

Entendiendo



Jorge Cano

Tabla de contenido

Capítulo 1 - Solo Angular

Introducción	1.1
Agradecimientos	1.2
Capítulo 1 - Solo Angular	1.3
1.1 Por qué Angular?	1.4
1.2 Conceptos de typescript	1.5

Capítulo 2 - Primeros Pasos

2 - Primeros pasos	2.1
2.1 Preparando el entorno	2.2
2.2 - Mi primera App	2.3
2.3 - Angular CLI	2.4
2.4 - Estructura de una Angular WebAPP	2.5
2.5 - Directivas	2.6

Capítulo 3 - Componentes

3 - Componentes	3.1
3.1 - Funciones	3.2
3.2 Input	3.3
3.3 Output	3.4
3.4 Constructor	3.5

Capítulo 4 - Compilación

4 - JIT vs AOT	4.1
----------------	-----

Capítulo 5 - NgModules

Tabla de contenido

Capítulo 1 - Solo Angular

Introducción	1.1
Agradecimientos	1.2
Capítulo 1 - Solo Angular	1.3
1.1 Por qué Angular?	1.4
1.2 Conceptos de typescript	1.5

Capítulo 2 - Primeros Pasos

2 - Primeros pasos	2.1
2.1 Preparando el entorno	2.2
2.2 - Mi primera App	2.3
2.3 - Angular CLI	2.4
2.4 - Estructura de una Angular WebAPP	2.5
2.5 - Directivas	2.6

Capítulo 3 - Componentes

3 - Componentes	3.1
3.1 - Funciones	3.2
3.2 Input	3.3
3.3 Output	3.4
3.4 Constructor	3.5

Capítulo 4 - Compilación

4 - JIT vs AOT	4.1
----------------	-----

Capítulo 5 - NgModules

5 - NgModule	5.1
--------------	-----

Capítulo 6 - Directivas

6 - Directivas	6.1
----------------	-----

Capítulo 7 - Componentes Avanzados

7 - Componentes Avanzados	7.1
---------------------------	-----

Capítulo 8 - Router

8 - Router	8.1
------------	-----

Capítulo 9 - Inyección de Dependencias

9 - Inyección de Dependencias	9.1
-------------------------------	-----

Capítulo 10 - LifeCycle

10 - LifeCycle	10.1
----------------	------

Capítulo 11 - Pipes

11 - Pipes	11.1
------------	------

Capitulo 12 - HTTP

12 - HTTP	12.1
-----------	------

Capítulo 13 - UI AngularMaterial V PrimeNG

13.1 - AngularMaterial	13.1
------------------------	------

13.2 - PrimeNG	13.2
----------------	------

Final

final	14.1
-------	------

Entendiendo Angular

Acerca del autor (@jorgeucano)

Después de más de 9 años desarrollando, Jorge es un full stack developer avocado a las tecnologías en javascript. Hoy en día Jorge es Director de “departamento de nuevas tecnologías” en el cual prueba y decide qué tecnologías utilizar para brindar el mejor servicio performance y seguridad con Angular / Firebase y otras tecnologías de punta... Profesor de varios cursos relacionados a javascript, Google Developer Expert en tecnologías web y Angular Developer Expert , speaker internacional, y escritor de artículos técnicos.

¿Sobre qué trata el libro?

En este libro vamos a pasar por las funcionalidades de Angular, para poder ir entendiendo todo lo que se puede hacer con este fantástico framework y entender un poco mas a fondo que utilizamos.

Todo lo referente a código lo veremos sobre typescript, debido a que el equipo que creo el framework lo hizo sobre typescript, y respetaremos su decisión sobre el mismo (mas allá que se puede trabajar con JS o Dart) .

¿Qué necesito para seguir los ejemplos?

Si vas a descargarte los ejemplos vas a necesitar una cuenta de GITHUB (para poder hacer fork) o simplemente descargarlo de la web.

Un browser (de preferencia Chrome).

En tu computadora necesitas tener instalado nodeJS (<https://nodejs.org/es/>), ya que lo vamos a utilizar.

Por supuesto un IDE o editor de texto de tu preferencia, yo voy a utilizar Visual Studio Code (<https://code.visualstudio.com/>), pero esto es a preferencia (si no tienes ninguno como preferido o te gusta probar alternativas, el VSC esta bueno para programar con Angular).

Aclaración: Todo lo que vas a ver en los ejemplos podes ejecutarlos en cualquier terminal, de cualquier sistema operativo que soporte NodeJS (esta salvedad es por ejemplo ChromeOS no soporta NodeJS por lo tanto no se puede ejecutar el proyecto desde el CLI)

Aclaración 2: Cuando en el libro se mencione "Angular" se refiere a la version 2.x en adelante, cuando se mencione "AngularJS" se refiere a la version 1.x .

¿Por qué este libro?

Hay muchos motivos para escribir un libro, tal vez, el mio era que realmente tenia ganas de escribir un libro, sentir que podes colaborar con quienes quieran aprender sobre Angular, y la segunda variable era que "no hay mucho en español", entonces porque no empezar a crear mas en español... con esto no quiero decir que no exista nada... digo que no hay MUCHO, y es algo casi obvio, la mayoría de las personas lo escriben en ingles, ya que es mas rentable o el mercado es mas grande.. yo simplemente quiero acercar a las personas a Angular y al desarrollo con este gran framework.

No quiero hacerme rico con un libro que se vende a 5 o 10 dolares, esta claro... el precio es un simple incentivo al tiempo empleado al escribir y realizar los ejemplos... y para pagar los gastos de publicación (si leanpub te cobra unos cuantos dolares por publicar el libro)...

El precio del libro esta en dolares ya que la plataforma es en dolares ... pero desde paypal podrán pagar con su tarjeta de credito o debito sin ningun problema (mas alla del cambio).

