





Introducción a Angular



Hola!



Soy José Julián Ariza

Fundador de **Impacto Tecnológico** desde 2006.

Líder estratégico de Equipos de Desarrollo Ágil en modalidad Online y Presencial

Arquitecto de Software por naturaleza y DevOps por hobby

@JJArizaV – josejulian@impactotecnologico.net

@impactotecno



Unidad 1

Sentando las bases

Herramientas, Typescript, etc..



Instalación de NodeJS y Angular



Instalaciones

- ◆ Node: <https://nodejs.org/en/>
- ◆ Visual Studio Code: <https://code.visualstudio.com>

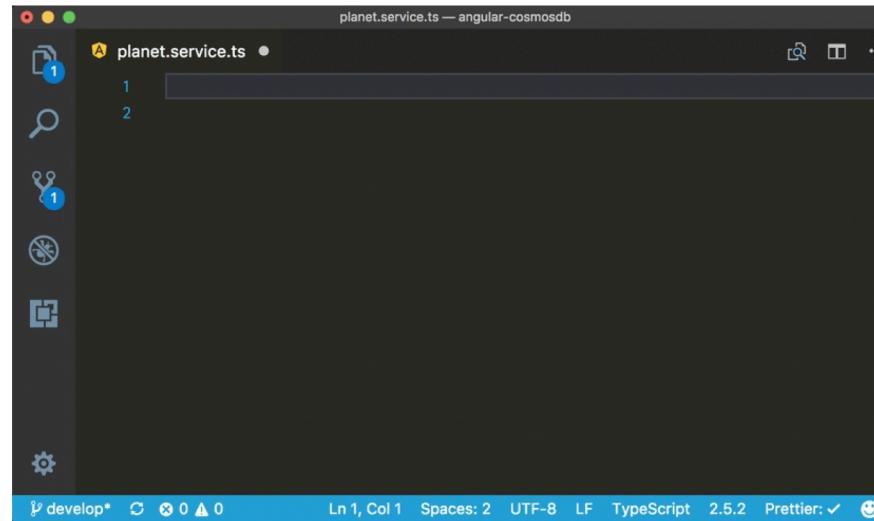


Visual Studio Plugins

- ◆ Auto Import: nos quita de lo más tedioso de trabajar con TypeScript que es hacer los imports necesarios.
- ◆ Angular Language Service: nos permite el autocompletado en los ficheros en los templates de los componentes y marca error cuando interpolamos una variable que no está definida como atributo.
- ◆ TSLint: en el propio editor nos marca los errores de estilo y como se integra con codelyzer nos aplica las reglas de estilo definidas por el equipo de Angular.
- ◆ Sort TypeScript Imports: nos da un atajo de teclado (o al salvar el fichero) para organizar los imports que utilicemos.
- ◆ TypeScript Hero: nos facilita un atajo de teclado para eliminar todos los imports que no se estén utilizando en el fichero

Visual Studio Plugins

- ◆ vscode-icons: muestra un ícono distinto en función de la naturaleza del fichero, ayuda a identificar más rápidamente los ficheros
- ◆ Angular v5 Snippets:



Visual Studio Plugins

- ◆ Bracket Pair Colorizer: Esta extensión permite identificar los brackets coincidentes con los colores
- ◆ Path Intellisense: Plugin para autocompletado de nombres
- ◆ JSON 2 TS: Convierte un json de clipboard a interface model de Angular
- ◆ Material Icon Theme: extensión que mejora la presentación de iconos y colores de Angular
- ◆ Angular Essentials: extensión de John PApa

Herramientas Online



- ◆ <https://codepen.io/>
- ◆ <http://jsfiddle.net/>
- ◆ <https://plnkr.co/>

Conceptos de ECMAScript

Aprendiendo a conocer el estándar de
codificación



EcmaScript 6, ¿Qué es?

Es la especificación más reciente sobre la que está construido Javascript, ActionScript, TypeScript y cualquier variante del mismo.

La versión 6 empezó a implementarse desde Junio del 2015 y es la especificación usada en los frameworks y navegadores más modernos

Entre los nuevos features podemos encontrar:

1. Variables y Parámetros
2. Clases
3. Promises
4. Módulos

Compatibilidad de EcmaScript 6

| Feature name | Current browser | Compilers/polyfills | | | | | | | | | | Desktop browsers | | | | | | | | | |
|--------------------------------------------------|-----------------|---------------------|------------------|------------------|-----------------|-----------------------|----------|--------------------------|-------|---------|---------|------------------|-----------|-------|-------|------------|---------------|--------------|--------------|--------------|--------------|
| | | Traceur | Babel 6+ core-js | Babel 7+ core-js | Closure 2018.09 | Type-Script + core-js | es6-shim | Kong 4.14 ^[1] | IE 11 | Edge 16 | Edge 17 | Edge 18 Preview | FF 60 ESR | FF 61 | FF 62 | FF 63 Beta | FF 64 Nightly | CH 68, OP 55 | CH 69, OP 56 | CH 70, OP 57 | CH 71, OP 58 |
| Optimisation | | | | | | | | | | | | | | | | | | | | | |
| proper tail calls (tail call optimisation) | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 |
| Syntax | | | | | | | | | | | | | | | | | | | | | |
| default function parameters | 7/7 | 4/7 | 4/7 | 4/7 | 5/7 | 5/7 | 0/7 | 0/7 | 0/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 |
| rest parameters | 5/5 | 4/5 | 3/5 | 3/5 | 2/5 | 4/5 | 0/5 | 0/5 | 0/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 |
| spread syntax for iterable objects | 15/15 | 15/15 | 13/15 | 13/15 | 11/15 | 14/15 | 0/15 | 0/15 | 0/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 |
| object literal extensions | 6/6 | 6/6 | 6/6 | 6/6 | 5/6 | 6/6 | 0/6 | 0/6 | 0/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 |
| for...of loops | 9/9 | 9/9 | 9/9 | 9/9 | 6/9 | 9/9 | 0/9 | 0/9 | 0/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 |
| octal and binary literals | 4/4 | 2/4 | 4/4 | 4/4 | 2/4 | 4/4 | 2/4 | 0/4 | 0/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 |
| template literals | 5/5 | 4/5 | 4/5 | 4/5 | 3/5 | 3/5 | 0/5 | 0/5 | 0/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 |
| RegExp 'y' and 'U' flags | 5/5 | 3/5 | 3/5 | 3/5 | 0/5 | 0/5 | 0/5 | 0/5 | 0/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 |
| destructuring declarations | 22/22 | 20/22 | 21/22 | 21/22 | 20/22 | 21/22 | 0/22 | 0/22 | 0/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 | 22/22 |
| destructuring assignment | 24/24 | 23/24 | 24/24 | 24/24 | 22/24 | 24/24 | 0/24 | 0/24 | 0/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 |
| destructuring parameters | 24/24 | 19/24 | 21/24 | 21/24 | 20/24 | 21/24 | 0/24 | 0/24 | 0/24 | 23/24 | 23/24 | 23/24 | 23/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 |
| Unicode code point escapes | 2/2 | 1/2 | 1/2 | 1/2 | 1/2 | 1/2 | 0/2 | 0/2 | 0/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 |
| new.target | 2/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 0/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 |
| Bindings | | | | | | | | | | | | | | | | | | | | | |
| const | 16/16 | 14/16 | 14/16 | 14/16 | 14/16 | 14/16 | 0/16 | 2/16 | 12/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 |
| let | 12/12 | 10/12 | 10/12 | 10/12 | 10/12 | 10/12 | 0/12 | 0/12 | 10/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 |
| block-level function declaration ^[17] | Yes | Yes | Yes | Yes | Yes | No | No | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Functions | | | | | | | | | | | | | | | | | | | | | |
| arrow functions | 13/13 | 11/13 | 9/13 | 9/13 | 10/13 | 9/13 | 0/13 | 0/13 | 0/13 | 13/13 | 13/13 | 13/13 | 13/13 | 13/13 | 13/13 | 13/13 | 13/13 | 13/13 | 13/13 | 13/13 | 13/13 |
| class | 24/24 | 17/24 | 19/24 | 19/24 | 14/24 | 19/24 | 0/24 | 0/24 | 0/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 |

Tabla de compatibilidad

(<http://kangax.github.io/compat-table/es6/>)

Módulos con EcmaScript 6

¿Qué son?

Un sistema de módulos es un mecanismo para cargar dependencias que serán usadas en una aplicación web. Normalmente esta carga de módulos (En JS) se realiza de forma asíncrona.

Javascript no tenía construida una especificación para módulos, pero la comunidad fue creando varios para suplir la necesidad. Actualmente hay dos importantes propuestas (incompatibles entre sí)

CommonJS y AMD

Módulos con EcmaScript 6

CommonJS: Esta es la implementación usada en [Node.js](#) y Browserify, se caracteriza por una sintaxis sencilla, carga los módulos de forma síncrona y se usa principalmente en el servidor.

Definición de Módulos Asíncronos (AMD): La implementación más popular de este estándar es RequireJS, se caracteriza por una sintaxis un poco más complicada, estar diseñado para cargar módulos de forma asíncrona y se usa principalmente en navegadores.

Asignación por destructuring

- ◆ La sintaxis de **destructuring assignment** es una expresión de JavaScript que hace posible la extracción de datos de arreglos u objetos usando una sintaxis que equivale a la construcción de arreglos y objetos literales.

```
var o = {p: 42, q: true};  
var {p, q} = o;  
  
console.log(p); // 42  
console.log(q); // true  
  
// Asigna nuevos nombres de variable  
var {p: foo, q: bar} = o;  
  
console.log(foo); // 42  
console.log(bar); // true
```

PRÁCTICA

Funciones Flecha

- ◆ La **expresión de función flecha** tiene una sintaxis más corta que una expresión de función convencional
- ◆ Las funciones flecha siempre son anónimas. Estas funciones son funciones no relacionadas con métodos y no pueden ser usadas como constructores

Funciones Flecha

```
var materials = [  
  'impacto',  
  'tecnologico',  
  'cinco',  
  'curso de angular' ];
```

```
console.log(materials.map(material => material.length) );
```

PRÁCTICA

Utilidades de EcmaScript 6

1. CONST
2. LET
3. Template strings
4. Parámetros REST
5. Desestructuración



ES6

- ◆ Las constantes en JavaScript siempre se las ha echado en falta y nos hemos visto obligados a crear pseudo-constantes simplemente estableciendo el nombre de la variable en mayúscula (var PSEUDO_CONSTANTE) y dando por hecho que nadie la modificará posteriormente, algo que se podía hacer sin problemas en JavaScript, y ahora como debe de ocurrir en una constante, ya no variará el propio valor por mucho que se intente modificar.

CONST

```
// x: constante  
const x = 20;  
console.log(x); // 20
```

```
x = 1;  
console.log(x); // 20  
// No cambia el valor de x
```

- ◆ Además const tiene un buen soporte, siendo quizás la característica de ECMAScript 6 con mayor soporte, pudiendo ser usado en Chrome, Safari, Firefox, Node.js e Internet Explorer 11, entre otros navegadores.

LET

- ◆ let se usa para declarar variables y controlar mejor su propagación. Es bastante simple de entender, evita que una variable sea visible cuando no debe como ocurre con var.

```
let a = 1;  
let b = 2;  
  
{  
  let c = 3;  
}  
  
console.info(a, b, c); // ReferenceError: c is not defined
```

Template Strings

- ◆ Con el nuevo estándar ES6 se han introducido las Plantillas de Texto de forma nativa permitiendo con ello cubrir, al menos parcialmente, una necesidad habitual del desarrollo moderno. Pasemos a estudiarlas con detalle.
- ◆ Las plantillas de texto (o Template Strings) son cadenas literales de texto incrustadas en el código fuente que permiten su interpolación mediante expresiones.

Template Strings

- ◆ Su sintaxis es sencilla y consta de dos aspectos fundamentales:
- ◆ Se utilizan las comillas invertidas para delimitar las cadenas
- ◆ Para añadir marcadores de posición (o placeholders) utilizamos el signo de dolar (\$) seguido de llaves

Template Strings

- ◆ Gracias a las plantillas, ahora es posible disfrutar de cadenas multilínea en Javascript sin necesidad de recurrir a concatenaciones, barras invertidas, unión de arrays u otros malabares habituales
- ◆ Es importante recalcar que las plantillas mantienen el formato introducido, incluyéndose los saltos de línea y las tabulaciones

Template Strings

```
var myTemplateString = 'La donna e mobile cual piuma al vento';
console.info(`String value: ${ myTemplateString }` );
// String value: La donna e mobile cual piuma al vento
```

Definición de elementos Promises y Observables





Un Observable es un objeto que guarda un valor y que emite un evento a todos sus suscriptores cada vez que ese valor se actualiza

Observables

- ◆ Angular está orientado a la Programación Reactiva. Esto significa que está diseñando para reaccionar en base a la propagación de cambios de flujos de datos (eventos) asíncronos.
- ◆ Desde Angular 4, cuando un objeto cambia se lo notifica a aquellos que están interesados en detectar sus cambios (suscriptores). Este tipo de objetos, que se pueden “observar”, se denominan **Observables**.

