributos	<button [attr.aria-label]="help">help</button>
Clases	Clases	<div [class.special]="isSpecial">Special
Estilo	Estilo	<button [style.color]="isSpecial ? 'red' : 'green'">

Posible error a tener en cuenta

Cuando intentamos modificar una clase, atributo o estilo Angular nos puede bloquear ese trozo de código por seguridad. Para `saltarnos' esa seguridad y poder bindear información se debe de importar la clase *DomSanitizer* y utilizarla como en el siguiente ejemplo:

Elemento en el HTML

```
[style.background-color]="data.color"
```

Elemento en el componente

```
this.sanitization.bypassSecurityTrustStyle('#666');
```

De esta forma le estamos diciendo a Angular que el código que queremos inyectar no es peligroso.

4.2.- Two-way binding [(...)]

Two-way binding hace dos cosas:

- Colecciona o captura el valor de una variable.
- Permanece ‘escuchando’ por los cambios de esa variable.

Size.component.ts

```
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-sizer',
  templateUrl: './sizer.component.html',
  styleUrls: ['./sizer.component.css']
})
export class SizerComponent {


  @Input()  size: number | string;
  @Output() sizeChange = new EventEmitter<number>();

  dec() { this.resize(-1); }
  inc() { this.resize(+1); }

  resize(delta: number) {
    this.size = Math.min(40, Math.max(8, +this.size + delta));
    this.sizeChange.emit(this.size);
  }

}
```

app.component.html

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
<div [style.font-size.px]="fontSizePx">Resizable Text</div>
```

Mismo ejemplo con diferente opción:

```
<app-sizer [size]="fontSizePx" (sizeChange)="fontSizePx=$event"></app-sizer>
```

4.3.- NgClass y NgStyle

NgClass

Con NgClass podremos añadir o eliminar todas las clases para CSS que queramos y actualizar nuestro diseño sin problema con esta directiva.

```
<!-- toggle the "special" class on/off with a property -->
<div [ngClass]="isSpecial ? 'special' : """>This div is special</div>
```

Con varias clases a la vez:

```
currentClasses: {};
setCurrentClasses() {
  // CSS classes: added/removed per current state of component properties
  this.currentClasses =  {
    'saveable': this.canSave,
    'modified': !this.isUnchanged,
    'special':  this.isSpecial
  };
}
```

```
<div [ngClass]="currentClasses">This div is initially saveable, unchanged, and special.</div>
```

La directiva NgClass coge cada una de esas variables y si son true las coloca.

NgStyle

Al igual que NgClass, NgStyle añade o elimina estilos.

```
<div [style.font-size]="isSpecial ? 'x-large' : 'smaller'">
  This div is x-large or smaller.
</div>
```

Usando NgStyle:

```
currentStyles: {};
setCurrentStyles() {
  // CSS styles: set per current state of component properties
  this.currentStyles = {
    'font-style':  this.canSave      ? 'italic' : 'normal',
    'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
    'font-size':   this.isSpecial    ? '24px'   : '12px'
  };
}
```

```
<div [ngStyle]="currentStyles">
  This div is initially italic, normal weight, and extra large (24px).
</div>
```

Lo que hace NgStyle es implementar un valor u otro dependiendo del booleano.

4.4.- [(ngModel)]: Two-way binding

Esta directiva nos permite mostrar el valor de una variable o de un objeto y a la vez actualizar su contenido. Su uso más frecuente es en la etiqueta <input>

```
<label for="example-ngModel">[(ngModel)]:</label>
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

Es importante no olvidar importar la clase *FormsModule* en el módulo principal o submodulo donde estemos usando esta directiva para que no de fallo el navegador.

Un ejemplo de esta directiva pero sin usarla sería:

```
<label for="example-change">(ngModelChange)="...name=$event":</label>
<input [ngModel]="currentItem.name" (ngModelChange)="currentItem.name=$event"
id="example-change">
```

4.5.- NgSwitch

```
<div [ngSwitch]="currentItem.feature">
  <app-stout-item    *ngSwitchCase="'stout'"    [item]="currentItem"></app-stout-item>
  <app-device-item   *ngSwitchCase="'slim'"     [item]="currentItem"></app-device-item>
  <app-lost-item     *ngSwitchCase="'vintage'"  [item]="currentItem"></app-lost-item>
  <app-best-item     *ngSwitchCase="'bright'"   [item]="currentItem"></app-best-item>
<!-- . . . -->
  <app-unknown-item  *ngSwitchDefault           [item]="currentItem"></app-unknown-item>
</div>
```

Al igual que la condición *switch() case* en ciertos lenguajes mostraremos un elemento según el resultado de una variable. Esta directiva añade o elimina el componente según su valor como se puede ver en el ejemplo anterior.

Esta directiva trabaja tanto con componentes como elementos nativos.

```
<div *ngSwitchCase="'bright'"> Are you as bright as {{currentItem.name}}?</div>
```

4.6.- Variable #var para referenciar elementos en una plantilla

Usaremos el símbolo # para declarar una variable en una plantilla. Esta lógica funciona del mismo modo que declarar una variable en en un componente y después trabajar con ella. En el siguiente ejemplo podemos ver como se declara una variable y luego trabajamos con ella:

```
<input #phone placeholder="phone number" />

<!-- lots of other elements -->

<!-- phone refers to the input element; pass its `value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>
```

El ejemplo más usual para esta lógica es su uso en formularios:

```
<form #itemForm="ngForm" (ngSubmit)="onSubmit(itemForm)">
  <label for="name"
    >Name <input class="form-control" name="name" ngModel required />
  </label>
  <button type="submit">Submit</button>
</form>

<div [hidden]="!itemForm.form.valid">
  <p>{{ submitMessage }}</p>
</div>
```