Por último más allá del desafío personal, si este libro tiene buenas criticas, voy a ir por mas, asi que les pido que hagan criticas constructivas para poder mejorar este libro y los que vienen también sean buenos.

¿Dónde encuentro los ejemplos del libro?

<https://github.com/jorgeucano/entendiendo-angular>

¿Tiene un bug? ¿Hay algo que no entiendes?

Simplemente levanta un issue en el mismo repositorio.

Agradecimientos

A todos aquellos que se tomen el tiempo de leer este libro... realmente espero que les guste y les sea util...

A el gran equipo de Angular por hacer un framework tan completo y funciona.

A la comunidad, que gracias a ella voy creciendo todos los días.

A mi novia, que me banca en todos los viajes, charlas, horas programando y me tendra que bancar en todo lo que se viene =D..

A mi familia :)

Y a todo quien me dedico una buena vibra para este libro...

Solo Angular

Cuando salió AngularJS allí por el 2009, era normal ponerle el "JS" en el nombre del framework ya que estaban 100% escritos en JavaScript...

Pero este nombre termino siendo un problema, ya que cuando empezó a crearse "Angular 2" la nueva version de este super poderoso framework, se empezó a escribir en AtScript (<https://en.wikipedia.org/wiki/AtScript>), pero Microsoft (si aquella evil corp que cambio) creo TypeScript (<https://www.typescriptlang.org/>) y por el tipado y su robustez el Angular's Team lo adopto y decidió reescribir todo lo que se había hecho en AtScript a TypeScript...

Por este motivo se tachó el JS del nombre... entonces quedó de AngularJS a Angular 2 ... Pero la historia no queda ahí, después de tanto esperar esta nueva versión que cambio muchísimo de la versión anterior, se tomó otra gran decisión.

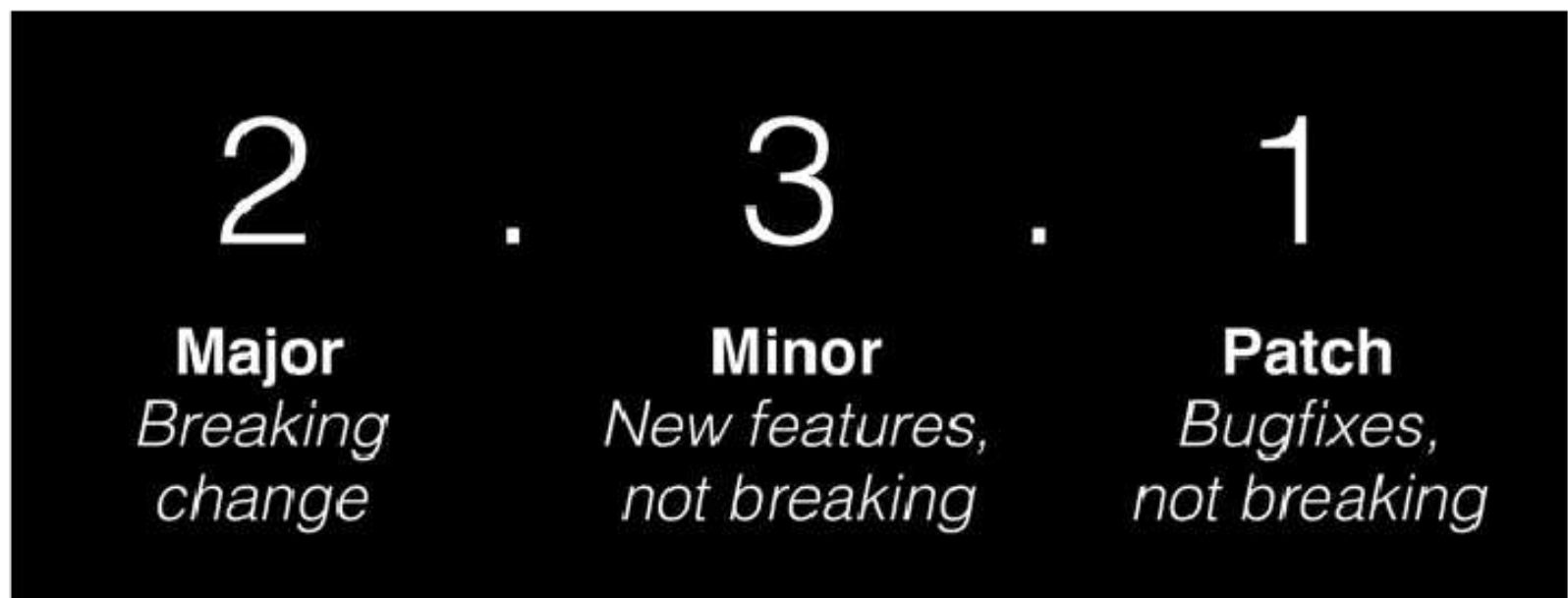
A partir de ahora se van a sacar versiones consecutivas utilizando el SEMVER, por lo tanto cada 6 meses vamos a tener una versión de Angular nueva ... WTF entonces para qué aprendo Angular 2 si Angular 4 está por salir en unos meses, y nuevamente empezamos con un nuevo problema en el nombre...

Igor Minar (<https://twitter.com/IgorMinar>) , uno de los core team de Angular se encargó en explicar exactamente que es esto de las versiones y porque no deberiamos preocuparnos, si no, que contentarnos ...

Pero qué es Semantic Versioning (SEMVER) ?

SEMREV es "Agrega contexto a los números de las versiones". Esto permite a los desarrolladores no sólo razonar sobre cualquier actualización que hacemos, pero incluso podemos dejar que herramientas como NPM lo hagan de manera automática y segura para nosotros.