- ◆ Angular utiliza la librería de flujos asíncronos RxJS, que ofrece varias utilidades para trabajar con *Observables*.
- ◆ La librería RxJS aporta operadores para transformar los resultados de los Observables, como el operador **map** (que se usa para manipular cada elemento de un array), el operador **debounce** para ignorar eventos demasiado seguidos, el operador **merge** para combinar los eventos de 2 o más observables en uno...

Observables

- ◆ Es un stream de eventos que pueden ser manipulados en cualquier momento. A diferencia de un promise, nos permite conocer el estatus de la petición durante su ejecución.
- ◆ Los observables son cancelables
- ◆ Los observables tienen métodos que nos permiten reintentar el servicio según una respuesta específica
- ◆ Un observable nos permite tener un listener en cada cambio de valores en la ejecución de un proceso



*Una promesa representa un valor que
puede estar disponible ahora, en el
futuro, o nunca.*

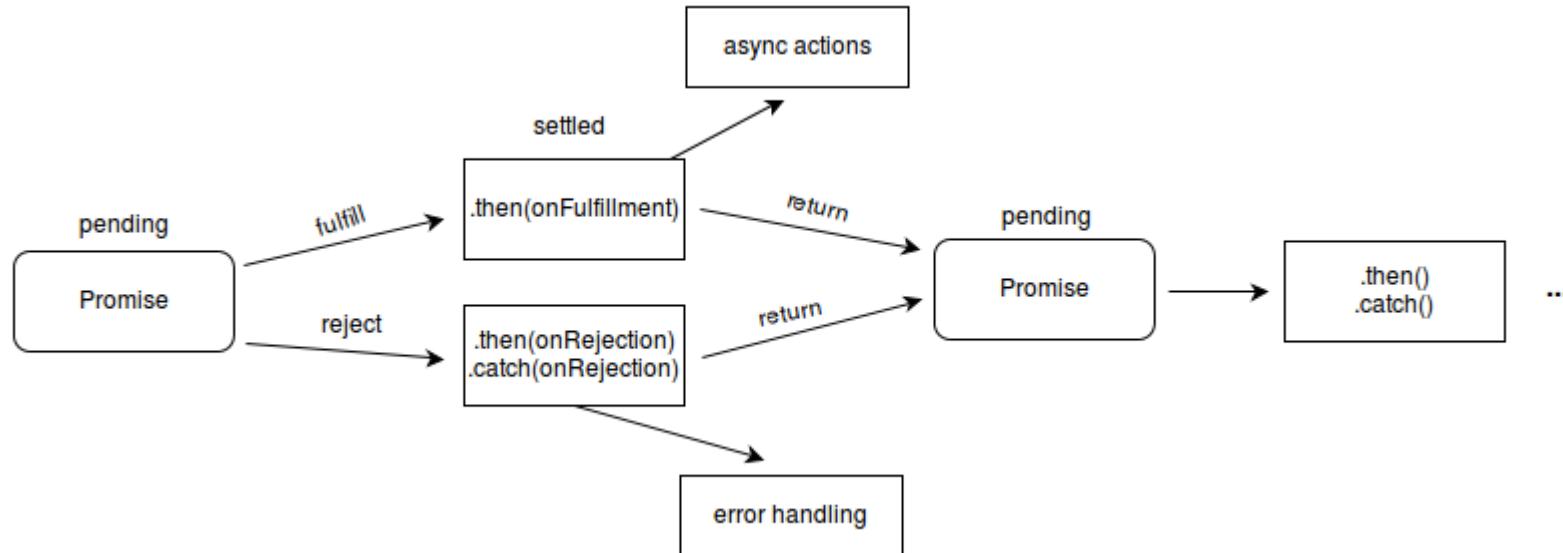
Promises

- ◆ Una **Promesa** es un proxy para un valor no necesariamente conocido en el momento que es creada la promesa.
- ◆ Permite que métodos asíncronos devuelvan valores como si fueran síncronos: en vez de inmediatamente retornar el valor final, el método asíncrono devuelve una *promesa* de suministrar el valor en algún momento en el futuro

Promises

- ◆ Una Promesa se encuentra en uno de los siguientes estados:
 - ◆ pendiente (pending): estado inicial, no cumplida o rechazada.
 - ◆ cumplida (fulfilled): significa que la operación se completó satisfactoriamente.
 - ◆ rechazada (rejected): significa que la operación falló.
- ◆ Una promesa pendiente puede ser *cumplida* con un valor, o *rechazada* con una razón (error).

Promises



Introducción a TypeScript

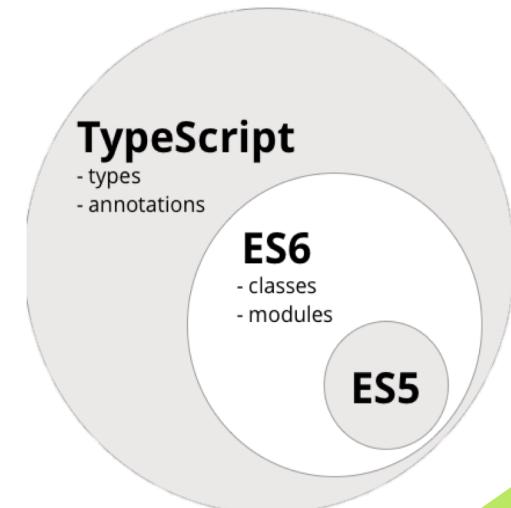


TypeScript

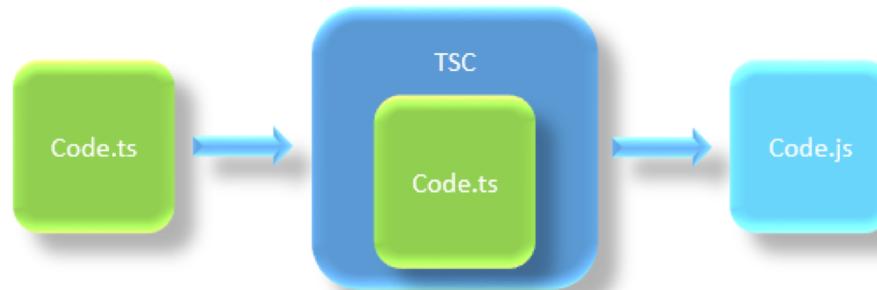
Es una **extensión** de javascript desarrollada por Microsoft que nos permite usar todas las features de ES2015.

Además proporciona un enfoque más orientado a objetos a nuestro código, haciéndolo **más escalable**.

Angular está escrito en Typescript



Esto quiere decir que en nuestro proyecto usaremos archivos **.TS** en lugar de **.JS**. Para poder ejecutar código TypeScript necesitaremos un **transpiler**



- ◆ La característica fundamental de TypeScript es que compila en Javascript nativo, por lo que se puede usar en todo proyecto donde se esté usando Javascript.
- ◆ Cuando se usa TypeScript en algún momento se realiza su compilación, convirtiendo su código a Javascript común. El navegador, o cualquier otra plataforma donde se ejecuta Javascript, nunca llegará a enterarse que el código original estaba escrito en TypeScript, porque lo único que llegará a ejecutar es el Javascript resultante de la compilación.

TypeScript

- ◆ TypeScript es lo que se conoce como un "superset" de Javascript, aportando herramientas avanzadas para la programación que traen grandes beneficios a los proyectos.
- ◆ Los superset compilan en el lenguaje estándar, por lo que el desarrollador programa en aquel lenguaje expandido, pero luego su código es "transpilado" para transformarlo en el lenguaje estándar, capaz de ser entendido en todas las plataformas.

TypeScript

- ◆ Existen dos superset populares para Javascript. Por un lado tenemos CoffeeScript y por otro lado TypeScript. La diferencia principal es que mientras que CoffeeScript te aleja del lenguaje, con TypeScript escribes en un lenguaje que es muy similar al propio Javascript.
- ◆ TypeScript es verdaderamente orientado a objetos, trayendo herramientas como la herencia, sobrecarga, etc.
- ◆ En resumen tiene cosas que suenan a lenguajes como Java, C++, C#, etc.

Qué necesitamos para usar TypeScript

- ◆ El TSC (Command-line TypeScript Compiler), la herramienta que nos permite compilar un archivo TypeScript a Javascript nativo.
- ◆ Este software está realizado con NodeJS y su instalación se realiza vía npm con el siguiente comando:

```
npm install -g typescript
```

tsconfig.json

- ◆ La presencia de un archivo tsconfig.json en un directorio indica que el directorio es la raíz de un proyecto de TypeScript. El archivo tsconfig.json especifica los archivos raíz y las opciones de compilación necesarias para compilar el proyecto.

```
◆ {  
◆   "compilerOptions": {  
◆     "module": "commonjs",  
◆     "removeComments": true,  
◆     "sourceMap": true  
◆   }  
◆ }
```

```
var a:number = 9;  
a += 4;
```

```
function mostrar(b:string) :void{  
    console.log(b);  
}  
mostrar('hola');
```

PRÁCTICA

- ◆ Luego transpilamos con:

```
tsc ejemplo.ts
```

Diferencias entre ES6 y TypeScript

- ◆ Variables tipadas
- ◆ La principal característica de **TypeScript** por encima de Javascript es que **permite definir de qué tipo son las variables** que se van a usar.
- ◆ Veamos un **ejemplo**

```
class MyClass{  
    constructor(someArg){  
        this.myArray = someArg;  
    }  
    someMethod(){  
        for(let item of this.myArray){  
            console.log(item);  
        }  
    }  
}  
  
let someVar = 123456;  
let myClassInstance = new  
MyClass(someVar);  
myClassInstance.someMethod();
```

```
class MyClass{  
    myArray:Array<number>;  
    constructor(someArg:string){  
        this.myArray = someArg;  
    }  
    someMethod(){  
        for(let item of this.myArray){  
            console.log(item);  
        }  
    }  
}  
  
let someVar:number = 123456;  
let myClassInstance:MyClass = new  
MyClass(someVar);  
myClassInstance.someMethod();
```

TypeScript

◆ <https://www.typescriptlang.org/play/>

PRÁCTICA



Declaración de variables

- ◆ En TypeScript debemos seguir los siguientes criterios al momento de definir identificadores de variables:
 - Pueden contener caracteres alfanuméricos, pero no puede empezar con valores numéricos.
 - No deben contener caracteres especiales, excepto guión bajo (_) o dólar (\$).
 - No deben contener palabras reservadas.
 - Son case sensitive y no debe tener espacios.
- ◆ Para TypeScript declarar una variable es similar a Javascript, aunque también se le puede indicar el tipo de dato.

```
var test: string = 'test test...';
```

Tipos de datos

- Booleans:
 - `var bool: boolean = false;`
- Strings
 - `var name: string = "lparra";`
- Number
 - `var age: number = 34;`
- Any
 - `var any: any = "any";`
 - `any = 23;`
 - `any = false;`
- Array
 - `var hobbies: Array<string> = ["Family", "Program"];`
- Enum
 - `enum Color {Red, Green, Blue};`
 - `var c: Color = Color.Blue;`
- Void
 - `function setId(id: number): void{`
 - `console.log(id);`
 - `}`
- Funciones
 - `function getId(): number{`
 - `return 1;`
 - `}`

Modificadores de acceso

- ◆ En TypeScript contamos con modificadores de acceso para atributos de clases.
 - **public**: cuando no se indica un modificador de acceso a un atributo o método se considera público. Cuando se añade a un parámetro dentro de un constructor, ese atributo se añadirá a la estructura de la aplicación.
 - **private**: los atributos o métodos de una clase con este modificador de acceso solo tendrán acceso dentro de si mismo.
 - **static**: los atributos o métodos de una clase con este modificador de acceso se tendrá acceso desde la clase.

Datos miembro públicos/privados

◆ En el constructor podemos **declarar directamente** los parámetros que recibimos como datos miembro públicos o privados. Es decir, no solo indico si es público o privado, sino que además automáticamente realizo la asignación del parámetro al dato miembro

◆ Práctica:

Hello.ts:

```
class HelloWorld {  
    constructor(private message: string) {}  
}
```

Interfaces

```
interface IUser{  
    name: string;  
    getName(): string;  
}  
  
class User implements IUser{  
    name: string;  
    constructor(name: string) {  
        this.name= name;  
    }  
    getName(): string{  
        return this.name;  
    }  
}  
var user = new User("Jose Julian");  
console.log(user);
```

Las **interfaces en typescript** nos permiten crear contratos que otras clases deben firmar para poder utilizarlos, al igual que en otros lenguajes de programación como java o php, En **typescript** también podemos implementar más de una interfaz en la misma clase.

PRÁCTICA

- ◆ Este es uno de los componentes vitales a la hora de conseguir una mejor arquitectura en las aplicaciones Javascript.
- ◆ Vamos a reestructurar el ejemplo anterior haciendo que la interfaz IUser pueda importarse en otras clases...