4.7.- Operaciones especiales

4.7.1.- Pipes (|)

El valor de cualquier variable a veces necesita ser transformada para mostrarla en la plantilla, por ejemplo, un texto en mayúsculas o filtrar una lista. Para casos como estos y más tenemos los *Pipes*, son simples funciones que aceptan un valor y lo devuelve transformado.

```
<p>Title through uppercase pipe: {{title | uppercase}}</p>
```

Podemos también aplicar varios filtros:

```
<!-- convert title to uppercase, then to lowercase -->
<p>Title through a pipe chain: {{title | uppercase | lowercase}}</p>
```

Aplicar parámetros:

```
<!-- pipe with configuration argument => "February 25, 1980" -->
<p>Manufacture date with date format pipe: {{item.manufactureDate | date:'longDate'}}</p>
```

O mostrar la información de un Json:

```
<p>Item json pipe: {{item | json}}</p>
```

Para saber más acerca de los *Pipes* pincha aquí.

4.7.2.- El operador (?)

```
<p>The item name is: {{item?.name}}</p>
```

Esta opción se utiliza para variables que son usadas en la plantilla pero que esperan respuesta de un servidor para ser inicializadas por lo que durante un breve momento son *null* o *undefined* y dan un error en el navegador. Dicho error no es grave y la aplicación continua su ejecución pero en algunos casos sí que hace que falle la plantilla y no se muestre la información.

Con este operador hacemos que la plantilla se renderice y una vez que se devuelva el valor del servidor mostrarlo.

4.8.- Tipos de eventos

Algunos de los eventos más usados en Angular son:

click	blur	change
focus	keypress	submit

En la siguiente web se pueden ver una lista completa de todos los eventos:

https://www.w3schools.com/jsref/dom_obj_event.asp

5.- Conceptos avanzados de Angular

5.1.- Módulos

5.1.1.- NgModule API

```
@NgModule({
  // Static, that is compiler configuration
  declarations: [], // Configure the selectors
  entryComponents: [], // Generate the host factory

  // Runtime, or injector configuration
  providers: [], // Runtime injector configuration

  // Composability / Grouping
  imports: [], // composing NgModules together
  exports: [] // making NgModules available to other parts of the app
})
```

Propiedad	Descripción
declarations	Clases declarables (componentes, directivas y *pipes*). Todas estas clases se declaran en un solo módulo, según necesitemos, si no dará fallo el compilador.
providers	Clases inyectables (*Services* y algunas clases). Angular tiene asociado a cada módulo un injector, por lo que el módulo raíz tiene su injector raíz y cada submódulo tiene el suyo propio que parte del principal.
imports	Todos aquellos módulos o librerías de Angular que se vayan a utilizar en la aplicación.
exports	Clases declarables (componentes, directivas y *pipes*) que quieran ser importadas por otro módulo
bootstrap	Componente de arranque de la aplicación
entryComponents	Componentes que pueden ser cargados dinámicamente en la vista

5.1.2.- Multimodulado o submódulos

Angular al ser un *framework* que trabaja de forma modular, tiene la posibilidad de dividir sus aplicaciones en diferentes áreas, por ejemplo por su funcionalidad, y así aligerar la primera carga de la web.

Esta división se hace a través de submódulos. En cada uno de estos módulos cargaremos todo lo necesario para esa área. Con esta forma de trabajar todos estos ficheros no se cargarán hasta que no accedemos a esa área de trabajo.

Esta forma de trabajo también es bastante útil para organizar nuestro proyecto si alcanza un tamaño considerable.

Para crear un nuevo módulo usaremos el siguiente comando:

```
ng generate module nameModule
```

Esto nos generará el siguiente código:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class CustomerDashboardModule { }
```

Como se puede ver es exactamente igual que el módulo principal de una aplicación en Angular. Entonces ¿cómo va a saber Angular cual es el módulo principal? Para eso es el archivo *main.ts* que se explica al principio del documento y que le 'dice' a Angular cual es el módulo principal a cargar.

```
platformBrowserDynamic().bootstrapModule(AppModule).catch(err => console.error(err));
```

Una vez creado el módulo podemos añadirle cualquier elemento (componente, directiva, services, etc) usando el Angular CLI.

Si hemos creado un módulo cuyo archivo se llama *customer-dashboard.module.ts* entonces el comando a introducir será:

```
ng generate component customerDashboard
```

Al crear el componente Angular irá buscando desde la ruta donde nos encontremos hacia la raíz del proyecto un módulo al que importar el componente hasta llegar al módulo principal e importarlo ahí

Una vez creado el nuevo módulo tan solo queda importalo en el módulo principal y añadirlo a *imports* para poder usarlo en la aplicación.

¿Cómo visualizar estos nuevos componentes de submódulos?

Existen dos formas de mostrar los componentes añadidos a submódulos:

- Añadiendo el componente a los *exports* del submódulo y así poder utilizar su selector en otro componente.
- Añadiendo rutas al submódulo y sus componentes.

El primer método tan solo se necesita añadir el componente del submódulo al array de *exports* de este mismo submódulo. De esta forma podemos usar su selector para cargar ese componente en otro lado de la aplicación, como por ejemplo en el AppComponent.

Para el segundo caso debemos seguir estos pasos:

1. Añadir la ruta al *app-routing*.