Una versión semántica consta de tres números:



Siempre que corrijan un error y lo liberen, aumenta el último número, si se agregan una nueva función, aumentan el segundo número y cada vez que liberan un cambio de fuerte aumentan el primer número.

Entonces, ¿qué significa esto?

Software evolutivo, los “breaking changes” se producirán en algún momento. Por ejemplo, al dar un error de compilador para los errores de aplicación existentes que pasaron desapercibidos con la versión del compilador anterior, cualquier cosa, que rompa una aplicación existente al actualizar Angular, requiere que el equipo cambie el número de versión principal.

Sólo para ser claro, como Igor también mencionó en su charla. En este momento, incluso actualizar la dependencia de TypeScript en Angular de v1.8 a v2.1 o v2.2 y compilar Angular con ella, técnicamente causaría un “breaking changes”. Así que están tomando a SEMVER muy, muy seriamente.

Entendiendo esta decisión parecería una locura... cambios constantes... pero en realidad son cambios evolutivos y con retro-compatibilidad... por lo tanto aprender Angular ahora se convierte en algo progresivo (nodeJS hace algo parecido).

Por lo tanto al día de la fecha tenemos una idea de que las nuevas versiones vendrían en poco tiempo

Predictable, Transparent & Incremental Evolution	
Version 4	March 2017
Version 5	September/October 2017
Version 6	March 2018
Version 7	September/October 2018
(tentative schedule)	

Aquí se puede ver el video en donde explica todo Igor

https://youtu.be/aJIMoLgqU_o

Entonces cuando estés aprendiendo#Angular, vas a estar aprendiendo algo en constante evolución...

Tienes grandes pros, el framework evoluciona constantemente tal y como lo hacen los navegadores y funcionalidades nuevas... por lo tanto si typescript saca una nueva versión la van a incluir rápidamente, si existe una nueva versión de RxJS pasaría lo mismo, las inclusiones son rápidas y ágiles, y gracias a estoAngular pasa a estar siempre actualizado ,o mejor dicho con actualizaciones "constantes o más rápidas", y de esta forma no estaría mucho tiempo sin una funcionalidad que aparezca en los navegadores.

Algo a tener muy en cuenta es que vamos a tener retro-compatibilidad, por lo tanto actualizar la versión no es botar todo el código y rehacerlo sino que simplemente es tener más funcionalidades para poder aprovechar en nuestros desarrollos.

¿Por qué Angular?

Hay muchos Frameworks y librerías para el frontend, hoy por hoy, uno podría elegir uno distinto para cada proyecto. Ultimamente ReactJS tomo mucha fuerza y aprecio VueJS también, pero desde el 2009 AngularJS fue una gran decisión para muchos (me incluyo) e hicimos muchísimos proyectos! Realmente AngularJS en su momento fue un visionario y creo las directivas, algo parecido a los componentes de hoy en día.

El equipo de Angular aprendió muchísimo en todos estos años, y decidió darle una vuelta de rosca (es una frase muy Argentina, significa: "querer hacer un cambio mas para mejor") y empezar a escribir Angular 2 , que luego se convertiría en solo Angular.

Hablando un poco mas a nivel arquitectura, el framework esta muy avanzado, y realmente es demasiado profesional !

¿Qué quiero decir con esto?

Angular (2 en adelante) esta pensando para poder hacer tanto pequeñas aplicaciones, como aplicaciones muy grandes, y a su vez poder mantener un orden fuerte, tiene grandes integraciones para los componentes (los veremos en capítulos siguientes) y como se maneja todo de una forma muy organizada entre si.

Para los que vienen de lenguajes tipados orientados a objetos, Angular va a ser la desmitificación del frontend, realmente es demasiado fácil de migrar (es hora que vengan al lado oscuro del frontend y dejen java atras).

El core de angular es nuevo y moderno, esta hecho sobre las bases de los nuevos estandartes de las webs modernas y escrito por veteranos del frontend, que ya trabajaron y aprendieron mucho de hacer un framework.

En angular 1, teníamos que entender el modelo de MVC (modelo vista controlador) y como conectarlos, ahora en Angular simplemente tendremos que construir todo en base a componentes, que se puede comunicar entre si, pero solamente escribiremos componentes.

Pensemos en una casa.. una casa es un conjunto de ladrillos (en perspectiva, esta claro que lleva mas cosas) , ahora pensemos en Angular como la casa, y que los ladrillos son componentes, esa seria la idea, nuestra WebApp va a ser un conjunto de componentes con algunas cosas mas!

El desarrollador se tiene que enfocar en crear componentes.

TypeScript

Realmente veo a typescript como uno de los puntos de porque Angular, tal y como leíste un par de paginas antes, Angular se empezo a escribir en AtScript y luego migraron a TypeScript.

TypeScript es un super set de ES6/7 , lo que quiere decir es que tiene esas funcionalidades y mas... entre otras el tipado (es opcional) pero esto nos da un montón de beneficios a la hora de trabajar con grandes equipos.

Un orden en el código es importante, tal y como comentar el código o hacer test.

Hoy en día una gran cantidad de equipos no se encuentran en la misma sala para poder trabajar codo a codo, si no que lo hacen desde sus casas o desde oficinas a miles de kilometros de distancia,

Gracias a typescript y su buenas practicas, vamos a poder hacer muchas cosas que van a mejorar esto... olvidarnos de los NaN (not a number) , un poco mas adelante vamos a ver los conceptos de typescript.

Si creen que no necesitan eso, realmente los invito a probarlo (el libro se escribió con typescript) ya que da grandes beneficios, sumado a que los IDE's y Editores de texto, vienen con addons para eso.