Modulos

```
// iuser.ts
interface IUser{
    name: string;
    getName(): string;
}
export = IUser;
```

```
import IUser = require('IUser');

class User implements IUser{
    name: string;
    constructor(name: string) {
        this.name= name;
    }
    getName(): string{
        return this.name;
    }
}
export = User
```

◆ Modificamos objeto usuario

PRÁCTICA

◆ Ahora vamos usar estos módulos en un nuevo fichero

Main.ts:

```
import User = require('user')
import Animal = require('IUser')

function sayHi(usuario:Usuario) {
    console.log("hola " + usuario.name);
}

sayHi(new Usuario("Jose Julian"))
```

Decoradores

- ◆ Los decorators son una de las características más interesantes de typeScript, vienen a ser una serie de metadatos adicionales que se pueden añadir a clases, métodos, propiedades y parámetros para modificar su comportamiento. Aunque no son estrictamente decorators de ts, sino una variante denominada annotations
- ◆ Vamos a crear un decorador/annotation para trazar métodos:

Decoradores

```
function trazar(target: Object, key: string, descriptor: any) {  
    console.log('Clase: ', target.constructor.name);  
    console.log('Método: ', key);  
    console.log('Características del método: ', descriptor);  
}
```

PRÁCTICA

Decoradores

```
class Ejercicio {  
    respuesta: string;  
    @trazar  
    responder(respuesta:string) {  
        this.respuesta = respuesta;  
    }  
    getRespuesta(): string {  
        return this.respuesta;  
    }  
}  
  
let ejercicio = new Ejercicio();  
ejercicio.responder('jump!');
```

Unidad 2

Conociendo Angular

Versiones, Estructura y HolaMundo



Introducción a novedades de Angular

- ◆ Angular no es una evolución del AngularJS, sino más bien la reescritura del framework. Por ello una migración de aplicación AngularJS a Angular, realmente es volver a hacerla. Algunas de las novedades son las siguientes:
- **Lazy Load:** En Angular existe la posibilidad de retrasar la carga de ciertas partes de la aplicación, hasta que realmente sea necesario usarlas.



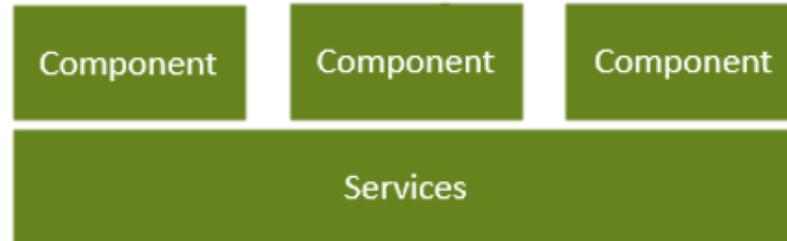
Introducción a novedades de Angular

- **Data-binding:** Ahora el desarrollador es capaz de decidir en qué sentido debe implementarse el binding. Esto permite al desarrollador, optimizar las aplicaciones al punto de llegar a ser hasta 5 veces más rápidas.
- **Aplicaciones universales:** Permite renderizar HTML del lado del servidor, lo que ofrece la posibilidad de creación de aplicaciones isomórficas.



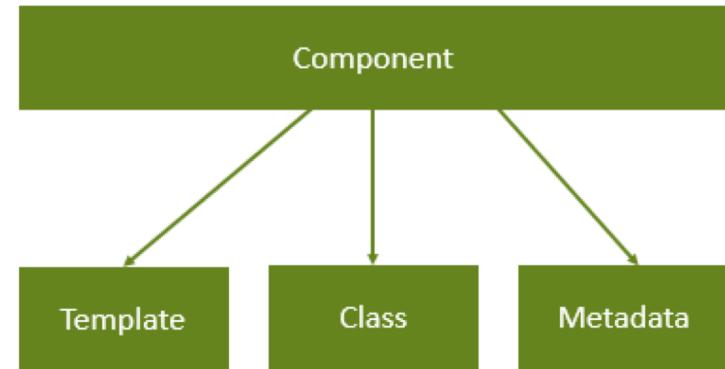
¿Qué elementos conforman una aplicación Angular?

- ◆ Una aplicación Angular está orientada a componentes. En ese sentido una aplicación está compuesta de múltiples componentes que cumplen con una funcionalidad determinada por su área funcional.
- ◆ El objetivo de los componentes es poder crear piezas de software altamente reutilizables dentro del proyecto, o en otros proyectos. Los componentes en una aplicación pueden utilizar servicios comunes.



¿Qué elementos conforman una aplicación Angular?

- ◆ Cada componente está compuesto por tres elementos esenciales:
 - ◆ **Clase:** son las clases representan a la implementación del componente.
 - ◆ **Metadatos:** es la configuración que se inyecta a un componente a través de un decorador.
 - ◆ **Plantillas:** es la plantilla HTML que representa la vista de ese componente



AngularJS VS Angular 2



- Los Controladores y \$scopes han desaparecido
- La sintaxis de las directivas ha cambiado
- Angular es capaz de usar los elementos y eventos estándares del DOM
- Cambia la sintaxis y el manejo de inyección de dependencias
- Cambios en el routing de la aplicación

Angular 2 vs Angular 4



¿Qué ha cambiado?

- Aplicaciones más ligeras y rápidas
- AoT vs JiT
- El paquete de animaciones se separa del core
- Mejoras en la internacionalización

Angular 2 vs Angular 4



¿Qué ha cambiado?

- Se agrega un else a la directiva *ngIf:

```
<div *ngIf="user.length > 0; else empty"><h2>Users</h2></div>
<ng-template #empty><h2>No users.</h2></ng-template>
```

- Se agrega el uso de un alias al *ngFor:

```
<div *ngFor="let user of users | slice:0:2 as total; index as = i">
  {{i+1}}/{{total.length}}: {{user.name}}
</div>
```

Angular 2 vs Angular 4



¿Qué ha cambiado?

- Se introduce un nuevo pipe: **titlecase**

```
<h2>{{ 'anil singh' | titlecase }}</h2>  
<!-- OUPPUT - It will display 'Anil Singh' --&gt;</pre>
```

Angular 2 vs Angular 4



¿Qué ha cambiado?

- Simplifica el paso de parámetros en llamadas http:

```
//Angular 4 -  
http.get(` ${baseUrl}/api/users` , { params: { sort: 'ascending' } } );  
  
//Angular 2-  
const params = new URLSearchParams();  
params.append('sort', 'ascending');  
  
http.get(` ${baseUrl}/api/users` , { search: params } );
```

Angular 2 vs Angular 4



¿Qué ha cambiado?

- Se incluye la disponibilidad de Angular Universal a través del paquete `@angular/platform-server`

¿Cómo llamarlo?



- ¿AngularJS?
- ¿Angular 1?
- ¿Angular 2?
- ¿Angular 4?
- ¿Angular 5?
- ¿Angular 6?



¿Muy bien, solo Angular, pero, por qué?



◆ Comando de añadir dependencias ng-add

- ◆ ng-add <package> utiliza el gestor de paquetes (npm/yarn/...) para descargar el paquete y sus dependencias, y de paso ejecutar su script de instalación (*schematics*) si existe.

HolaMundo

Empecemos a programar en Angular



Angular CLI

Angular Cloning

Starting

- ◆ Clonamos:
- ◆ git clone

<https://github.com/angular/quickstart.git>

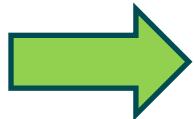
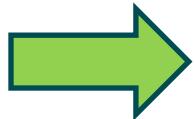
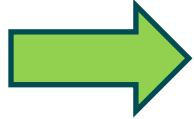
Creando y configurando una app

Archivos de configuración

Nuestra app debe tener los siguientes archivos en el directorio raíz

- ◆ package.json Identifica las dependencias NPM del proyecto
- ◆ tsconfig.json Define cómo el compilador typescript generará los build JS

Package.json



```
1  [
2    "name": "chatbotbluetab",
3    "version": "0.0.0",
4    "scripts": {
5      "ng": "ng",
6      "start": "ng serve",
7      "build": "ng build",
8      "test": "ng test",
9      "lint": "ng lint",
10     "e2e": "ng e2e"
11   },
12   "private": true,
13   "dependencies": {
14     "@angular/animations": "^6.1.0",
15     "@angular/common": "^6.1.0",
16     "@angular/compiler": "^6.1.0",
17     "@angular/core": "^6.1.0",
18     "@angular/forms": "^6.1.0",
19     "@angular/http": "^6.1.0",
20     "@angular/platform-browser": "^6.1.0",
21     "@angular/platform-browser-dynamic": "^6.1.0",
22     "@angular/router": "^6.1.0",
23     "core-js": "^2.5.4",
24     "rxjs": "^6.0.0",
25     "zone.js": "~0.8.26"
26   },
27   "devDependencies": {
28     "@angular-devkit/build-angular": "~0.7.0",
29     "@angular/cli": "~6.1.3",
30     "@angular/compiler-cli": "^6.1.0",
31     "@angular/language-service": "^6.1.0".
```



Instalando

- ◆ Empezaremos con una **instalación global** que permita usar la herramienta desde cualquier directorio.
- ◆ Comprobaremos la versión instalada y accederemos a la ayuda en línea. La ayuda está disponible tanto de modo general como para cada comando que vayamos a usar.

```
$ npm i -g @angular/cli@latest  
$ ng -v  
$ ng help  
$ ng help new
```

- ◆ Para poder trabajar con Angular utilizamos nuestra querida herramienta Angular CLI.
- ◆ **Angular CLI** es un proyecto desarrollado por la misma gente de Google para facilitarnos el trabajo de crear una estructura nueva de proyecto, creación de componentes / servicios / directivas, servir la aplicación en un entorno de desarrollo y realizar build, entre otras útiles tareas

- ◆ Para que Angular CLI sepa que hacer, se vale de un archivo llamado **Angular CLI Workspace**.
- ◆ Este archivo se encuentra dentro de la estructura de nuestro proyecto y se identifica con el nombre **angular.json**.

Crear y ejecutar una aplicación Angular 6

- ◆ Una vez que hemos instalado el CLI de manera global ya puedes empezar a usarlo en tu directorio de trabajo.
- ◆ El primer comando será `ng new` que va a generar toda una aplicación funcional y las configuraciones necesarias para su depuración, pruebas y ejecución.

Componentes de nuestra app

Como en Angular 1.x debemos definir un **módulo principal** donde se especificarán las dependencias de nuestra app. En angular a los módulos los llamamos **componentes** y son clases con la siguiente forma

```
/* Elemento principal de nuestra app */

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

@NgModule({ //Decorator
  imports:      [ BrowserModule ]
})

//Definimos nuestro componente/clase
export class AppModulol { }
```

- ◆ **Imports** : Como en cualquier lenguaje de POO importamos los componentes necesarios para nuestra aplicación
- ◆ **Decorators** : Identificados con el símbolo @ (@NgModule / @Component). Especifica metadata importante para la definición de nuestra clase, debe ser declarada justo antes de la clase/módulo/componente.
- ◆ **Módulo** : Un Módulo, define un conjunto de componentes.
- ◆ **Component** : Los **components** son el bloque básico de construcción de una app en Angular 2, un component **controlará** cierta porción de un template (Análogo a controller/ng-controller)

- ◆ En una aplicación normal necesitaremos dependencias como
 - ◆ **FormsModule**
 - ◆ **RouterModule** : análogo a ui-router de Angular 1.x
 - ◆ **HTTP** : Análogo a \$http en Angular 1.x

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

@Component({
  selector: 'my-app',
  template: '<h1>Mi primera app Angular!</h1>'
})

export class AppComponent { }
```

Ahora, veamos el **módulo** con la inclusión del componente

```
import { NgModule }           from '@angular/core';
import { BrowserModule }    from '@angular/platform-browser';
import { AppComponent1 }     from './app.componente1'; //Componente que acabamos de definir

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent1 ],
  bootstrap:   [ AppComponent1 ]
})

export class AppModulo1 { }
```

Creando nuestra app

El archivo **principal.ts o main.ts** es el archivo donde importaremos nuestro módulo y haremos bootstrap de nuestra aplicación

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModulo1 } from './app/app.modulol';

const platform = platformBrowserDynamic();

platform.bootstrapModule(AppModulo1);
```

Creando nuestra app con el CLI

```
ng new cursoAngular -S  
cd cursoAngular  
npm start  
// Si todo va bien, en segundos podemos ir a http://localhost:4200 para ver en marcha la aplicación
```

- ◆ Volvamos a la terminal y analicemos la primera línea:
 - ◆ `ng new cursoAngular -S`

Los archivos más relevantes

- ◆ **angular.json** : *configuración del propio CLI. La base de todos los configuradores*
- ◆ **package.json** : *dependencias de librerías y scripts*
- ◆ **src/** : *la carpeta donde están los archivos fuentes*
 - ◆ **index.html** : *un fichero HTML índice estándar*
 - ◆ **main.ts** : *fichero TypeScript de arranque de la aplicación*
 - ◆ **app/** : *la carpeta con el código específico de tu aplicación*
 - ◆ **app.module.ts** : *las aplicaciones son árboles de módulos, y este es su raíz*
 - ◆ **app.component.ts** : *las páginas son árboles de componentes, y este es su raíz*

- ◆ Angular CLI instala y configura un conjunto de herramientas que nos harán la vida más fácil. Entre otras, destaca la capacidad de recargar la aplicación en caliente en cuanto guardamos el trabajo.
- ◆ Para probarlo sólo tenemos que dejar arrancada la aplicación con el comando `npm start`; cambiar un fichero de código y comprobar el resultado en el navegador. Lo haremos más adelante....