```
{ path: 'customer', loadChildren: '.. rutadelarchivo/customer-dashboard.module#CustomerDashboarModule' }
```

2. Crear en el *imports* del submódulo las subrutas para 'llegar' hasta el componente.

```
…
imports: [
  …
  RouterModule.forChild([
    { path: "", component: CustomerDashboardComponent }
  ]),
  …
]
...
```

5.1.3.- Providers

Providers es una instrucción del Inyector de Dependencias para obtener el valor de una dependencia. Esto quiere decir que todos los archivos que Angular detecte que se almacenan en el ID deben de ser importados en el *providers* para poder ser utilizados. La mayoría de estos archivos suelen ser *services*.

Dependiendo del nivel donde importemos ese archivo en el *providers* correspondiente podremos usarlo en toda o solo parte de la aplicación:

- Toda la aplicación:
    - El archivo se importará en el *providers* del módulo principal de nuestra aplicación.

    - @Injectable({ providedIn: ‘root’ }) Cuando un archivo, normalmente un *service,* tiene este metadata significará que Angular cargará ese archivo en el providers principal sin necesidad de tener que hacer el punto anterior.
- Para un solo submódulo cuando tenemos multimodulado:
    - En el *providers* de este submódulo.
    - @Injectable({ providedIn: nombreModulo }) donde nombreModulo será la clase del submódulo.

```
import { Injectable } from '@angular/core';
import { UserModule } from './user.module';

@Injectable({
  providedIn: UserModule,
})
export class UserService {
}
```

- Para un solo componente:
    - En el *providers* de ese componente.

5.2.- *Routing*

Hasta ahora se había visto como crear rutas sencillas entre componentes. Pero se pueden hacer varios niveles de enrutado para una aplicación web.

```
const crisisCenterRoutes: Routes = [
  {
    path: 'crisis-center', component: CrisisCenterComponent, children: [
      {
        path: '', component: CrisisListComponent,  children: [
          {
            path: ':id',  component: CrisisDetailComponent
          },
          {
            path: '', component: CrisisCenterHomeComponent
          }
        ]
      }
    ]
  }
];
```

Pero no basta con esto, no se debe olvidar colocar la etiqueta *router-outlet* en cada componente que tenga rutas ‘hijo’ para que Angular las reconozca y sepa donde renderizarlas.

CanActivate

Se utiliza para limitar a los usuarios el acceso a las rutas que nosotros no queramos que accedan, por ejemplo, una zona de administración de una web.

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  canActivate( next: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    // Código para comprobar el token o clave para devolver true o false. Lo habitual justo antes de devolver
false se reenvie con la clase router de Angular a otra vista, si no la app se quedará parada en este punto.
  }
}
```

Una vez creado el código anterior se debe de implementar en el array de rutas de la aplicación.

```
{ path: 'categorias', canActivate: [AuthGuard], component: 'CategoriasComponent' }
```

Eventos

Durante la navegación, el objeto *Router* emite una serie de eventos a través de la propiedad *Router.events*.

```
this.router.events.subcribe( (event) => {
  console.log(event);
});
```

Algunos de esos eventos son:

NavigationStart	NavigationEnd	NavigationError
GuardsCheckEnd	ChildActivationStart	ResolveStart

Para saber la lista entera pulse aquí.

6.- Inyector de dependencias

El inyector de dependencias (ID) es un patrón de diseño de Angular que generalmente se usa en el diseño de una aplicación para aumentar su eficiencia y modularidad.

Las dependencias son servicios u objetos que una clase necesita para realizar su función. ID es un patrón de codificación en el que una clase solicita dependencias de fuentes externas en lugar de crearlas por sí mismas.

En Angular, el marco de ID proporciona dependencias declaradas a una clase cuando se instancia esa clase.

Servicios que necesitan de otros servicios

Aunque no es habitual un *service* puede tener sus propias dependencias de otro *service* del proyecto.

```
import { Injectable } from '@angular/core';
import { HEROES }     from './mock-heroes';
import { Logger }     from '../logger.service';

@Injectable({
  providedIn: 'root',
})
export class HeroService {

  constructor(private logger: Logger) {  }

  getHeroes() {
    this.logger.log('Getting heroes ...');
    return HEROES;
  }
}
```

Dependencias opcionales

Cuando un componente o servicio declara una dependencia, el constructor de la clase toma esa dependencia como parámetro pero podemos decirle a Angular que la dependencia es opcional anotando el parámetro *@Optional()* en el constructor.

```
constructor(@Optional() private logger: Logger) { }
```

7.- HttpClient

7.1.- Solicitud de respuesta

Se puede declarar el tipo de objeto para la respuesta del servidor, para que el consumo de la salida sea más fácil y más obvio. Para especificar el tipo de objeto de respuesta, primero se debe de definir una interfaz con las propiedades requeridas.

```
export interface Config {
  heroesUrl: string;
  textfile: string;
}
```

Es importante recordar que debe ser una *interface* y no una clase.
En la petición http:

```
getConfig() {
  // now returns an Observable of Config
  return this.http.get<Config>(this.configUrl);
}
```

Y en la devolución de la llamada en el componente:

```
config: Config;

showConfig() {
  this.configService.getConfig()
    // clone the data object, using its known Config shape
    .subscribe((data: Config) => this.config = { ...data });
}
```

7.2.- Leer la respuesta completa

A veces necesitamos leer la respuesta completa del servidor, para ello primero en la petición http:

```
getConfigResponse(): Observable<HttpResponse<Config>> {
  return this.http.get<Config>(
    this.configUrl, { observe: 'response' });
}
```