Performance y Mobile

El equipo de Angular se dedico a tener gente experta para poder hacer que la aplicación sea mas performante y además que el trabajo en mobile sea rápido y fácil de hacer... Los dispositivos actuales tienen menos limitaciones que los de hace años, pero las conexiones no siempre son las mejores, entonces al pensar en funcionalidades, velocidad y tiempos de carga... se dedicaron a hacer cosas exclusivas para mobile.

Angular, como muchos frameworks/librerias modernas, mejoran la performance gracias a los **WebWorkers**, y si sabemos utilizarlo bien, esto va a dar una experiencia al usuario de que nuestra app es rápida y fluida.

También tenemos que mencionar a **IONIC** (en su version 2) que funciona 100% en Angular (2 en adelante) para hacer aplicaciones híbridas. Y a **NativeScript** que funciona con Angular también (pueden programar solo con JS también) y que compila a nativo (si a nativo dije).

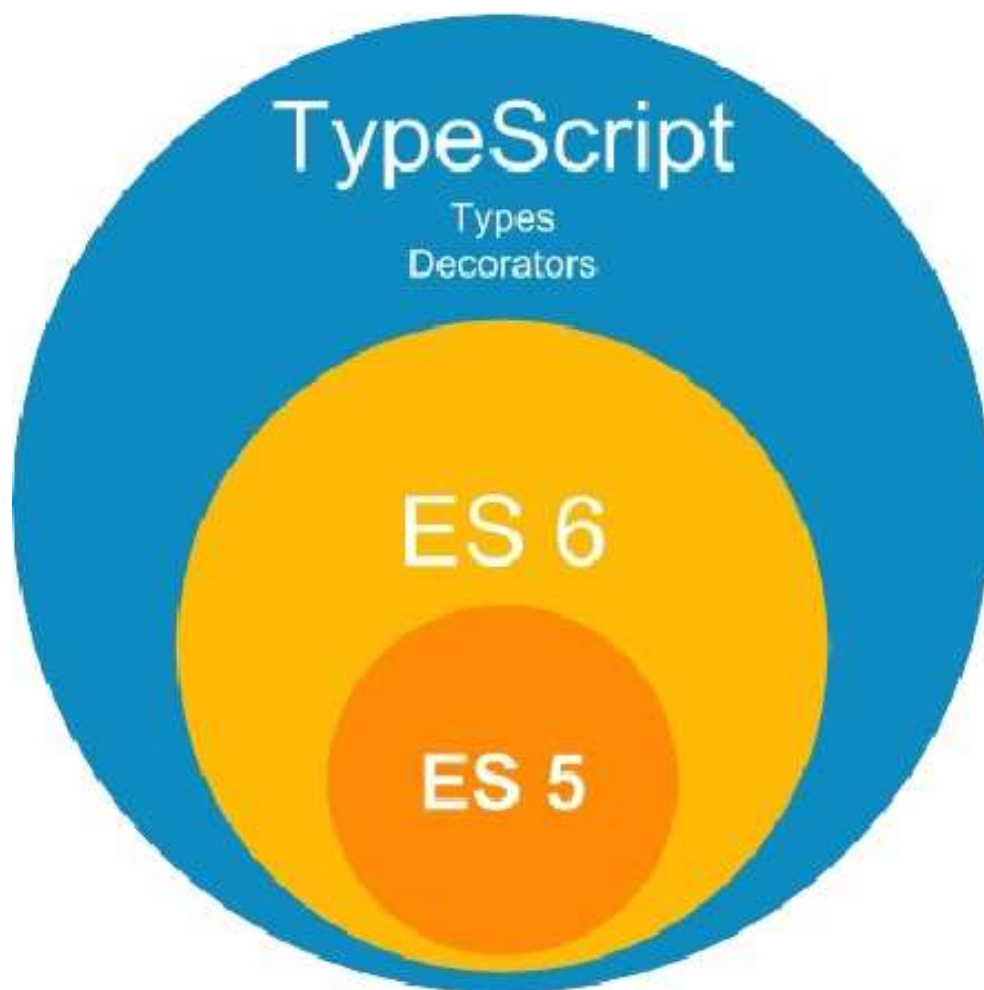
Por ultimo, me gustaria mencionar, que no es solo un "hago todo de otra forma" si no que Angular viene con muchisimas mejoras ademas del typescript mobile y componentes.

1.1 Por qué Angular?

Si no que viene con funcionalidades como "form builder", "routing", "annotations", "observables", "shadow dom" entre otras.. por lo tanto hay cosas nuevas por aprender y aprovechar.

Conceptos de TypeScript

Tal y como comente anteriormente, TypeScript es un SuperSet de ES6, ahora ES6 es un SuperSet de ES5... que quiere decir esto..



TypeScript is a javascript superset

Tal y como vemos en la imagen, TypeScript, contiene todo lo que contiene ES6 y además agrega types y decorators.

Pero bueno vayamos un poco mas allá con TypeScript, osea vamos al código, bueno primero a la consola.

```
npm install -g typescript
```

Aclaración: todos los ejemplos se encuentran en el github del libro.

Los ejemplos del código que vamos a ver en esta sección estan en la carpeta **"entendiendo-angular-typescript"**

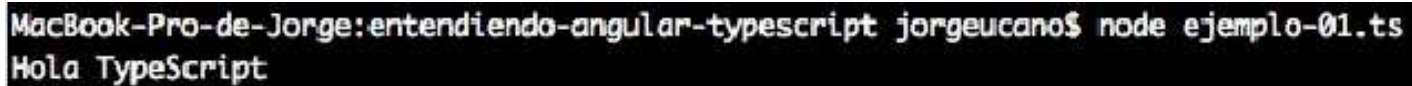
ejemplo-01.ts

1.2 Conceptos de typescript

```
console.log("Hola TypeScript");
```

Ahora podemos ejecutar en consola nuestro ejemplo, simplemente poniendo:

```
node ejemplo-01.ts
```



```
MacBook-Pro-de-Jorge:entendiendo-angular-typescript jorgeucano$ node ejemplo-01.ts
Hola TypeScript
```

Exactamente, node lo ejecuto directo y podemos ver que ahora imprimió nuestro "console.log", pero a nosotros nos importa que funcione en el navegador...