El generador de angular CLI

- ◆ El comando `ng generate` será el más utilizado durante el desarrollo, nos ayuda a generar nuevos componentes, módulos, servicios, rutas, pipes y otras funcionalidades.
- ◆ Gracias al **generador de angular cli** podemos realizar lo siguiente de forma simple.
 - ◆ **Generar módulos** con `ng generate module mi-modulo`
 - ◆ **Generar componentes** con `ng generate component mi-componente`
 - ◆ **Generar servicios** con `ng generate service mi-servicio`
 - ◆ **Generar pipe** con `ng generate pipe mi-servicio`

Otras utilidades que se pueden generar con angular cli

- ◆ Generar clases: `ng generate class [nombre]`
- ◆ Generar interface: `ng generate interface [nombre]`
- ◆ Generar Guard: `ng g guard mi-guard`
- ◆ Generar directivas: `ng generate directive [nombre]`

El generador de angular CLI

- ◆ La gran mayoría los comandos utilizado en el **generador de angular clic** tienen una abreviatura:
 - ◆ `ng g m [nombre]` : Genera un módulo.
 - ◆ `ng g c [nombre]` : Genera un componente.
 - ◆ `ng g s [nombre]` : Genera un servicio.
 - ◆ `ng g i [nombre] <type>` : Genera una interfaz.
 - ◆ `ng g d [nombre]` : Genera directiva.
 - ◆ `ng g p [nombre]` : Genera Pipe.
 - ◆ `ng g g [nombre]` : Genera Guard.
 - ◆ `ng g e [nombre]` : Genera Enum.
 - ◆ `ng g cl [nombre]` : Genera clase.

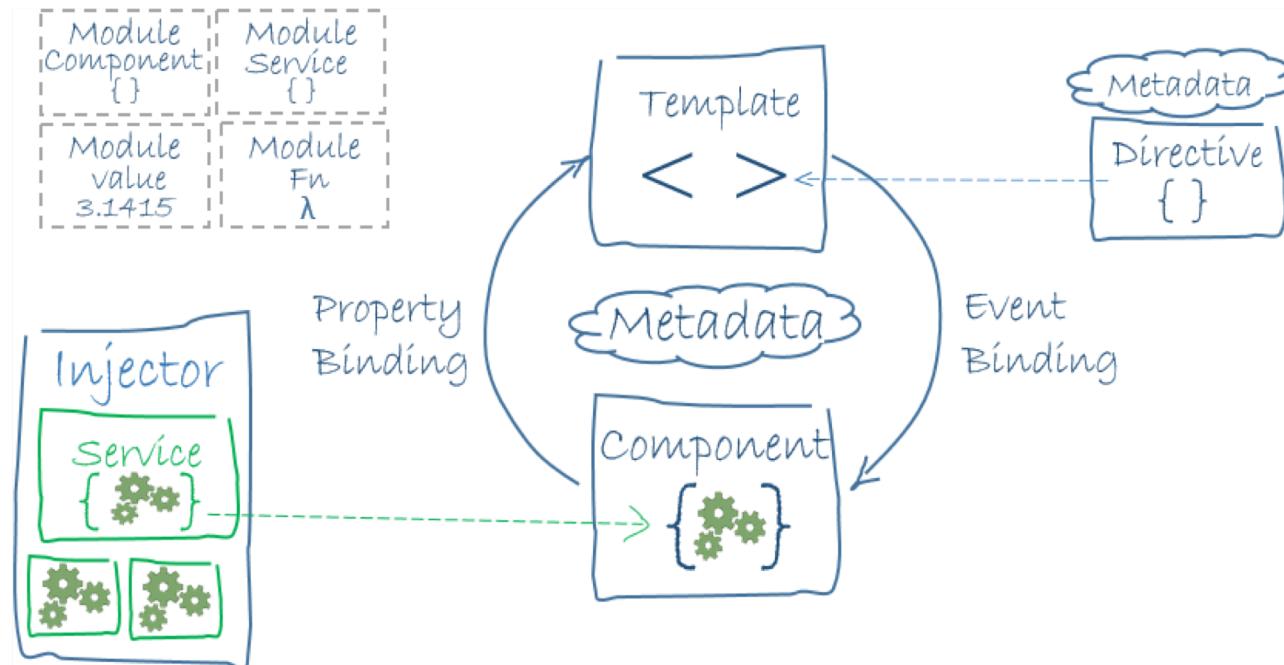
Unidad 3

Los Elementos Angular

Módulos, Componentes, Directivas...



- Módulos
- Componentes
- Templates
- Metadata
- Data binding
- Directivas
- Servicios
- Inyección de dependencias



Módulos

- ◆ Las apps de Angular son modulares, gracias a su propio sistema de módulos llamado Angular Modules (o NgModules).
- ◆ Por otro lado, al desarrollar Angular en TypeScript utilizarás también los módulos de ES6 (para gestionar librerías de JS). No los confundas, no tienen nada que ver entre sí.

Módulos: exportar / importar

- ◆ Esto aplica a todo código que utilice el nuevo estandar de JS, no tiene nada que ver con Angular.
- ◆ Imagina que quieres exportar un nuevo componente AppComponent que tienes definido en el archivo app.component.ts. Con la palabra reservada export:

```
//app/app.component.ts
export class AppComponent {
    //aquí va la definición del componente
}
```

Módulos: exportar / importar

- ◆ Para importarlo en otro lado, por ejemplo en main.ts, utilizarás la palabra reservada import, junto con nombre del objeto a importar y el path del archivo:
 - ◆ //app/main.js
 - ◆ import { AppComponent } from './app.component';

Librerías de Angular

- ◆ Angular está empaquetado como una colección de librerías Javascript vinculadas a distintos paquetes npm. Así solo tienes que importar lo que realmente necesitas y consigues una web más ligera.
- ◆ Las librerías principales de Angular (siempre comienzan por @angular) son:
 - ◆ @angular/core
 - ◆ @angular/platform-browser
 - ◆ @angular/router
 - ◆ @angular/forms
 - ◆ @angular/http

Módulos

- ◆ Un módulo de Angular, es un conjunto de código dedicado a un ámbito concreto de la aplicación, o una funcionalidad específica y se define mediante una clase decorada con `@NgModule`.
- ◆ Toda aplicación de Angular tiene al menos un módulo de Angular, el módulo principal (o root module).

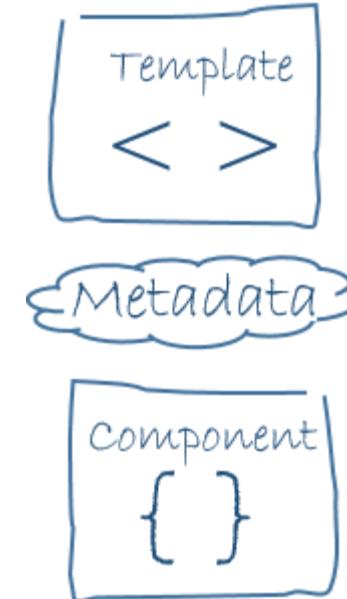
- ◆ Los metadatos más importantes de un *NgModule* son:
- ◆ **declarations**: Las vistas que pertenecen al módulo. Hay 3 tipos de clases de tipo vista: componentes, directivas y *pipes*.
- ◆ **exports**: Conjunto de declaraciones que deben ser accesibles para templates de componentes de otros módulos.
- ◆ **imports**: Otros *NgModules*, cuyas clases exportadas son requeridas por templates de componentes de este módulo.
- ◆ **providers**: Los servicios que necesita este módulo, y que estarán disponibles para toda la aplicación mediante inyección de dependencias.
- ◆ **bootstrap**: Define la vista raíz. Utilizado solo por el *root module*. El componente que se asigne será el componente raíz de la app. De él penderá cualquier otro componente que se utilice.

Root Module

- ◆ Es el módulo principal de Angular. Por convenio se le llama AppModule y se encuentra en el archivo app.module.ts.
- ◆ La clase AppModule sirve para cargar la aplicación e indicar todas sus dependencias. Como el resto de módulos, se declara con el decorador NgModule
- ◆ Vamos al código...!

Componente

- ◆ Un Component controla una zona de espacio de la pantalla que podríamos denominar vista.
- ◆ Un componente es una clase estándar de ES6 decorada con `@Component`.
- ◆ Vamos al código



Metadatos

- ◆ La forma de añadir metadatos a una clase en TypeScript es mediante el patrón decorador, justo antes de la declaración de la clase.
 - ◊ Selector: Es un selector de CSS que indica a Angular que debe crear e instanciar mi componente cuando se encuentra un elemento con ese nombre en el HTML. Es decir, cuando Angular se encuentra <todo-list></todo-list> en un template HTML, entre esos tags inserta una instancia de mi TodoListComponent.
 - ◊ templateUrl: La url en la que se encuentra el template que quieras vincular al componente
 - ◊ styleUrls: Un array con urls a archivos de estilos que queremos aplicar a nuestro componente. No entraré ahora al detalle, pero Angular implementa una característica interesante de HTML5 denominada Shadow DOM, que permite aislar los estilos de un componente con respecto al resto.

Data binding

- ◆ Uno de los principales valores de Angular es que nos abstrae de la lógica pull/push asociada a insertar y actualizar valores en el HTML y convertir las respuestas de usuario (inputs, clicks, etc) en acciones concretas. Escribir toda esa lógica a mano (lo que típicamente se hacía con JQuery) es tedioso y propenso a errores, y desde Angular 2 lo resuelve por nosotros gracias al Data Binding.

Interpolación: (Hacia el DOM)

Es la mejor forma conocida y más fácil de hacer data binding. En Angular 1, Se veía mas o menos así

```
$scope.employee = { name : "Jose"};  
<h1>{ employee.name }</h1>
```

Donde employee debía estar dentro de nuestra variable **\$scope**

En angular 4, **no existe \$scope**. Cada objeto o variable son atributos de nuestra clase/componente. La sintaxis incluye llaves dobles.

```
<h1>{ employee.name }</h1>
```

Event binding: (Desde el DOM)

- ◆ Al hacer `(click)="selectTodo(todo)"`, le indicamos a Angular que cuando se produzca un evento click sobre esa etiqueta `<div>`, llame al método `selectTodo` del Componente, pasando como atributo un objeto *todo* presente en ese contexto.

Two-way binding: (Desde/Hacia el DOM)

- ◆ Un caso importante que no hemos visto con los ejemplos anteriores es el binding bi-direccional, que combina event binding y property binding, como podemos ver en el siguiente ejemplo:

```
<input [(ngModel)]="todo.subject">
```

- ◆ En este caso, el valor de la propiedad fluye a la caja de input como en el caso property binding, pero los cambios del usuario también fluyen de vuelta al componente, actualizando el valor de dicha propiedad.

¿Cómo añadimos dinamismo?

```
...  
  
@Component({  
  selector: 'my-app',  
  template: '<h1>????</h1>'  
})  
  
class AppComponent1 {  
  title = 'Mi primera app Angular!';  
}  
  
...
```

```
...  
  
@Component({  
  selector: 'my-app',  
  template: '<h1>{{title}}</h1>'  
})  
  
class AppComponent1 {  
  title = 'Mi primera app Angular!';  
}  
  
...
```

TemplateUrl

- ◆ Podemos usar un archivo html propio en lugar del tener junto con el componente todo el código de la vista
- ◆ Angular ofrece la propiedad templateUrl para asignar un archivo html a nuestro componente.
- ◆ Todas las variables de clase que tenga nuestro componente estarán accesibles en el template vía interpolación

```
...
@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component1.html'
})
```

- ◆ Vamos a añadir un nuevo componente usando la extensión V6 Snippets
- ◆ Añadamos data-binding usando interpolaciones de una variable definida en el componente
- ◆ Añadimos el componente al módulo

EJERCICIO

Atributos

- ◆ Los componentes pueden tener atributos, tanto de entrada como de salida

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

@Component({
  selector: 'componente-uno',
  templateUrl: 'app/app.component1.html'
})

export class AppComponent1 implements OnInit {
  @Input()
  entrada:string;

  titulo = 'Mis cursos!';
  cursos = [...];

  ngOnInit() {
    console.log(this.entrada);
  }
}
```

EJERCICIO

Atributos

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

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>
<componente-uno [entrada]="name"></componente-uno>
` ,
})
export class AppComponent { name = 'Angular'; }
```

¿Cómo añadimos MAS dinamismo?

```
...  
  
@Component({  
    selector: 'my-app',  
    template: `<h1>{{titulo}}</h1>  
        <h2>{{curso.nombre}}</h2>  
        <p>{{curso.descripcion}}</p>  
        <p>{{curso.duracion}} horas</p>  
`)  
  
class AppComponent1 {  
    titulo = 'Mi primera app Angular con interpolaciones!';  
    curso = { "id": 1, "nombre": "Angular 2", "descripcion":  
    "Introducción al desarrollo avanzado en Js", "duracion": 8 };  
}  
...
```

Usando ` en vez de comillas simples, establecemos un string multilinea

EJERCICIO

Añadiendo estilos

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{titulo}}</h1>
    <h2>{{curso.nombre}}</h2>
    <p>{{curso.descripcion}}</p>
    <p>{{curso.duracion}} horas</p>`  

  styles:[`  

    .titulo{  

      font-weight: 600;  

      font-style: italic;  

      margin-bottom: 0.2em;  

    }  

  `]  

})
...
`]
```

EJERCICIO

Directivas

- ◆ En Angular 2, una directiva se define como una clase que utiliza el decorador @Directive.
- ◆ Las directivas permiten añadir comportamiento dinámico a nuestras apps Angular
- ◆ Existen 3 tipos de directivas:
 - Components
 - Structural
 - Attribute



Directivas estructurales

- ◆ Estas directivas, que se diferencian fácilmente al ser precedidas por un asterisco, alteran el *layout* añadiendo, eliminando o reemplazando elementos en el DOM
- ◆ ***ngIf:** si la condición se cumple, su elemento se inserta en el DOM, en caso contrario, se elimina del DOM. (equivale al *ng-if* de AngularJS)
- ◆ ***ngFor:** repite su elemento en el DOM una vez por cada item que hay en el iterador que se le pasa, siguiendo una sintaxis de ES6. (equivale al *ng-for* de AngularJS)

Directivas Estructurales

- ◆ ¿Cómo manejamos el tener más objetos dentro de nuestra clase pero en el mismo atributo?