En la devolución de la llamada:

```
showConfigResponse() {
  this.configService.getConfigResponse()
    // resp is of type `HttpResponse<Config>`
    .subscribe(resp => {
      // display its headers
      const keys = resp.headers.keys();
      this.headers = keys.map(key =>
        `${key}: ${resp.headers.get(key)}`);

      // access the body directly, which is typed as `Config`.
      this.config = { ... resp.body };
    });
}
```

7.3.- Peticiones de datos no JSON

No todas las peticiones a las APIs son para obtener datos JSON. Ejemplo de una petición para descargar un archivo de texto:

```
getTextFile(filename: string) {
  // The Observable returned by get() is of type Observable<string>
  // because a text response was specified.
  // There's no need to pass a <string> type parameter to get().
  return this.http.get(filename, {responseType: 'text'})
    .pipe(
      tap( // Log the result or error
        data => this.log(filename, data),
        error => this.logError(filename, error)
      )
    );
}
```

8.- Formularios

En desarrollo web las aplicaciones usan formularios para loguear usuarios, actualizar un perfil, introducir información sensible y actualizar muchos otros datos.

Angular provee dos formas diferentes de crear formularios: *reactive* y *template-driven*. Ambos recogen los eventos de las etiquetas *input* desde la vista, validan los datos introducidos, crean un modelo de formulario y proveen una manera de recoger posibles cambios.

En general:

- *Reactive forms*: son más robustos, son más escalables, reusables y testeables. Si la aplicación se base mayoritariamente en formularios, una zona de administración por ejemplo, se debe usar esta forma de generar formularios.
- *Template-driven forms*: son más útiles para añadir simples formularios.

Diferencias claves

	Reactive	***Template-driven***
Preparación	Se crea en el componente	Creado por directivas
Modelo	Estructural	No estructural
Previsibilidad	Síncrona	Asíncrona
Validación	Funciones	Directivas
Mutabilidad	Inmutable	Mutable
Escalabilidad	Acceso a API a bajo nivel	Abstracción de la API

8.1.- *Reactive Forms*

Los formularios *reactive* utilizan un enfoque explícito e inmutable para administrar el estado de un formulario. Cada cambio en el estado del formulario devuelve un nuevo estado, que mantiene la integridad del modelo entre cambios. También se debe de tener en cuenta a la hora de trabajar con ellos que se crean bajo la lógica de *Observable* de Angular, proporcionan por lo tanto un acceso síncrono al modelo de datos, con operaciones del objeto *Observable* y seguimiento de cambios.

Registro del formulario

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule { }
```

Debemos importar el módulo de *ReactiveFormsModule* en el módulo principal o submódulo donde vayamos a trabajar con formularios.

Primer formulario

```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'app-name-editor',
  templateUrl: './name-editor.component.html',
  styleUrls: ['./name-editor.component.css']
})
export class NameEditorComponent {
  name = new FormControl('');
}
```

FormControl es la clase más básica para realizar un formulario *reactive*. Esto creará una instancia para guardar el valor de *name* como una propiedad de clase. Se debe de inicializar la variable a través del constructor de *FormControl(‘’)*.

En el html debemos colocar la propiedad bindeada a la variable:

```
<label>  Name:
  <input type="text" [formControl]="name">
</label>
```

Para mostrar el valor de *name* fuera de la etiqueta de input:

- Plantilla:

```
<p>
  Value: {{ name.value }}
</p>
```

- Componente:

```
this.name.valueChanges.subcribe( (value) => {
  console.log(value);
});

//O también

this.name.value
```

8.1.1.- FormGroup

Para crear un objeto con varios *FormControl* utilizaremos el objeto *FormGroup*.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
  });
}
```

En la plantilla:

```
<form [formGroup]="profileForm">

  <label>
    First Name:
    <input type="text" formControlName="firstName">
  </label>

  <label>
    Last Name:
    <input type="text" formControlName="lastName">
  </label>

</form>
```

Crearemos un binding de la propiedad de *formGroup* con el objeto creado en el componente. Esta instancia es suficiente para que Angular reconozca en cada propiedad de *formControlName* el bindeo con las variables del objeto *profileForm*.

Para recuperar los valores del formulario se hace igual que para el caso anterior:

```
console.warn(this.profileForm.value);
```

8.1.2.- Grupos anidados

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl('')
    })
  });
}
```

Con este ejemplo podemos ver lo sencillo que es crear subgrupos dentro del *FormGroup* principal. Y para implementarlo en el html debemos colocar la etiqueta *formGroupName* para referenciar el objeto:

```
<div formGroupName="address">
  <h3>Address</h3>
  <label> Street:
    <input type="text" formControlName="street">
  </label>
  <label> City:
    <input type="text" formControlName="city">
  </label>
  <label> State:
    <input type="text" formControlName="state">
  </label>
  <label> Zip Code:
    <input type="text" formControlName="zip">
  </label>
</div>
```

8.1.3.- FormBuilder

FormBuilder es una forma diferente de realizar también un formulario *reactive*. Tan solo se debe de cambiar la forma de crear una instancia del objeto a usar en el html.

```
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';

@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
  profileForm = this.fb.group({
    firstName: [''],
    lastName: [''],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });

  constructor(private fb: FormBuilder) { }
}
```

Para más información de este objeto visitar la web de Angular.

8.2.- *Template-driven forms*

Registro del formulario

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    // other imports ...
    FormsModule
  ],
})
export class AppModule { }
```

Debemos importar el módulo de *FormsModule* en el módulo principal o submódulo donde vayamos a trabajar con formularios.