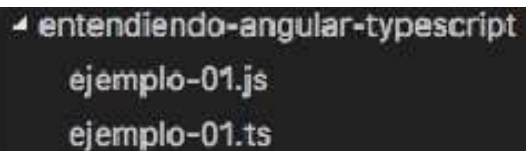
¿por lo tanto deberíamos tener un HTML que llame a nuestro TypeScript?

La respuesta es un rotundo **NO**, TypeScript necesita ser **TRANSPILADO**, osea que no es compilado, si no que el comando que instalamos por node, va a pasar de typescript a ES5, a menos que le indiquemos que pase a ES6 (por ahora lo mejor que podemos hacer es pasar a ES5, ya que ES6 no esta soportado por todos los navegadores).

Para poder transpilar lo que vamos a hacer es:

```
tsc ejemplo-01.ts
```

Vamos a ver que la consola no devuelve nada, pero si vemos nuestro "navegador de carpetas" (que raro que suena), vamos a ver que se genero un JS , que se llama exactamente igual al TS que teníamos.



```
└─ entendiendo-angular-typescript
   ├── ejemplo-01.js
   └── ejemplo-01.ts
```

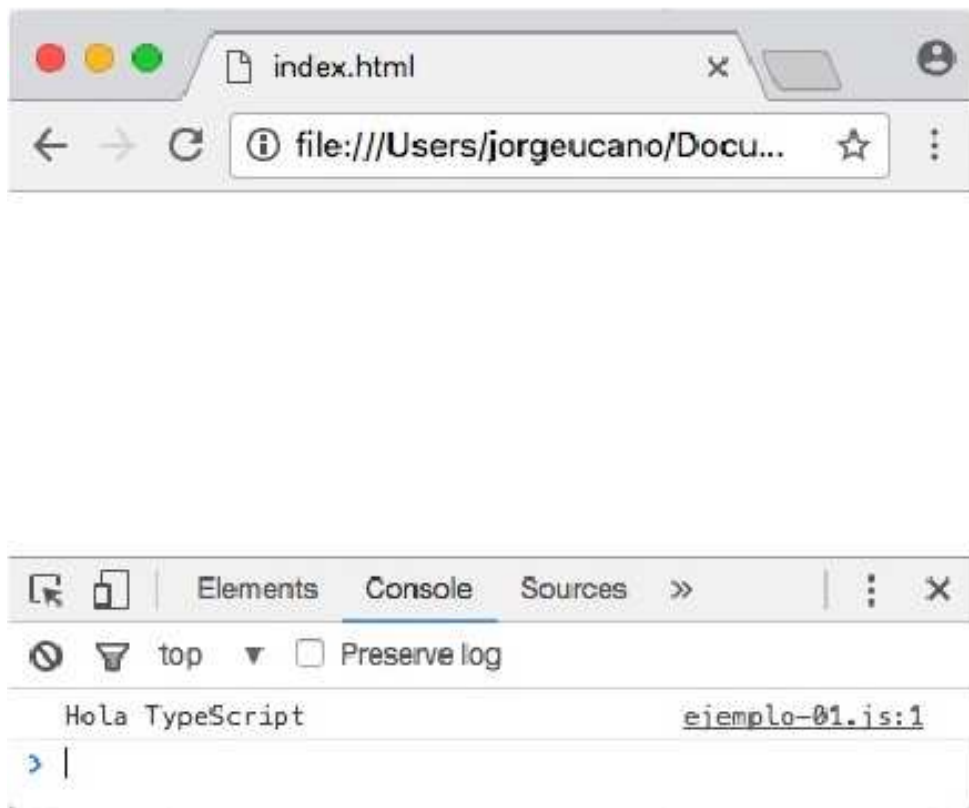
El código que genero es:

```
console.log("Hola TypeScript");
```

Si , bueno el "console.log" es igual para ambos, pero lo importante en este ejemplo súper básico es que vamos a poder utilizarlo en un html... vamos a crear algo básico:

index.html

```
<script src="ejemplo-01.js"></script>
```

Sin errores, venimos bien hasta ahora ;)

Vamos a ver algunas bondades de TypeScript

Veamos un ejemplo en JS y TS para ver cuales son los resultados, el ejemplo es simplemente llamar a una funcion que deberia recibir 2 numeros para sumarlos y luego mostrarlos por consola.

JS

```
function add (a,b){  
  console.log( a + b );  
}  
add (1,3);  
add (1, '3');
```

Resultado (ejecutado desde la consola de chrome directo):

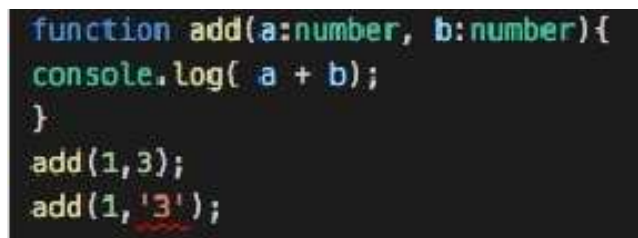
```
> function add (a,b){  
  console.log( a + b );  
}  
add (1,3);  
add (1, '3');  
4  
13
```

Ahora veamos el mismo ejemplo pero en TS (este va a tener que transpilarse, por lo tanto lo vamos a agregar a *ejemplo-02.ts*):