```
...  
  
class AppComponent {  
    titulo = 'Mis cursos!';  
    cursos = [  
        { "id": 1, "nombre": "Angular 2", "descripcion": "Introducción  
al desarrollo avanzado en Js", "duracion": 8 },  
        { "id": 2, "nombre": "NodeJs", "descripcion": "Introducción al  
server-side en Js", "duracion": 4 }];  
}  
...
```

Directivas Estructurales

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{titulo}}</h1>
    <ul>
      <li *ngFor="let curso of cursos">
        <h2>{{curso.nombre}}</h2>
        <p>{{curso.descripcion}}</p>
        <p>{{curso.duracion}} horas</p>
      </li>
    </ul>`})
...
```

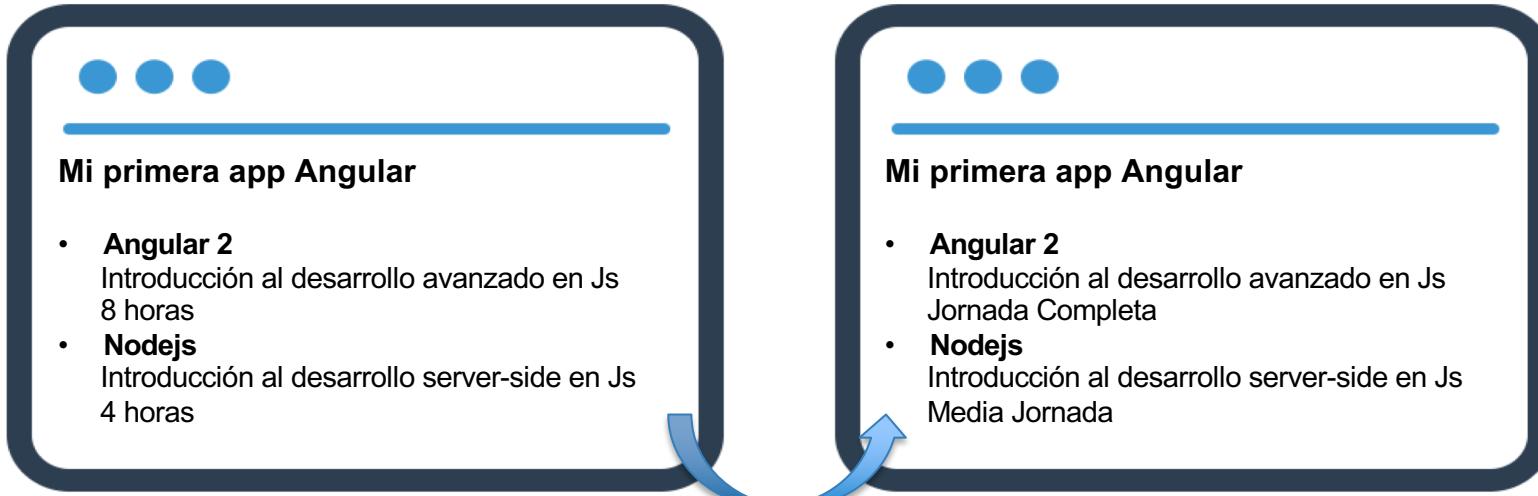
*ngFor es una directiva estructural

curso es una variable local

cursos es el array

Directivas Estructurales

- Alteran nuestra interfaz gráfica añadiendo, eliminando o reemplazando componentes html
- También nos permiten evaluar condiciones: *ngIf



Directivas Estructurales: *ngIf

```
...  
  
@Component ({  
    selector: 'my-app',  
    template: `<h1>{{titulo}}</h1>  
        <ul>  
            <li *ngFor='let curso of cursos'>  
                <h2>{{curso.nombre}}</h2>  
                <p>{{curso.descripcion}}</p>  
                <p *ngIf='curso.duracion === 8'>Jornada Completa</p>  
                <p *ngIf='curso.duracion < 8'>Media Jornada</p>  
            </li>  
        </ul>`  
})  
...  
...
```

También podemos usar else y then

```
<div *ngIf="condition; else elseBlock">...</div>
<ng-template #elseBlock>...</ng-template>
```

Usando then:

```
<div *ngIf="condición; then thenBlock else elseBlock"></div>
<ng-template #thenBlock>...</ng-template>
<ng-template #elseBlock>...</ng-template>
```

Práctica:

Mejorar el ejercicio anterior de *ngIf para usar else

Directivas Estructurales: NgSwitch

- ◆ Para usar ng switch debemos escribir las posibles opciones

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

@Component({
  selector: 'feature3',
  template: `
    <button (click)="value=1">select - 1</button>
    <button (click)="value=2">select - 2</button>
    <button (click)="value=3">select - 3</button>
    <h5>You selected : {{value}}</h5>
```

Directivas Estructurales: NgSwitch

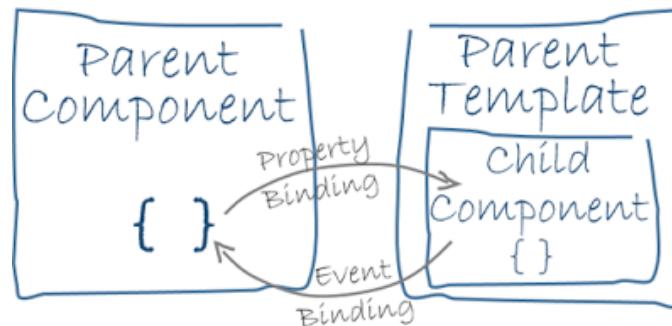
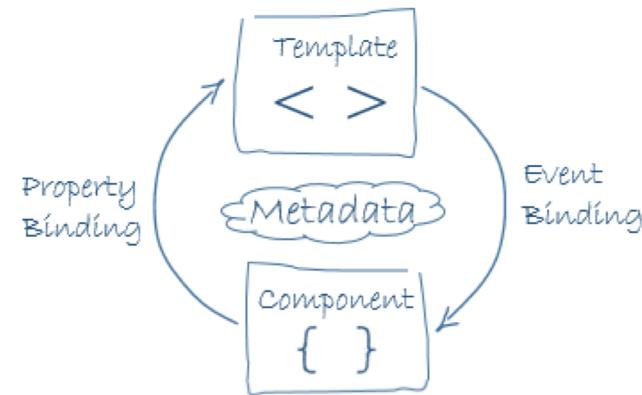
```
<hr>
<div [ngSwitch]="value">

<div *ngSwitchCase="1">1. Template - <b>{{value}}</b> </div>
<div *ngSwitchCase="2">2. Template - <b>{{value}}</b> </div>
<div *ngSwitchCase="3">3. Template - <b>{{value}}</b> </div>
<div *ngSwitchDefault>Default Template</div>

</div>
` ,
}

export class Feature3Component {
value:number = 0;
}
```

Procesamiento del Data Binding



Introducción a eventos

- Añadamos un click simple:

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{titulo}}</h1>
    <ul>
      <li *ngFor='let curso of cursos' (click)="seleccion(curso)">
        <h2>{{curso.nombre | uppercase}}</h2>
        <p>{{curso.descripcion}}</p>
        <p *ngIf='curso.duracion === 8'>Jornada Completa</p>
        <p *ngIf='curso.duracion < 8'>Media Jornada</p>
      </li>
    </ul>`})
...
...
```

Directivas Atributo

- ◆ Las directivas atributo alteran la apariencia o el comportamiento de un elemento existente en el DOM y su apariencia es igual a la de atributos HTML
- ◆ ngClass: Esta directiva permite añadir/eliminar varias clases a un elemento de forma simultánea y dinámica.
- ◆ Por un lado, necesitaremos un método en nuestro componente que gestione el estado de las clases CSS que queremos aplicar a nuestro elemento, por ejemplo:

En el componente

```
@Component ({  
    selector: 'selector-dos',  
    templateUrl: './app.componente2.html',  
    styleUrls:[ './style.css' ]  
})  
.....  
  
getCSSClasses(modo:string) {  
    let cssClasses;  
    if(modo == 'noche') {  
        cssClasses = {  
            'one': true,  
            'two': true  
        }  
    } else {  
        cssClasses = {  
            'two': true,  
            'four': false  
        }  
    }  
    return cssClasses;  
}
```

Style.css

```
.one {  
color: green;  
}  
.two {  
font-size: 20px;  
}  
.three {  
color: red;  
}  
.four {  
font-size: 15px;  
}
```

En el template

```
<div [ngClass]="getCSSClasses('noche')">  
  Texto en modo noche  
</div>  
  
<div [ngClass]="getCSSClasses('dia')">  
  Texto en modo dia  
</div>
```

Directivas de atributo

- ◆ ngModel:
- ◆ Implementa un mecanismo de binding bi-direccional. El ejemplo típico es con el elemento HTML <input>, donde asigna la propiedad value a mostrar y además responde a eventos de modificación.
- ◆ <input [(ngModel)]="todo.subject" >

Directivas Atributo

- ◆ **ngStyle:** esta directiva permite asignar varios estilos inline a los elementos

```
setStyles() {
    let styles= {
        'text-decoration': this.finalizado ? 'line-through' :
'none',
        'font-weight': this.important ? '600' : 'normal',
    };
    return styles;
}
```

```
<div [ngStyle]="setStyles()"></div>
```

Directivas Propias

- ◆ Es posible escribir nuestras propias directivas para modificar las cualidades de algún elemento del DOM que tenga algunos de los componentes que componen nuestra app. También podríamos resumir una directiva como un componente sin template.

Directivas Propias

- ◆ Primero definimos la directiva dentro del contexto de un componente

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[colorer]'
})

export class ColorerDirective{
  constructor(el: ElementRef) {
    el.nativeElement.style.color = 'red';
  }
}
```

Directivas Propias

- ◆ Ahora para hacer uso de esa directiva en nuestro template sería algo similar a lo que veremos a continuación:
 - ◆ <h1>My directive</h1>
 - ◆ <p colorer>Color from directive!</p>
- ◆ Ahora como ultimo paso debemos incluir nuestra directiva en el ngModules que se encuentra en el punto de entrada de nuestra app

Pipes

- ◆ Los pipes en Angular >2 son lo mismo que los filtros en AngularJS, es decir, nos permiten alterar la forma en la que se van a visualizar los datos, entre otras cosas, prácticamente todos los filtros existentes en AngularJS siguen existiendo en Angular >2

Pipes

- Usamos pipes para formatear data

```
...
@Component({
  selector: 'my-app',
  template: `<h1>{{titulo}}</h1>
    <ul>
      <li *ngFor='let curso of cursos'>
        <h2>{{curso.nombre | uppercase}}</h2>
        <p>{{curso.descripcion}}</p>
        <p *ngIf='curso.duracion === 8'>Jornada
Completa</p>
        <p *ngIf='curso.duracion < 8'>Media Jornada</p>
      </li>
    </ul>`})

```

◆ Vamos a formatear el precio...