Plantilla HTML

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm">
    <div class="form-group">
      <label for="name">Name</label>
      <input type="text" class="form-control" id="name" required [(ngModel)]="model.name" name="name"
            #name="ngModel">
      <div [hidden]="name.valid || name.pristine" class="alert alert-danger"> Name is required </div>
    </div>

    <div class="form-group">
      <label for="alterEgo">Alter Ego</label>
     <input type="text" class="form-control" id="alterEgo" [(ngModel)]="model.alterEgo"
           name="alterEgo">
    </div>

    <div class="form-group">
      <label for="power">Hero Power</label>
      <select class="form-control" id="power" required  [(ngModel)]="model.power" name="power"
            #power="ngModel">
        <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
      </select>
      <div [hidden]="power.valid || power.pristine" class="alert alert-danger"> Power is required </div>
    </div>

    <button type="submit" class="btn btn-success" [disabled]="!heroForm.form.valid">Submit</button>
    <button type="button" class="btn btn-default" (click)="newHero(); heroForm.reset()">
          New Hero</button>
</form>
```

Como se puede ver un formulario *Template-driven* es una combinación de la arquitectura básica de Angular con alguna excepción, como:

- #heroForm=”ngForm” => ngForm añade las funciones de formularios a nuestra variable heroForm dentro de nuestra etiqueta <form>
- (ngSubmit) => Evento para ejecutar la función de enviar los datos tras pulsar el boton con *type=”submit”*
- #var=”ngModel” => Al igual que la variable del formulario, añadiremos las funcionalidades de formulario a cada variable.

Componente

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

import { Hero }    from '../hero';

@Component({
  selector: 'app-hero-form',
  templateUrl: './hero-form.component.html',
  styleUrls: ['./hero-form.component.css']
})
```

```
export class HeroFormComponent {

  powers = ['Really Smart', 'Super Flexible',
          'Super Hot', 'Weather Changer'];

  model = new Hero();

  submitted = false;

  onSubmit() { this.submitted = true; }

  newHero() {
    this.model = new Hero(42, '', '');
  }
}
```

8.3.- Validación

8.3.1.- Template-driven validation

Para añadir validación a este tipo de formularios, se añadirá la misma forma de validación que cualquier formulario de HTML. Angular usa directivas para unir esos atributos con sus funciones de validación:

- Los atributos *required*, *minlenght* o *forbiddenName* en la etiqueta <input>
- *#var=”ngModel”* exporta la clase *NgModel* en una variable. Se usa para validar los estados de la etiqueta donde se coloque.

```
<input id="name" name="name" class="form-control" required minlength="4" appForbiddenName="bob"
    [(ngModel)]="hero.name" #name="ngModel" >

<div *ngIf="name.invalid && (name.dirty || name.touched)" class="alert alert-danger">

  <div *ngIf="name.errors.required"> Name is required. </div>
  <div *ngIf="name.errors.minlength"> Name must be at least 4 characters long.  </div>
  <div *ngIf="name.errors.forbiddenName"> Name cannot be Bob. </div>

</div>
```

8.3.2.- Reactive form validation

Para formulario *reactive* la validación se puede implementar también en el HTML pero *FormControl* tiene una opción para pasarle los parámetros de la validación:

```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    'name': new FormControl(this.hero.name, [ Validators.required, Validators.minLength(4),
      forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom validator.
    ]),
    'alterEgo': new FormControl(this.hero.alterEgo),
    'power': new FormControl(this.hero.power, Validators.required)
  });

}

get name() { return this.heroForm.get('name'); }

get power() { return this.heroForm.get('power'); }
```

El parámetro de validación puede ser un solo valor o un array de valores de validación. La lógica del HTML será parecida a la validación de *Template form*:

```
<input id="name" class="form-control" formControlName="name" >

<div *ngIf="heroForm.get('name').invalid && (heroForm.get('name').dirty ||
heroForm.get('name').touched)" class="alert alert-danger">

  <div *ngIf="heroForm.get('name').errors.required"> Name is required. </div>
  <div *ngIf="heroForm.get('name').errors.minlength"> Name must be at least 4 characters long. </div>
  <div *ngIf="heroForm.get('name').errors.forbiddenName"> Name cannot be Bob. </div>
</div>
```

8.3.3.- Clases CSS

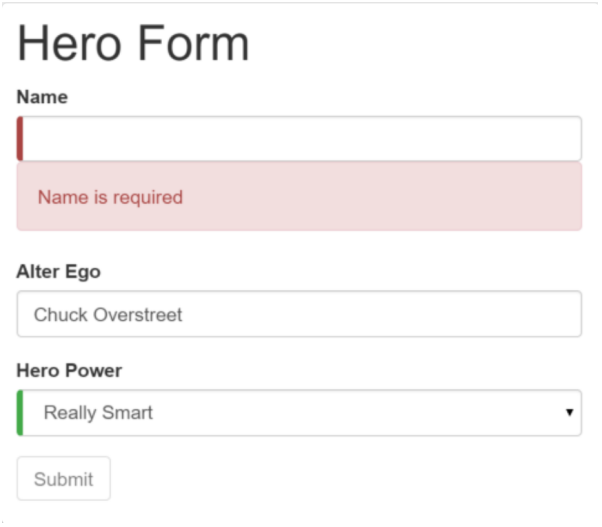
Angular provee una serie de clases que se añaden automáticamente a los elementos de control, con ellas se puede editar el estilo del elemento según su estado de validación.