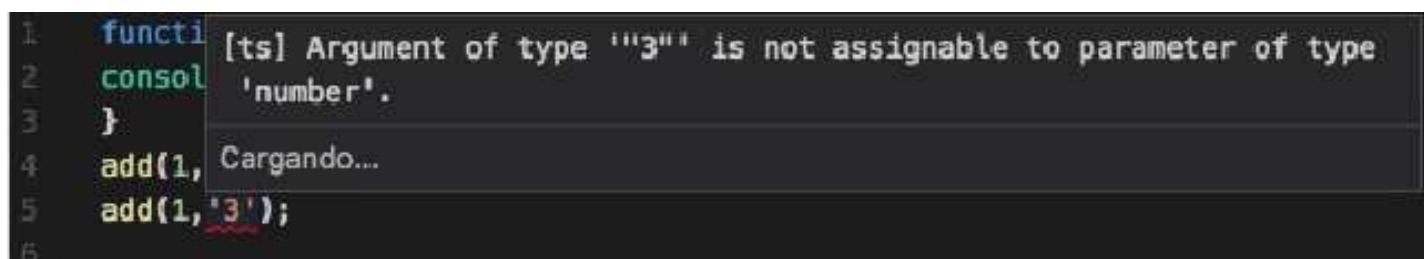
1.2 Conceptos de typescript

```
function add(a:number, b:number){
    console.log( a + b);
}
add(1,3);
add(1,'3');
```

Al escribir este ejemplo en el VSC, me aparece lo siguiente:



Y cuando paro el cursor sobre el error:



Ya nos indica que un tipo "String" no puede asignarse a un tipo number.. claro un string no es un numerico... pero si JS me dejo hacerlo sin problemas, claro porque no tiene types...

Pero igual vamos a ejecutar el tsc para ver que pasa:

```
tsc ejemplo-02.ts
```



Claramente el VSC no se equivoco, no se puede asignar un string a un tipo number...

A pesar de esto, el transpilador nos creo el JS, debido a que este ERROR, es casi como un warning en realidad... debito a que JS permite hacer esto (pero deberiamos arreglarlo igual)...

Vamos un poco a la explicación mas tecnica...

TypeScript al tener types nos deja asignar un tipo a una variable, como vemos en el ejemplo anterior, le dijimos a nuestras variables de la funcion, que son del tipo number, por lo tanto, espera que lleguen dos numeros, no un numero y un string...

En TypeScript, se pone **nombreDeVariable : tipo = asignacionDeValor**

```
var miVariableNumerica:number = 10;
```

1.2 Conceptos de typescript

No es necesario que este inicializado con un valor.

Pero veamos los tipos que maneja typescript.

- string
- number
- boolean
- Array
- Object
- undefined
- Enum
- any
- void

```
var name: string = "Jorge";
var age: number = 27;
var live: boolean = true;
var skills: Array<string> = ['Developer', 'Architect web', 'Teacher'];
var skills: string[] = ['Developer', 'Architect web', 'Teacher'];
var skillsNumers: Array<number>=[1,2,3,4,5];
var skillsNumers: number[]=[1,2,3,4,5];
enum Role {Employee, Mananger, Admin};
var role: Role = Role.Employee;
var something : any = 'now as string';

function setName(name: string):void{
  this.name = name;
}
```

Mas alla de los tipos para las variables, podemos tipar la devolucion de las funciones... en el ejemplo de la imagen, vemos que esta tipado con void, osea que no devuelve nada... pero podriamos ponerle cualquiera de las enumeradas arriba y en el caso de no devolver el tipo que estamos esperando nos va a dar un error. Ademas si la funcion tiene una devolución, la variable que reciba esa devolución va a tener que ser del mismo tipo.

Features:

- Types
- Interfaces
- Shapes
- Decorators

Arrow function:

1.2 Conceptos de typescript

```
var data: string[] = ['Bob Klose', 'Roger Waters', 'Nick Mason', 'Rick Wright'];
data.forEach((artist)=>
  console.log(artist);
);
```

Arrow function, hace que no perdamos el scope de lugar.. por lo tanto el "this" nos va a funcionar en todos lados (no sabes que es "this", no te hagas problema, ya lo vamos a ver).

La estructura de un arrow function es:

```
() => {}
```

Las parentesis estan vacias a menos que reciba algun tipo de parametro, y luego lo que esta entre las llaves es lo que se ve a ejecutar en la funcion.

Multi-Line String:

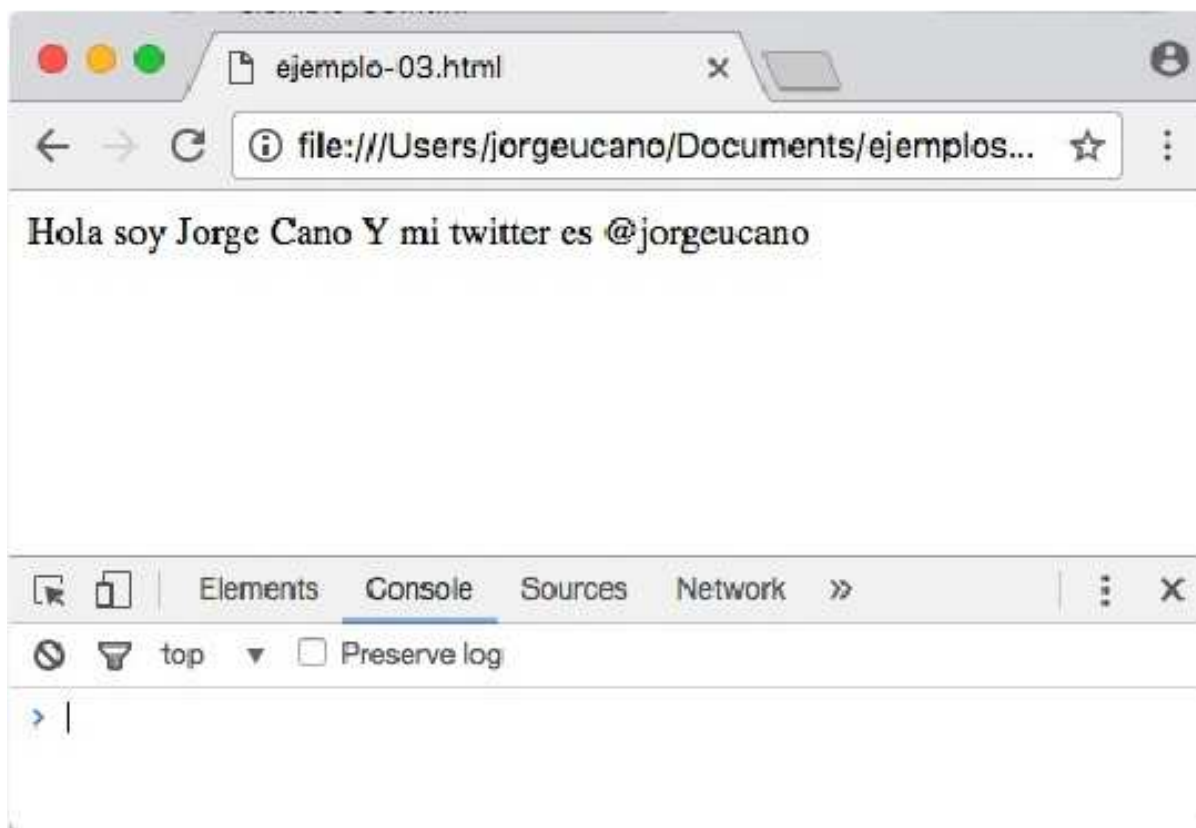
```
var first_name: string= "Jorge";
var last_name: string = "Cano";
var twitter_user: string = "@jorgeucano";
var interpolation = `
  <div>
    <p>
      Hola soy <span> ${first_name} ${last_name} </span>
      Y mi twitter es ${twitter_user}
    </p>
  </div>
`
```

;

WOW really ??? yes :D !!!

El acento frances ` genera string multilineas... y ademas tenemos interpolation, podemos mezclar string con variables, con \${nombre_variable}

Podemos chequear el ejemplo, mira el archivo _ejemplo-03.ts , ejemplo-03.js y ejemplo-03.html, _el resultado de todo esto es:



Veamos que genero el el transpilador para poder hacer esto (*ejemplo-03.js*):

```
var first_name = "Jorge";
var last_name = "Cano";
var twitter_user = "@jorgeucano";
var interpolation = "\n    <div>\n        <p>\n            Hola soy <span> " + first_n
ame + " " + last_name + " </span>\n            Y mi twitter es " + twitter_user + "\n
        </p>\n    </div>\n";
document.getElementById('result').innerHTML = interpolation;
```

Claramente prefiero TS para esto jeje...

Clases

```
class Foo { foo:number; }
class Bar { bar: string; }

class Baz{
    constructor (foo:Foo, bar:Bar){}
}

let baz = new Baz(new Foo(), new Bar());
```

Veamos un poco el ejemplo:

Foo y Bar no tienen ningun tipo de inicializador.. solo indican que tiene una variable cada una y su tipo.

Ahora **Baz** tiene un constructor (para los que nunca vieron un constructor, es una función que se ejecuta con la creación del objeto, por lo tanto cuando lo creamos tenemos que pasarle los parametros del contructor que se ejecuta, el constructor esta en todas las clases por defecto esta vacio, por lo tanto no es necesario escribirlo si va a estar vacio).

```
class Person{
    first_name:string;
    last_name:string;
    twitter_user?:string;
}

var persona = new Persona();
persona.first_name = "Jorge";
persona.last_name = "Cano";
persona.twitter_user = "@jorgeucano";
```

Las variables con el "?" antes del ":" para indicar el tipo, es que son opcionales.

En este caos, *first_name* y *last_name* son obligatorios para la clase, pero *twitter_user* es opcional.

Interfaces

Una interfaz es como un "contrato" para dentificar como es algo..

```
interface unaPersona{
    first_name:string;
    last_name:string;
    twitter_user?:string;
}

let a : unaPersona = {
    first_name = "Jorge",
    last_name = "Cano",
    twitter_user = "@jorgeucano"
}
```

Comparado la interfaz unaPersona contra la clase Person, veremos que ya de por si la forma de Crearlos son distintos.

Uno es mas como una "especificación" de como tiene que ser y el otro es un objeto directo.

Shapes

Por debajo de TypeScript esta JavaScript, y por debajo de JavaScript esta normalmente un JIT (Just-In-Time compilador). Dada la semántica subyacente de JavaScript, los tipos suelen ser razonados por "formas". Estas formas subyacentes funcionan como interfaces de TypeScript y, de hecho, son como TypeScript compara tipos personalizados como clases e interfaces.

```
interface unaPersona{
  first_name:string;
  last_name:string;
  twitter_user?:string;
}

let a : unaPersona = {
  first_name = "Jorge",
  last_name = "Cano",
  twitter_user = "@jorgeucano"
}

class NotAnAction {
  type: string;
  constructor() {
    this.type = 'Constructor function (class)';
  }
}

class Person{
  first_name:string;
  last_name:string;
  twitter_user?:string;
  constructor(){
    this.first_name = "Jorge";
    this.last_name = "Cano";
    this.twitter_user = "@jorgeucano";
  }
}

a = new Person(); // valid TypeScript!
```

A pesar de que unaPersona y Person tienen diferentes identificadores, tsc nos permite asignar una instancia de Person a una que tiene un tipo de unaPersona.

Esto se debe a que TypeScript sólo se preocupa de que los objetos tengan la misma forma. En otras palabras, si dos objetos tienen los mismos atributos, con las mismas tipologías, estos dos objetos se consideran del mismo tipo.