```
...  
  
class AppComponent {  
    titulo = 'Mis cursos!';  
    cursos = [  
        { "id": 1, "nombre": "Angular 2", "descripcion": "Introducción  
al desarrollo avanzado en Js", "duracion": 8, "precio":50.50 },  
        { "id": 2, "nombre": "Nodejs", "descripcion": "Introducción al  
desarrollo server-side en Js", "duracion": 4, "precio":75.25 }  
    ];  
    ...  
}
```

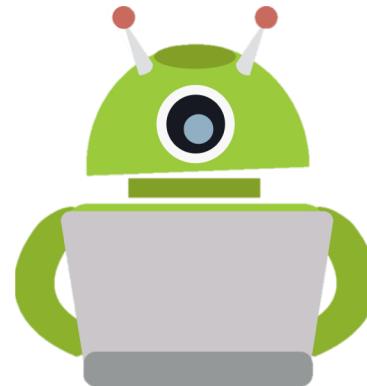
Pipes

- Cuales pipes están disponibles?
- <https://angular.io/docs/ts/latest/> -> API Reference -> Pipe -> CurrencyPipe

```
...  
  
    <li *ngFor='let curso of cursos'>  
        <h2>{{curso.nombre}}</h2>  
        <p>{{curso.descripcion}}</p>  
        <p *ngIf='curso.duracion === 8'>Jornada  
Completa</p>  
        <p *ngIf='curso.duracion < 8'>Media Jornada</p>  
        <p>{{curso.precio | currency:'EUR':true}}</p>  
    </li>  
...
```

Pipes - Ejercicio

- ◆ Añadir una propiedad a cada objeto de cursos con la fecha del día o fecha específica
- ◆ Formatear la presentación para verla en formato “30-12-2017”



Pipes

- ◆ Los **pipe** son funciones separadas a un componente, son funciones que te servirá en cualquier parte de la web.
- ◆ Se pueden utilizar en 1 o varios componentes, para no programar varias veces la misma función, o crear un componente solo para esto una tarea, **angular** nos da la oportunidad de crear una componente “lite”.

```
<!-- html -->
<p>Mi aniversario es el {{ day | date }}</p>
```

```
//component.ts
day = new Date(1988, 3, 15);
```

Pipes

- ◆ Un pipe en angular se activa usando (|) y el nombre del “filtro”, en este caso solo muestra la fecha pero podemos cambiar el formato por ejemplo.

```
<p> {{ day | date:"MM/dd/yy" }} </p>
```

Personalizar pipe en angular

- ◆ Crearemos un archivo multiplicador.pipe.ts. Exportamos una clase llamada MultiplicadorPipe
- ◆ Para añadir el nombre a nuestro Pipe, utilizaremos el decorador @Pipe

```
import { Pipe } from '@angular/core';
@Pipe({ name: ' multiplicador' })
export class MultiplicadorPipe { }
```

Personalizar pipe en angular

- ◆ Para transformar los datos de entrada en un pipe, implementamos la interfaz PipeTransform importada de @angular/core

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({ name: ' multiplicador' })
export class MultiplicadorPipe implements PipeTransform {
  transform(value: any, ...args: any[]): any;
}
```

- ◆ Al implementar esta interfaz, es obligatorio utilizar la función transform la cual recibe un parámetro obligatorio y debe retornar un valor transformado.

Transformar valor de entrada

- ◆ ¿Como recibe un pipe el valor de entrada? Al utilizar en nuestro HTML {{ unValor | multiplicador}} recibiremos en la función transform como primer parámetro el valor de **unValor**
- ◆ Y ahora podemos implementar la lógica necesaria para retornar una fecha válida.
- ◆ `transform(valorRecibido: number, factor: string): number`

Pipes personalizados

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'multiplicador'})

export class MultiplicadorPipe implements PipeTransform {

  transform(valorRecibido: number, factor: string): number {
    let fac = parseFloat(factor);
    let mult = (valorRecibido * fac);
    return mult;
  }
}
```

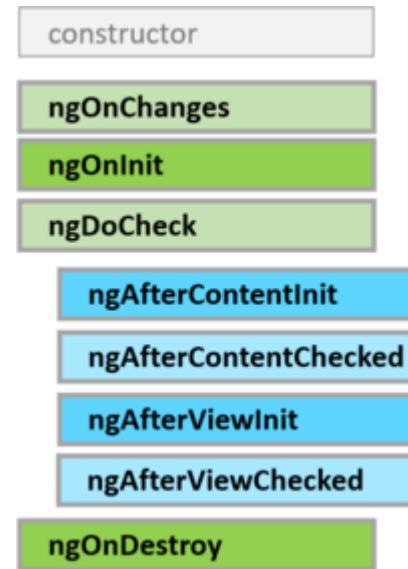
¿Qué nos falta por hacer?



Los hooks del ciclo de vida

- ◆ Los hooks del ciclo de vida, también llamados *lifecycle hooks* son unos mecanismos de Angular que reaccionan ante ciertos eventos muy generales, como el inicio o destrucción de un componente, y que nos permiten programar lógicas que se desencadenen cuando se produzcan esos eventos.

Los hooks del ciclo de vida



LOS PRINCIPALES HOOKS

- ◆ En Angular existen varios hooks. Además, algunos pueden variar de una versión a otra. Los principales, que existen desde Angular 2 y se han mantenido por su utilidad:
- ◆ **OnInit.** Nos permite implementar el método `ngOnInit()`, que se ejecuta al inicio de un componente, justo después del constructor de la clase. Ya lo hemos utilizado en varias ocasiones.
- ◆ **OnChanges.** Mediante el método `ngOnChanges()` detecta cambios en una variable que esté bindeada a una propiedad de un componente padre, y que llegue al componente por `@Input()`.

LOS PRINCIPALES HOOKS

- ◆ **DoCheck.** Con el método `ngDoCheck()` detecta cualquier cambio en cualquier parte de la aplicación.
- ◆ **OnDestroy.** El método `ngOnDestroy()` se ejecuta cuando se sale de un componente para entrar en otro, o para cerrar la aplicación. Detecta el final del ciclo de vida del componente.

AfterContentInit y AfterContentChecked

- ◆ **ngAfterContentInit** se ejecuta una vez después del primer ngDoCheck y no se vuelve a ejecutar en todo el resto de vida del componente.
- ◆ **ngAfterContentChecked** se ejecuta después de cualquier ejecución de ngDoCheck en todo la vida del componente.
- ◆ Ambos se ejecutan después de que angular proyecta contenido externo en el componente

AfterViewInit y AfterViewChecked

- ◆ Estos hooks son muy similares a ngAfterContentInit y ngAfterContentCheck, solo que se invocan *después* de ngAfterContentChecked, siguiendo el mismo patrón
- ◆ **ngAfterContentInit** se invoca una sola vez después del primer ngAfterContentChecked, y no se vuelve a invocar.
- ◆ **ngAfterViewChecked**, se ejecuta después de cualquier invocación de ngAfterContentChecked
- ◆ La diferencia entre ngAfterContentChecked es que estos hooks marcan el final de la composición de la vista de un componente y se define dentro del componente mismo, no desde el exterior

LOS PRINCIPALES HOOKS

- ◆ Todos estos hooks requieren, para su uso, ser importados en la lógica del componente donde los vayamos a utilizar, desde @angular/core (es decir, son parte del núcleo de Angular).
- ◆ Además, son interfaces que deben ser implementadas en la clase de dicho componente. Además, en el caso de OnChanges, debemos importar (aunque no implementar) SimpleChanges

Unidad 4

Manejo de Servicios

Modelos, Servicios y Http



- ◆ Angular permite crear clases específicas para definir los objetos:

- ◆ **curso.ts**

```
export class Curso{  
    id: number;  
    nombre: string;  
    descripcion: string;  
    duracion: number;  
    precio: number;  
}
```

- ◆ Ver:

Class-validator

- ◆ `npm install class-validator –save`
- ◆ Se instalará y se añadirá el módulo a la carpeta `node_modules` del proyecto

◆ En el modelo importamos los decoradores

```
import { IsCurrency, IsDate, IsInt, Length, Max, Min
} from 'class-validator';
```

◆ En el componente importamos el método validate

```
◆ import { validate } from '../../node_modules/class-
validator';
```

◆ En el código validamos:

```
validate(curso).then(resultado => {
  if (resultado.length > 0) {
    console.log('validation failed. errors: ', resultado);
  } else {
    console.log('validation succeed');
  }
});
```

Uso de class-validator

- ◆ Validar que la propiedad Id sea entero y tenga como mínimo valor 1
- ◆ Validar que el nombre tenga longitud mínima de 10 y máxima de 20 personalizando el mensaje
 - ◆ 'El nombre no tiene la logintud requerida, debe tener mínimo [minimo] y máximo [maximo], pero tiene [longitud actual]'
- ◆ Validar que el precio sea un número y que tenga valor positivo
- ◆ Validar que la duración sea un entero y que sea mayor de 1 y menor de 50

PRÁCTICA



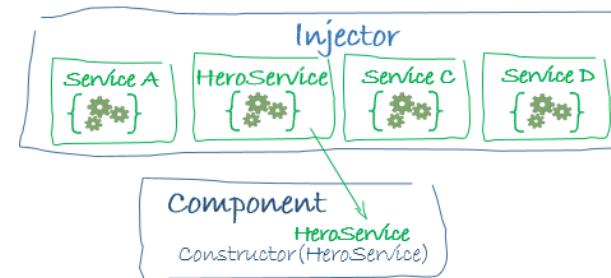
Los servicios nos proveen el mecanismo para compartir funcionalidades entre componentes.

Servicios

- ◆ Se utilizan para encapsular todo valor, función o característica o lógica de negocio necesaria para la webapp
- ◆ Sus usos más comunes son:
 - ◆ Recuperar datos del servidor
 - ◆ Validar *inputs* de usuario
 - ◆ Realizar logs en consola
- ◆ Ejemplos de servicios:
 - ◆ Servicio de logging
 - ◆ Servicio de datos
 - ◆ Bus de mensajes
 - ◆ Cálculo de Impuestos
 - ◆ Configuración de la app



- ◆ La primera vez que se inyecta un servicio, el inyector lo instancia y lo guarda en un contenedor. Cuando inyectamos un servicio, antes de nada el inyector busca en su contenedor para ver si ya existe una instancia.



- ◆ Cuando el inyector no tiene el servicio que se le pide, sabe cómo instanciar uno gracias a su *Provider*

Servicios

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

Servicio para Log

◆ Creamos logger.service.ts

```
import { Injectable } from '@angular/core';
@Injectable()
export class Logger {
  log(msg: any) { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any) { console.warn(msg); }
}
```

- ◆ **@Injectable: avisa a Angular de que el servicio espera utilizar otros servicios** y genera los metadatos que necesita el servicio para detectar la Inyección de Dependencias (*DI*) en el constructor.

Providers

- ◆ En Angular el ***provider*** es la propia **clase** que define el **servicio**
- ◆ Los ***providers*** pueden registrarse en cualquier nivel del árbol de componentes de la aplicación a través de los metadatos de componentes, o a nivel raíz, en el ***NgModule*** de la aplicación.
- ◆ Al registrar un ***provider*** en el ***NgModule***, éste **estará disponible para toda la aplicación**. Si el servicio solo afecta a una pequeña parte de la webapp como puede ser un componente o un componente y sus hijos, tiene más sentido que se declare a nivel de componente.

Uso de providers hasta Angular 5

```
import { Logger } from './logger.service';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, AppComponent1 ],
  bootstrap:    [ AppComponent ],
  providers:    [ Logger ]
})

-----

@Component({
  selector:    'componente-uno',
  templateUrl: 'componente1.component.html',
  providers:    [Logger]
})
export class ComponenteUno {
```



los servicios solo son *singletons* dentro del *scope* de su *Injector*. Al registrar un provider en un componente, cada vez que se instancie el componente se obtiene una nueva instancia del servicio disponible para dicha instancia del componente y todos sus subcomponentes.

Cómo usar un servicio

- ◆ En el módulo:
 - ◆ Importamos la clase Logger
 - ◆ Usamos el metadato **providers**
- ◆ En el componente:
 - ◆ Importamos la clase Logger
 - ◆ Añadimos el parámetro en el constructor
 - ◆ Usamos la clase y métodos para loggear
- ◆ Gracias a *TypeScript*, Angular sabe de qué servicios depende un componente con tan solo mirar su constructor.

PRÁCTICA

- ◆ Ahora bien, el consumidor de nuestro servicio (component), no sabe de dónde el servicio obtendrá la data. Nuestro service pudiese traer data de cualquier lugar y es completamente aislado de nuestros componentes. Puede venir de una API, de localStorage, de una cookie o de un mock de data.
- ◆ Esa es "la magia" de la independencia de componentes y la modularización.

Providers en Angular 6

- ◆ Hasta ahora, declarabas los providers o bien en el NgModule, o bien en un componente concreto.
- ◆ A partir de ahora ya no tienes que declararlos en ningún lado. Solo tienes que especificar en qué nivel hay que introducir el provider desde el propio decorador Provider.
- ◆ De este modo los servicios solo se incluirán en los módulos que realmente lo necesiten, reduciendo el tamaño del *bundle* de la aplicación.

Tree-shaking en los providers

- ◆ El Tree-shaking es un proceso de evaluación de código fuente capaz de decidir qué funciones se utilizan o no en los módulos que importamos.
- ◆ Si alguna función no se utiliza, esta no se incluye en el *bundle* final para reducir su peso con código innecesario.
- ◆ La novedad en esta versión es que ahora esto puede aplicarse a los providers, ahora en vez de que los módulos hagan referencia a los servicios, serán los propios servicios los que refencien a los módulos.

Tree-shaking en los providers

- ◆ En el decorador @Injectable tenemos una nueva propiedad llamada providedIn.
- ◆ Con esta propiedad podemos decirle a Angular a qué módulo registrar nuestro servicio en lugar de tener que importar el módulo y registrarlo a los proveedores de un NgModule.
- ◆ Por defecto, esta sintaxis lo registra en el inyector raíz, lo que hará que nuestro servicio sea un singleton de toda la aplicación. El proveedor root es un valor razonable predeterminado para la mayoría de los casos de uso.

Tree-shaking en los providers