.ng-valid	.ng-invalid	.ng-pending	.ng-dirty
.ng-pristine	.ng-untouched	.ng-touched	

Ejemplo:

```
.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}
```

```
.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /* red */
}
```



Estas clases son añadidas automáticamente en el HTML con los atributos de HTML por lo que si se quiere modificar el CSS a través de estas clases se deben colocar los atributos, como por ejemplo *'required'*, aunque no hagan falta como en el caso de *Reactive Forms* en realidad, pero si queremos modificar el estilo como en el ejemplo anterior se deben de colocar. Esto no afectará a la funcionalidad del formulario en el caso de ser *Reactive Form*.

9.- *Material Design*

9.1.- ¿Qué es *Material Design*?

Material es un lenguaje de diseño que define un conjunto de pautas que muestran cómo diseñar mejor un sitio web. Le indica qué botones se debe usar y cuáles, cómo animarlos o moverlos, así como dónde y cómo deben colocarse, etc.

Es un lenguaje diseñado principalmente para dispositivos móviles, pero que se puede usar en otras plataformas y aplicaciones.

Además de su facilidad de implementación y uso, otra gran característica de esta plataforma de diseño es la falta de dependencias de modelos y librerías de JavaScript.

9.2.- Diferencias entre *Material* y *Bootstrap*

Propósito

Si bien ambos se utilizan para el desarrollo web, *Material* está más orientado hacia la apariencia de un sitio web o aplicación. Esto es evidente en la forma en que se usa, sus pautas de diseño y las numerosas plantillas y componentes con los que viene, que se centran en el diseño.

Bootstrap, por otro lado, se centra principalmente en crear fácilmente sitios webs *responsive* y aplicaciones web, que sean funcionales y de alta calidad en lo que respecta a la experiencia del usuario.

Proceso de diseño

Material viene con numerosos componentes que proporcionan un diseño base, que luego puede ser modificados por los propios desarrolladores.

Bootstrap es más una biblioteca UI. Presenta una serie de componentes, como su sistema de cuadrícula avanzado.

En cuanto a componentes de terceros, hay varios compatibles con *Bootstrap*, mientras que con *Material* no hay ninguno.

De los dos, *Material* tiene mucho más atractivo en cuanto a apariencia por sus llamativos colores y sus animaciones, mientras que *Bootstrap* tiene un diseño más estándar.

Compatibilidad del navegador y marcos de trabajo

Ambos son compatibles con todos los navegadores de hoy día.

En cuanto al framework, *Material* es compatible con *material angular*, *React Material* y utiliza el preprocesador de SASS. *Bootstrap* admite marcos de *React Bootstrap*, *Angular UI Bootstrap* y puede usar los idiomas de SASS y LESS.

Documentación y soporte

Bootstrap tiene una documentación y una comunidad bastante más amplia que *Material* debido sobre todo a que este segundo es más nuevo pero para ambos podemos encontrar muchas soluciones a problemas y muchas opciones para una web.

Elección

Lo más adecuado sería usar *Bootstrap* para sitios web *responsive* y más profesionales. *Material* es ideal para crear sitios webs centrados en la apariencia con gran detalle.

9.3.- Instalación

Para instalar *Material Design* debemos hacerlo a través de Angular CLI, para ello introducimos el siguiente comando:

```
ng add @angular/material
```

Este comando instalará el CDK (*Component Deb Kit*), las librerías de animación de Angular y mientras se instalan todas las demás librerías nos preguntará que deseamos instalar además:

- Tema de *Material*. Nos pedirá que elijamos un estilo de diseño o una plantilla en blanco.
- HammerJS. Nos preguntará si instalar o no esta librería. HammerJS se usa para poder reconocer y usar algunos gestos en la pantalla.
- Importar *BrowserAnimationsModule*.

Además de instalar todas estas librerías, modificará los siguientes archivos:

- Añadir dependencias a *package.json*.
- Añadir la fuente *Roboto* y los iconos de *Material* al index.html
- Añadir código CSS al style.css

9.4.- Cómo usar *Material*

Para usar material se seguirán estos pasos normalmente:

1. Importar la clase que queremos usar en el módulo.

```
import { MatSliderModule } from '@angular/material/slider';
…
@NgModule ({....
  imports: [...,
  MatSliderModule,
…]
})
```

2. Añadir la etiqueta al html.

```
<mat-slider min="1" max="100" step="1" value="1"></mat-slider>
```

Para más componentes e información visitar la web de material.angular.io.

10.- Testing en Angular

Realizar test en proyectos cuando superar cierto tamaño es muy importante para probar la aplicación sin tener que hacerlo manualmente. Además si se combina con la integración continua se puede minimizar el riesgo de futuros bugs

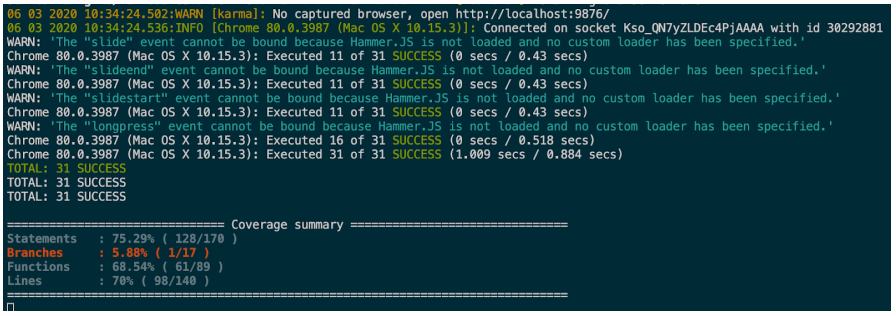
Tipos de test que existen:

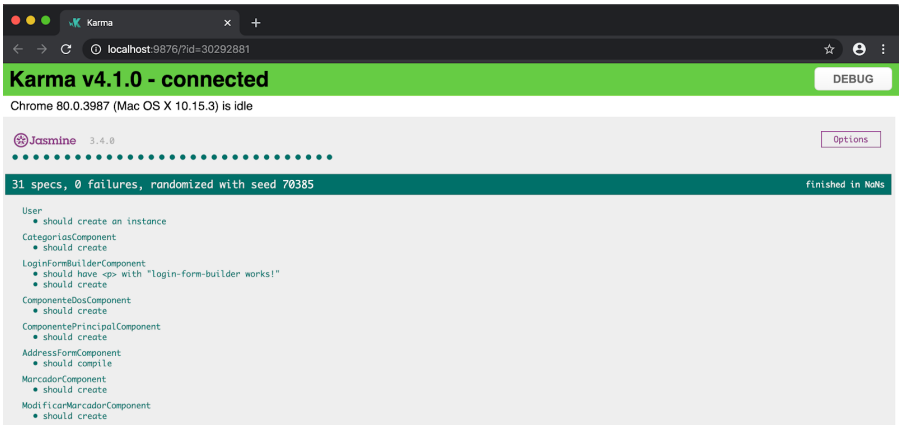
- **Tests Unitarios**: Consiste en probar unidades pequeñas (componentes por ejemplo).
- **Tests End to End (E2E)**: Consiste en probar toda la aplicación simulando la acción de un usuario, es decir, por ejemplo para desarrollo web, mediante herramientas automáticas, se abrirá el navegador y navegará y usará la página como lo haría un usuario normal.
- **Tests de Integración**: Consiste en probar el conjunto de la aplicación asegurando la correcta comunicación entre los distintos elementos de la aplicación. Por ejemplo, en Angular observando cómo se comunican los servicios con la API y con los componentes.

Angular CLI descarga e instala todo lo necesario para testear la aplicación con el framework de *Jasmine*. Para realizar un test de nuestra aplicación:

ng test

Al introducir este comando Angular realizará un testeo de toda la aplicación e irá recorriendo los diferentes *.spec.ts* de cada clase.





Configuración

Angular CLI configura *Jasmine* y *Karma* para su correcto funcionamiento. Los archivos de configuración de tests son *karma.conf.js* y *test.ts* que se encuentran en la raíz del proyecto.

Integración continua

Los servicios de integración continua nos ayudan a mantener nuestro proyecto libre de bugs. Enlazando el repositorio del proyecto con alguna de las herramientas de integración continua se realizarán los test cada vez que se haga un *commit* y un *pull request*.

Las herramientas recomendadas por Angular son Circle CI, Travis CI o Jenkins

Informe de cobertura

Angular CLI puede ejecutar pruebas unitarias y crear informes de cobertura. Estos informes muestran cualquier parte de nuestro código que no pueda ser probada adecuadamente por las pruebas unitarias. Para generar el informe:

```
ng test --code-coverage
```

Este comando creará una carpeta llamada *coverage* en la raíz del proyecto. Al abrir el *index.html* que está dentro de esta carpeta se podrán ver los resultados. Si queremos que se genere automáticamente un informe en cada test debemos de activar esta opción en el *angular.json*:

```
"test": {
  "options": {
    "codeCoverage": true
  }
}
```

Para más información de testing en Angular ver su web.

10.1.- *Jasmine*

Para hacer los test en Angular se suele usar Jasmine. Jasmine es un framework Javascript, para la definición de test usando un lenguaje natural entendible por todo tipo de personas.

Un ejemplo de test sería:

```
describe("A suite name", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

- **describe**: Define una colección de tests. Ésta función recibe dos parámetros, un string con el nombre de la colección y una función donde definiremos los tests.
- **it**: Define un test. Recibe como parámetro el nombre del test y una función a ejecutar por el test.
- **expect**: Lo que espera recibir el test. Es decir, con expect hacemos la comprobación del test. Si la comprobación no es cierta el test falla. En el ejemplo anterior se comprueba si *true* es *true*, por lo que el test pasa.

También podemos ejecutar funciones antes de realizar un test, o después:

- **beforeAll**: Se ejecuta antes de pasar todos los tests de una colección.
- **afterAll**: Se ejecuta después de pasar todos los tests de una suite.
- **beforeEach**: Se ejecuta antes de cada test de una colección.
- **afterEach**: Se ejecuta después de cada test de una colección.

```
describe('Hello world', () => {

  let expected = "";

  beforeEach(() => {
    expected = "Hello World";
  });

  afterEach(() => {
    expected = "";
  });

  it('says hello', () => {
    expect(helloWorld())
        .toEqual(expected);
  });
});
```

Para conocer mejor todo el lenguaje de programación de *Jasmine* visitar su web.

Cuando se crea un proyecto con Angular CLI nos creará un archivo *.spec.ts*, y eso es porque Angular CLI se encarga por nosotros de generar un archivo para testear cada componente. Además se genera el código mínimo necesario para empezar a probar y testear los componentes.

Un ejemplo de test de componente básico:

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { NotesComponent } from './notes.component';

describe('NotesComponent', () => {
  let component: NotesComponent;
  // Esta variable nos añadirá más información para que sea más fácil el testeo.
  let fixture: ComponentFixture<NotesComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ NotesComponent ]
       // Si tuviéramos algún servicio o dependencia debemos colocarla aquí también como si de un módulo se
tratara.
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(NotesComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Componente con dependencias de un servicio