```
import { Injectable } from '@angular/core';
@Injectable({ providedIn: 'root' })
export class SharedService {
  constructor() { }
}
```

- ◆ Con esta nueva API, no tuvimos que importar el servicio en NgModule para el registro que no hicimos una dependencia explícita. Como no existe una instrucción de importación, las herramientas de compilación pueden garantizar que este servicio solo se incluya en nuestra aplicación si un componente lo usa

Práctica

- ◆ Crear una nueva variante del servicio de logs para que utilice la nueva sintaxis

PRÁCTICA

Singleton

- ◆ Se crea un *singleton* por cada módulo en el que se provea un servicio. Normalmente si el servicio es para un sólo módulo funcional se provee en este y nada más.
- ◆ Si va a ser compartido, entonces gana la opción de auto proveerlo en el raíz, garantizando así su disponibilidad en cualquier otro módulo de la aplicación.
- ◆ Pero siempre será **una instancia única por módulo**. Si un *singleton* no es lo adecuado, entonces puedes proveer el mismo servicio en distintos módulos.
- ◆ De esa forma se creará una instancia distinta para cada uno. Si se provee la misma clase en dos o más módulos se genera una instancia en cada uno de ellos. Los componentes recibirán la instancia del módulo jerárquicamente más cercano.

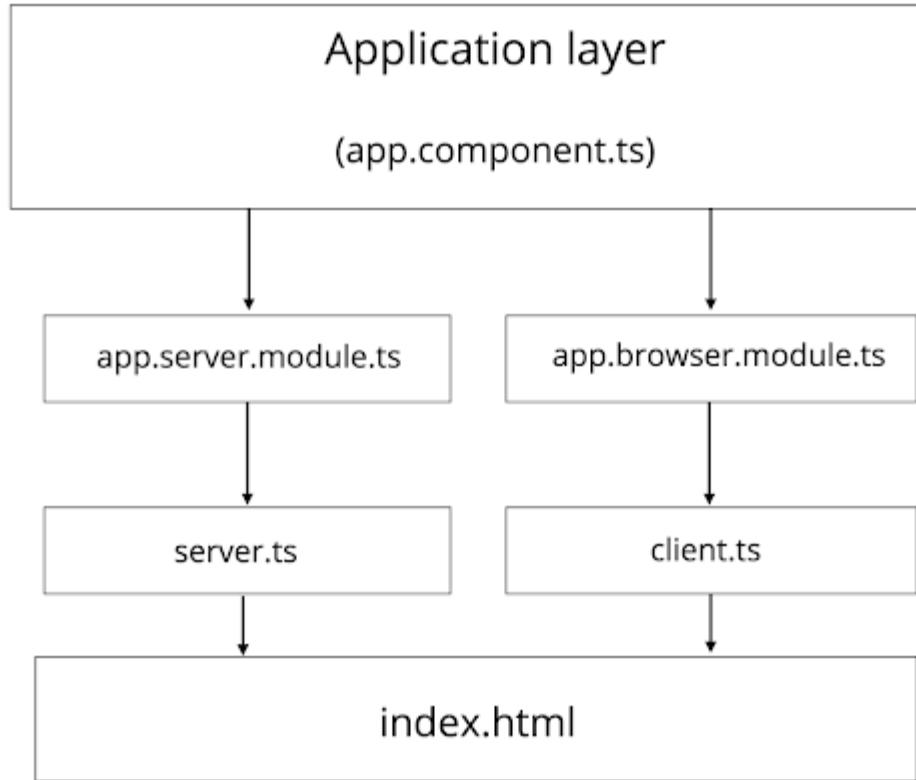
Angular Universal

- ◆ Angular Universal es la tecnología que te **permite ejecutar una aplicación Angular desde el servidor.**
- ◆ El *Server Side Rendering (SSR)* es útil por cuestiones de SEO, para compartir páginas en redes sociales (con su *thumbnail*), o para reducir el tiempo de carga inicial
- ◆ Para ejecutar Angular en servidor necesitamos un módulo que actúe como punto de entrada: **src/app/app.server.module.ts**.

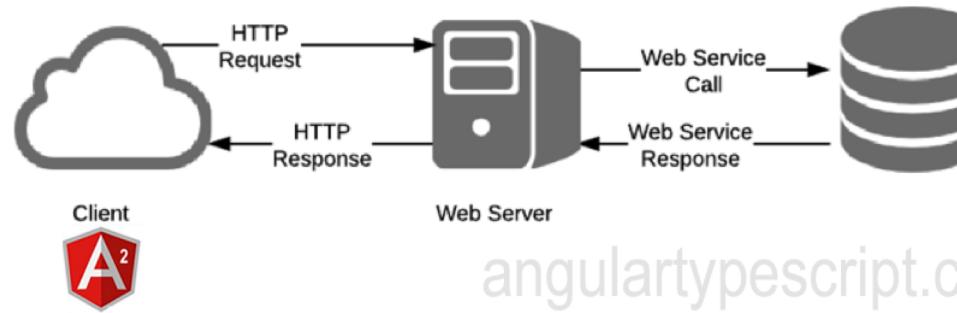
Angular Universal

- ◆ Angular Universal nos da las herramientas para que casi no tengamos que tocar el código: La capa de aplicación se mantiene intacta y en la capa de renderizado se añade un segundo elemento que permite generar el DOM como un string para devolverlo como respuesta a la petición del navegador.

Angular Universal Architecture



HTTP en Angular



angulartypescript.com

<https://reqres.in/>

Vamos a organizarnos

- ◆ Crear un nuevo módulo llamado Feature1Module
- ◆ Crear un nuevo model llamado Usuario con
 - ◆ "id",
 - ◆ "first_name"
 - ◆ "last_name"
 - ◆ "avatar"
- ◆ Crear un nuevo componente llamado UserComponent dentro del módulo Feature1Module
- ◆ Importar en Feature1Module el servicio Logger y hacerlo disponible en el componente
- ◆ Importar Feature1Module en AppModule

Vamos a organizarnos

◆ Crear un nuevo modelo Response con:

- ◆ page: number;
- ◆ per_page: number;
- ◆ total: number;
- ◆ total_pages: number;
- ◆ data: Array<Usuario>;

◆ Para añadir acceso a apis externas necesitamos:

- Tener data accesible vía http
- Tener las librerías necesarias para acceso http
- Informar al inyector sobre el http provider
- Inyectar la dependencia http a nuestro servicio y hacer la llamada de obtención de data
- Escuchar la data recibida y procesarla

- ◆ Las peticiones HTTP en Angular nos permiten enviar peticiones normalmente desde nuestro cliente a un servidor.
- ◆ En versiones anteriores de Angular teníamos disponible la clase Http (`import {Http} from '@angular/http'`), la cual ya ha deprecado y no debe utilizarse, en su lugar, desde Angular 5 ha aparecido HttpClient (`import {HttpClient} from '@angular/common/http'`), que presenta varias mejoras

Añadiendo soporte HTTP

- ◆ Lo primero que debemos hacer es añadir las librerías Angular para peticiones http.
- ◆ Podemos añadirlas en cualquier módulo del sistema, pero lo haremos en el módulo Feature1Module

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

@NgModule ({
  declarations: [ UsuarioComponent ],
  imports: [ HttpClientModule ],
  providers: [ LoggerService ]
})
class Feature1Module { }
...
```

services/remote-data.service.ts

```
import { Response } from './models/response';
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable()
export class RemoteDataService {

  constructor(private httpclient: HttpClient) {
  }

  getUserData() {
    return this.httpclient.get<Response>('https://reqres.in/api/users');
  }
}
```

Componente

¿Qué nos falta por hacer?

```
...  
export class UserComponente {  
    responseLocal: Response;  
...  
    ngOnInit () {  
        this.remoteDataService.getUserData().subscribe(response => this.  
responseLocal = response);  
    }  
}
```



Error Handling

- ◆ Podemos incluir en nuestro código un manejo sencillo de errores de las peticiones http.

```
...  
  
ngOnInit() {  
    this.remoteDataService.getConfigData().subscribe(  
        response => {  
            this.response = response  
        },  
        err => {  
            console.log("Ocurrio un error." + err)  
        }  
    );  
}
```



Práctica

- ◆ Crear bloque html ul li para recorrer el array de Response.data y muestre los usuarios obtenidos y en un tag img el avatar de cada uno

Ejercicio

- ◆ Crear una función en el componente llamada: GuardarUsuario que ejecute una llamada post via httpClient

Interceptores

- ◆ El uso de interceptores se trata de cambiar las solicitudes salientes y las respuestas entrantes, pero no podemos alterar la solicitud original: debe ser inmutable. Para hacer cambios, necesitamos clonar el request original.
- ◆ Al clonar el request original, podemos establecer los encabezados que queremos. Un caso es muy simple y común es: agregar un Authorization Header

Interceptores

- ◆ El primer paso para crear un interceptor es crear una clase Injectable que implemente HttpInterceptor.
 - ◆ `export class AuthInterceptor implements HttpInterceptor { ... }`
- ◆ Esto significa que nuestra nueva clase debe tener un método llamado intercept con los parametros HttpRequest y HttpHandler.
 - ◆ `intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {`

```
intercept(request: HttpRequest<any>, next: HttpHandler):  
Observable<HttpEvent<any>> {  
  
    const token = this.getToken();  
    if (!!token) {  
        request = request.clone({  
            setHeaders: {  
                Authorization: 'Bearer ' + token  
            }  
        });  
    }  
    return next.handle(request);  
}
```

- ◆ El interceptor necesita ser agregado a la matriz `HTTP_INTERCEPTORS`.
- ◆ Esto se hace haciendo que la matriz `HTTP_INTERCEPTORS` existente use la nueva clase que hemos creado.
- ◆ Agregamos esto en el provider del módulo principal de nuestra aplicación.

```
import { HttpClientModule, HTTP_INTERCEPTORS } from  
'@angular/common/http';  
...  
  
providers: [  
  {  
    provide: HTTP_INTERCEPTORS,  
    useClass: AuthInterceptor,  
    multi: true  
  }  
]
```



EXAMEN

Examen

1) Modificamos comportamiento de nuestra página usando:

- a) classes
- b) decorators
- c) directives

2) Para tener disponibles los servicios debemos decirle al inyector de dependencias el _____ que necesitamos:

- a) component
- b) provider
- c) directive
- d) model

3) el elemento básico de una app angular es:

- a) module
- b) class
- c) directive

Examen

- 4) Como best practices en Angular ¿cuántos componentes se recomienda que tenga un archivo?
- a) máximo 2
 - b) 1
 - c) no es importante
- 5) Es buena práctica especificar clases con propiedades y data en su propio archivo y estos son:
- a) models
 - b) templates
 - c) components
 - d) mocks
- 6) Un componente como mínimo contiene:
- a) template
 - b) decorator
 - c) module

Examen

7) Usamos _____ para agrupar componentes y servicios que dependen entre sí en un bloque de código particular

- a) class
- b) module
- c) component

8) En la declaración de un servicio debemos usar la anotación:

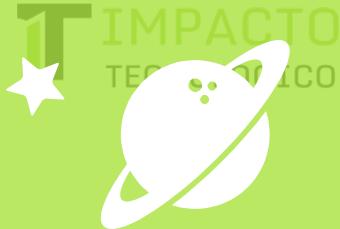
- a) @Service
- b) @Injectable
- c) @Annotation
- d) No hace falta hacerlo

9) *ngFor y *ngIf son ejemplos de directivas

- a) structural
- b) attribute
- c) components

RxJs

*Una API para la programación asíncrona con
flujos observables*



¿Qué es la programación reactiva?

- ◆ La programación Reactiva es programación orientada al manejo de streams de datos asíncronos y la propagación del cambio.
- ◆ Un stream es un flujo de datos y tradicionalmente los streams han estado ligados a operaciones I/O como lectura/escritura de ficheros o querys a base de datos.
- ◆ En RxJs, no es muy diferente ya que sea cual sea el origen de la información, será tratada como un stream.

¿Qué es RxJS?

- ◆ Es una librería para crear programas asincrónicos y basados en eventos mediante el uso de secuencias observables
- ◆ Documentación:
 - ◆ <http://reactivex.io/rxjs/>
- ◆ Repositorio de la librería:
 - ◆ <https://github.com/ReactiveX/rxjs>

Elementos Clave

- ◆ **Observable:** Representa la idea de una colección invocable de valores o eventos futuros.
- ◆ **Observer:** Es una colección de callbacks que “sabe” escuchar los valores entregados por el Observable.
- ◆ **Subscription:** Representa la ejecución de un Observable, es principalmente útil para cancelar la ejecución.
- ◆ **Operators:** Son funciones puras que permiten un estilo de “programación funcional” de tratar con colecciones con operaciones como mapa, filtro, concat, flatMap, etc.
- ◆ **Subjects:** Es el equivalente a un EventEmitter, y la única forma de multi-difundir un valor o evento a varios *observers*.

Streams

- ◆ Una de las máximas en la programación reactiva es que "todo es un stream" por lo que con RxJs, cualquier flujo de información será tratada como un stream. Eventos del ratón, Arrays, rangos de números, promesas, etc. Todo será un stream.
 - ◆ //Una letra es un stream
 - ◆ const letterStream\$ = Rx.Observable.of('A');
 - ◆ // Un rango de números es un stream
 - ◆ const rangeStream\$ = Rx.Observable.range(1,8);
 - ◆ // Los valores de un Array son un stream
 - ◆ const arrayStream\$ = Rx.Observable.from([1,2,3,4]);
 - ◆ // Valores emitidos cada 100 ms son un stream
 - ◆ const intervalStream\$ = Rx.Observable.interval(100);
 - ◆ // Cualquier tipo de evento del ratón es un stream
 - ◆ const clicksStream\$ = Rx.Observable.fromEvent(document, 'click');
 - ◆ // La respuesta a un servicio basada en una promesa es un stream
 - ◆ const promiseStream\$ = Rx.Observable.fromPromise(fetch('/products'));

Patrón Observer

- ◆ El patrón Observer juega un papel fundamental y explica a la perfección, el concepto de reactivo.
 - ◆ El patrón Observer define un productor de información, nuestro stream y que en RxJs está representado por una secuencia Observable o simplemente Observable y un consumidor de la misma, que sería el Observer.
 - ◆ Como hemos visto, en RxJs el Observable es nuestro stream y que nos sirve para prácticamente todo: eventos del ratón, rangos de números, promesas, etc.,

Patrón Observer

- ◆ Un stream es una secuencia de eventos ordenados en el tiempo. Puede emitir tres cosas diferentes: un valor (de algún tipo), un error y una señal de “completado”.

Con un ejemplo lo entendemos



JavaScript

```
var button = document.querySelector('button');
button.addEventListener('click', () =>
console.log('Clicked!'));
```



```
var button = document.querySelector('button');
Rx.Observable.fromEvent(button, 'click')
  .subscribe(() => console.log('Clicked RxJS!'));
```

la principal diferencia en nuestro ejemplo es que uno genera un evento directamente (event listener) que se ejecuta y el otro (RxJS) va a agregar un “ posible evento futuro ” en un “ array ” para luego poder “ subscribirse ” al mismo y poder ejecutarlo.

◆ Vamos a importar la librería:

◆ `import { interval } from 'rxjs';`

◆ Crearemos un observable que publique cada segundo

◆ `const secondsCounter = interval(1000);`

◆ Nos suscribimos a los cambios

```
secondsCounter.subscribe(n =>  
  this.time = `It's been ${n} seconds since  
  subscribing!`;  
  console.log(this.time);  
}
```

PRÁCTICA

Unidad 5

Manejo de Formularios

Formularios, Peticiones Post y Paginación



Formularios

- ◆ Angular >4 nos ofrece dos tipos de formularios
 - ◆ 1. template-driven forms
 - ◆ 2. Reactive forms.
- ◆ Template Drive Forms usa las directivas de Angular el template para manejar los formularios.
- ◆ Para activar template-driven forms debemos importar FormsModule en el módulo de nuestra aplicación

Formularios en Angular

- ◆ Angular ofrece un módulo que nos ayuda manipular fácilmente nuestros formularios
- ◆ Vamos a crear un nuevo módulo llamado FormulariosModule y dentro un componente Form1Component junto con un modelo Contacto con los atributos:
 - ◆ public name:string;
 - ◆ public email:string;
 - ◆ public phone:number;
 - ◆ public subject:string;
 - ◆ public msn:string;

Formularios en Angular

- ◆ Esta importación nos va a permitir usar el data binding en el formulario.
- ◆ El data binding nos va a permitir, que todo cambio que se produzca en los campos del formulario, se actualice automáticamente en la clase del modelo, y no tengamos que estar accediendo a las propiedades del DOM, para recoger estos cambios.
- ◆ La importación también nos va a permitir el manejo de los eventos del formulario.

Formularios en Angular

- ◆ Los formularios se enlazan a un objeto (modelo) y los cambios hechos a nivel del objeto se actualizarán en el template. Y los valores que introduce el usuario en el template se actualizan en el modelo
- ◆ El modelo debe ser una clase, no una interfaz
- ◆ El evento onSubmit será procesado por una función. Comunmente nombrada: onSubmit

Formularios

◆ Añadimos el formulario en el template:

- ◆ <form (ngSubmit)="onSubmit(contactForm);" #contactForm="ngForm">
- ◆ Establecemos el nombre del formulario y la acción ante el evento submit

- ◆ <input placeholder="Nombre" #nombre maxlength="80" [(ngModel)]="contacto.name" name="Name" required>
- ◆ Creamos los input y los vinculamos a las propiedades de nuestro objeto del modelo

- ◆ <button [disabled]="!contacto.form.valid">Enviar</button>
- ◆ Añadimos validación en el botón de submit

Formulario

- ◆<button type="button" (click)="limpiar()>Limpiar</button>
 - ◆Añadimos botón para simular el reset
 - ◆Añadimos la función limpiar para ejecutar el proceso reset
-
- ◆<div [hidden]="enviado">
 - ◆Englobamos el formulario en un div para mostrarlo o no
 - ◆Añadimos una variable booleana “enviado” para la lógica de ocultado

Formularios

- ◆ Editamos nuestro componente añadiendo 1 variable para detectar el formulario fue enviado y afectar el CSS mediante property binding
 - ◆ enviado = false;
- ◆ Añadimos un objeto tipo Post vacío como receptor de la entrada del formulario y lo inicializamos:
 - ◆ usuario: Usuario;
 - ◆ // en el constructor
 - ◆ this.usuario = new Usuario;

Formularios

- ◆ Añadimos 2 funciones para manejar el submit y el reset

```
onSubmit(f: NgForm) {  
  this.enviado = true;  
  console.log(this.usuario);  
}  
  
limpiar(f: NgForm) {  
  console.log("limpiando....");  
  f.resetForm();  
  this.enviado = false;  
}
```

Uso de eventos en formularios

- ◆ <div [hidden]="!enviado">
 - ◆ Añadimos presentación de los datos después del envío

- ◆ <h2>Datos enviados:</h2>
- ◆ <div >
- ◆ <div >Id:</div>
- ◆ <div >{{ usuario.userId }}</div>
- ◆ </div>
- ◆ <div > ... </div>
- ◆

- ◆ Añadimos un botón de “editar” que presentará nuevamente el formulario preservando los datos
- ◆ <button (click)="enviado=false">Editar</button>
- ◆ </div>

Ejercicio

- ◆ Añadir al servicio RemoteDataService una función que reciba un objeto Post y ejecute una petición http de tipo post cada vez que se envíe el formulario



Formularios Reactivos

Formularios Reactivos

- ◆ El doble enlace automático entre elementos *html* y propiedades de objetos fue el primer gran éxito de **Angular**.
- ◆ Ese *doble-binding* facilita mucho el desarrollo de formularios. Pero esa magia tienen un coste en escalabilidad; impacta en el tiempo de ejecución y además dificulta la validación y el mantenimiento de formularios complejos.
- ◆ La solución en Angular pasa por desacoplar el modelo y la vista, introduciendo una capa que gestione ese doble enlace.

Formularios Reactivos

- ◆ Los servicios y directivas del módulo `ReactiveFormsModuleModule` que viene en la librería `@angular/forms` permiten programar **formularios reactivos conducidos por el código**.

Clases para formularios reactivos

- ◆ Usaremos cuatro clases fundamentales para construir un form reactivo:

| Clase | Descripción |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AbstractControl | AbstractControl es la clase base abstracta para las tres clases de control de formulario concreto; FormControl , FormGroup y FormArray . Proporciona sus componentes y propiedades comunes. |
| FormControl | FormControl rastrea el valor y el estado de validez de un control de formulario individual. Corresponde a un control de formulario HTML como <input> o <select> . |
| FormGroup | FormGroup rastrea el valor y el estado de validez de un grupo de instancias de AbstractControl . Las propiedades del grupo incluyen sus controles secundarios. El formulario de nivel superior en su componente es un FormGroup . |
| FormArray | FormArray rastrea el valor y el estado de validez de una matriz indexada numéricamente de instancias de AbstractControl . |

Form Builder

- ◆ FormBuilder es un servicio del que han de depender los componentes que quieran desacoplar el modelo de la vista.
- ◆ Se usa para construir un formulario creando un FormGroup, (un grupo de controles) que realiza un seguimiento del valor y estado de cambio y validez de los datos.
- ◆ **Form Group**
 - ◆ El formulario se define como un grupo de controles. Cada control tendrá un nombre y una configuración. Esa definición permite establecer un valor inicial al control y asignarle validaciones.

Formularios Reactivos

Vamos a generar un nuevo componente y añadimos lo siguiente:

```
import { FormBuilder, FormGroup } from '@angular/forms';

public form: FormGroup;
public name:string = "Reactivos"
constructor(private formBuilder: FormBuilder) {}
public ngOnInit() {
    this.form = this.formBuilder.group({ });
}
```

Formularios Reactivos

- ◆ Vamos a añadir data de prueba

```
let cont: Contacto = new Contacto();  
cont.email = 'tutorial@angular.io';  
cont.name = this.name.toLowerCase();  
cont.msn = "mensajeria" ;  
cont.phone = 600123456;  
this.form = this.formBuilder.group(cont);
```

La vista

- ◆ Lo único necesario será asignar por nombre el elemento html con el control typescript que lo gestionará.
- ◆ Para ello usaremos dos directivas que vienen dentro del módulo *reactivo* son **[formGroup]="objetoFormulario"** para el formulario en su conjunto, y **FormControlName="nombreDelControl"** para cada control.

Formularios Reactivos

```
<form [formGroup]="form" (submit)="onSubmit(form.value)">
<label for="email">E-mail</label>
<input name="email"
formControlName="email"
type="email" />
<br>
<label for="name">Name</label>
<input name="name"
formControlName="name"
type="text" />
<br>
<label for="msn">Msn</label>
<input name="msn"
formControlName="msn"
type="text" />
<br>
<label for="phone">Phone</label>
<input name="phone"
formControlName="phone"
type="number" />
<br>
<button type="submit"
[disabled]="form.invalid">Guardar</button>
</form>
```

setValue() y patchValue()

- ◆ Anteriormente, creamos un control e inicializaba su valor al mismo tiempo.
- ◆ También se puede inicializar o restablecer los valores más adelante con los setValue() y patchValue() .

setValue()

- ◆ `setValue()` , asigna cada valor de control de formulario de una vez al pasar un objeto de datos cuyas propiedades coinciden exactamente con el modelo de formulario detrás de `FormGroup` .
- ◆ El método `setValue()` comprueba el objeto de datos a fondo antes de asignar ningún valor de control de formulario.

setValue()

- ◆ No aceptará un objeto de datos que no coincida con la estructura de FormGroup o que le falten valores para cualquier control en el grupo. De esta forma, puede devolver mensajes de error útiles si tiene un error tipográfico o si anotó los controles incorrectamente.

```
this.form.setValue({  
  name: this.name,  
  phone: this.phone[0] || new Phone()  
  ...  
}) ;
```

patchValue()

- ◆ **patchValue()** , puede asignar valores a controles específicos en un FormGroup suministrando un objeto de pares clave / valor para ellos.
- ◆ Este ejemplo establece solo el control del name del formulario.
- ◆ **patchValue()** no puede verificar los valores de control que faltan y no arroja errores útiles.
- ◆

patchValue()

```
this.heroForm.patchValue ( {  
  name: this.hero.name  
} ) ;
```

Agregando Validaciones

- ◆ Tenemos varias formas de hacer validaciones pero todas usan la clase FormControl

```
this.phone = new FormControl('', Validators.required);
this.email = new FormControl('', [ Validators.required,
Validators.email ]);
this.form = this.formBuilder.group({
  email: this.email,
  name: [
    '',
    Validators.required
  ],
  msn : '',
  phone: this.phone
});
```

Agregando Validaciones

- ◆ Podemos informar al usuario de los errores. Cada objeto FormControl tiene un array de los errores encontrados:

```
<div *ngIf="email.errors?.email">  
  Debe ser un email  
</div>
```

- ◆ PRÁCTICA: Explorar las diversas validaciones existentes y la forma de mostrar mensajes de error.



Gracias!

Alguna pregunta?

Pueden contactarnos mediante
info@impactotecnologico.net y apuntarse a algunos de
nuestros cursos