```
 import {LoginComponent} from './login.component';
  import {AuthService} from "./auth.service";

  describe('Login component', () => {

    let component: LoginComponent;
    let service: AuthService;

    beforeEach(() => {
      service = new AuthService();
      component = new LoginComponent(service);
    });

    afterEach(() => {
      localStorage.removeItem('token');
```

```
    service = null;
    component = null;
  });


  it('canLogin returns true when the user is authenticated', () => {
    localStorage.setItem('token', '12345');
    expect(component.isLogged()).toBeTruthy();
  });


});
```

En este ejemplo se puede ver una forma de inyectar el servicio diferente al ejemplo anterior. Ambas son correctas.

Spy de Jasmine

Jasmine ofrece también la posibilidad de coger una clase y devolver directamente lo que nos interese sin tener que ejecutar internamente sus métodos:

```
import {LoginComponent} from './login.component';
import {AuthService} from "./auth.service";

describe('Component: Login', () => {

  let component: LoginComponent;
  let service: AuthService;
  let spy: any;

  beforeEach(() => {
    service = new AuthService();
    component = new LoginComponent(service);
  });

  afterEach(() => {
    service = null;
    component = null;
  });


  it('canLogin returns true when the user is authenticated', () => {
    spy = spyOn(service, 'isAuthenticated').and.returnValue(true);
    expect(component.isLogged()).toBeTruthy();
  });
});
```

Con la función *spyOn* se puede hacer que el servicio devuelva directamente *true* en la llamada al nombre de función que le pasemos como parámetro al spy.

Testeando llamadas asíncronas

Si por ejemplo tenemos un test que testa un método asíncrono del componente o del servicio (una llamada a una API por ejemplo):

```
it('Should get the data', fakeAsync(() => {
  fixture.componentInstance.getData();
  tick();
  fixture.detectChanges();
  expect(component.data).toEqual('new data');
}));
```

Angular proporciona el método *fakeAsync* para realizar llamadas asíncronas, dejándonos acceso a la llamada a *tick()* el cual simula el paso del tiempo para esperar a que la llamada asíncrona se realice.

Accediendo a la vista

Para acceder a los elementos html de la vista de un componente, podemos usar su *fixture*:

```
describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;
  let submitButton: DebugElement;


  beforeEach(() => {

    TestBed.configureTestingModule({
      declarations: [LoginComponent]
    });

    // create component and test fixture
    fixture = TestBed.createComponent(LoginComponent);

    // get test component from the fixture
    component = fixture.componentInstance;

    submitButton = fixture.debugElement.query(By.css('button_submit'));

  });
});
```

11.- Angular 9, pequeños cambios con Angular 8

La última versión de Angular trae cambios eficientes y adiciones al framework. Esto significa que el estilo del código no tiene nada que cambiar, pero la versión actualizada ofrece correcciones de errores. El lanzamiento de la actualización de Angular 9 que abarcan toda la plataforma, incluyendo el framework, el material angular y la CLI.

Angular 9 es una versión importante porque contiene un nuevo procesador para Angular, llamado "Ivy" . Ivy hace que las aplicaciones sean más pequeñas y rápidas. Las características más importantes aparte de *Ivy* serían:

- Las aplicaciones CLI angulares se compilan en modo AOT de forma predeterminada tanto para la versión developer como producción.
- La última versión incluye comprobación de tipo de plantilla más estrictas que hasta ahora.
- No hay soporte para Typecript 3.1 y 3.5. Tienes que actualizar a Typecript 3.7

Pero para saber bien como actualizar de Angular 8 a Angular 9 debemos de ir a la web oficial.

12.- Bibliografía

- Documentación oficial de Angular 8. Recuperado de https://v8.angular.io/docs
- Wikipedia. Angular (framework). Recuperado de https://es.wikipedia.org/wiki/Angular_(framework)
- Wikipedia. Single page application. Recuperado de https://es.wikipedia.org/wiki/Single-page_application
- Documentación oficial de Typescript. Recuperado de https://www.typescriptlang.org/index.html
- Wikipedia. TypeScript. Recuperado de https://es.wikipedia.org/wiki/TypeScript
- W3schools. HTML DOM Events. Recuperado de https://www.w3schools.com/jsref/dom_obj_event.asp
- Sharing data between Angular components - Four methods (20 Abr 2017) by Jeff Delaney. Fireship. Recuperado de https://fireship.io/lessons/sharing-data-between-angular-components-four-methods/
- Best Practices for Writing Angular Apps | Angular Guidelines (31 Ene 2020). Jsmount. Recuperado de https://www.jsmount.com/best-practices-for-writing-angular-apps/
- Material Design vs Bootstrap: Which One is Better? (6 May 2019) by Anli. Azmind. Recuperado de https://azmind.com/material-design-vs-bootstrap/
- Ractive FormGroup validation with AbstractControl in Angular 2 (26 Oct 2016) by Todd Motto. Ultimatecourses. Recuperado de https://ultimatecourses.com/blog/reactive-formgroup-validation-angular-2
- Documentación oficial de Angular Material 8. Recuperado de https://v8.material.angular.io
- Sitio web: https://jsonplaceholder.typicode.com/
- Angular - Cómo hacer testing unitario con Jasmine (14 Dic 2018). Codingpotions. Recuperado de https://codingpotions.com/angular-testing
- What's new in Angular 9? Top New Features and Ivy (10 Feb 2020) by Manigandan. Agiratech. Recuperado de https://www.agiratech.com/top-new-features-angular-9/
- Angular 9: What's new? (12 Feb 2020) by John Papa and Kapehe Jorgenson. Auth0. Recuperado de https://auth0.com/blog/angular-9-whats-new/