Diferencias entre VAR y LET

1.2 Conceptos de typescript

Varios ya me hicieron la pregunta de porque a veces uso VAR o LET en el código, por lo tanto queria dar a entender para que sirve cada uno...

Lo primero que vamos a hacer es ejecutar los siguiente en nuestra consola del navegador:

```
var a = 1;
let b = 1;
while ( a < 10 ){
    var a = a + 1;
    let b = a;
}
console.log(a);
console.log(b);
```

```
> var a = 1;
   let b = 1;
   while ( a < 10 ){
       var a = a + 1;
       let b = a;
   }
   console.log(a);
   console.log(b);

10
1
```

Claramente algo esta pasando, si 'b = a;' y ' a = 10' porque 'b = 1'.

Aquí esta la diferencia, var es una variable "global" frente a let que es una variable de "scope", cuando utilizamos let, esta variable va a funcionar sobre el bloque en el que se encuentra, por lo tanto la variable "b" en el while NO es la misma que la variable "b" que esta por fuera, por eso mismo, "b" dentro del while en la ultima iteración vale 10, pero por fuera de la iteración vale 1.

PARA, no todo es tns archivo por archivo!

TypeScript, tiene un config para configurarlo a nuestro gusto:


```
{
  "compilerOptions":{
    "module":"commonjs",
    "target":"es5",
    "emitDecoratorMetadata":true,
    "experimentalDecorators":true,
    "noImplicitAny":false,
    "removeComments":false,
    "sourceMap":true
  },
  "exclude":{
    "node_modules",
    "dist/"
  }
}
```

Ya tenemos todos los conceptos basicos que tiene typescript para que podamos trabajar con Angular, si te quedo algo en el tintero, puedes hacerme una pregunta, o chequear en su web, todas las funcionalidades.

Primeros pasos

Angular es un framework con muchísimas funcionalidades pero para llegar a un destino siempre tenemos el mismo inicio... Dar el primer paso, por lo tanto en esta unidad nos vamos a enfocar en lo que vamos a necesitar para dar nuestros primeros pasos y vamos a hacer una aplicación muy chiquita (casi un hello word) en distintas formas.

Angular no es mas que un arbol de componentes

Por lo tanto vamos a tener que entender un poco mejor que es un componente, y realmente creo que la primer aproximación es la siguiente:

Pensemos en el TAG `<a>` este tag que tiene HTML se puede "crear" o "utilizar" de varias formas, por ejemplo:

```
<a href="#">Mi Link</a>
```

```
<a href="#" class="miClassA" style="padding:10px" onClick="ejecutar()"> Soy un link</a>
```

En estos dos TAG's "a" que utilizamos le dimos distintos atributos para que dependiendo lo que necesitemos tenga una determinado texto, funcionalidad y diseño. Por lo tanto nosotros le dijimos a el navegador, como se va a **ver**, como va a **funcionar** y que va a **mostrar** cuando lo dibuje, esto quiere decir, que le estamos "enseñando" al navegador como queremos que sea.

Este TAG "a" en realidad lo podriamos ver como un **componente** del HTML, que recibe una serie de INPUTS con los cuales se va a dibujar, y este Componente es oficial de HTML.

Esto esta genial para una landing-page que no necesitamos mas que los tags que existen en HTML para hacerla... Pero si pensamos en una **WEBAPP** tenemos que pensar que vamos a necesitar mucho mas funcionamiento que el que posee una landing-page, y que posiblemente ciertos comportamientos se repitan.

Entonces imaginemos que dentro de nuestra WEBAPP tenemos un boton para pagar, pero ese boton en realidad se replica en varias pantallas/paginas, lo que haríamos normalmente (si fuese una app solo de js) tendríamos una funcion que hiciera el pago obteniendo los datos de un determinado formulario o inputs.

Pero con este modelo, podemos crear un componente que tenga la visual que necesitamos, el funcionamiento que necesitamos y que lo podamos utilizar con los parametros que necesitamos. WOW dije muchas veces necesitamos pero mejor vamos a un ejemplo un poco mas demostrativo (el código):

```
<pagar-button  
  monto="5"  
  articulo="libro"  
  procedente="twitter"  
  texto="Click aquí para pagar su libro"  
></pagar-button>
```

En este componente (o la vista HTML del mismo) estamos viendo que tenemos un "TAG" que no conocemos, mejor dicho, que no está en los TAGS creados por HTML en ninguna de sus versiones.

Pero a simple deducción, es un "boton" para pagar, que tiene un precio de "5", a su vez es un articulo del tipo "libro" tiene un procedente que es "twitter" y además tiene un texto, que por lo que parece sería el texto del boton (si es un poco largo, a veces en los ejemplos tenemos que exagerar un poco).

Entonces sin mucho más que un simple "HTML" parecería que acá estaríamos pagando un libro.

Pero si nosotros ponemos esto es nuestro HTML, el engine del navegador, vamos a suponer V8 de chrome, no va a entender que está pasando ahí y simplemente lo va a poner como texto plano.

Pero aquí aparece Angular, quien como mencione un poco más arriba es un "Arbol de componentes" por lo tanto Angular nos va a dar todas las herramientas para que nuestra WEBAPP le enseñe al navegador cada uno de los nuevos "TAGS" o mejor dicho a partir de ahora y para el resto del libro COMPONENTE que vamos creando y conectando a nuestras funcionalidades.

Volviendo a la explicación de componentes, el componente es un "TAG" de HTML que funciona mediante determinados "Inputs" y devuelve determinados "Outputs" mediante los cuales se contacta con otros sectores del sistema (WEBAPP), los cuales pueden tener estilos personalizados, y pueden ser utilizados por toda la aplicación sin la necesidad de volver a crearlos completos.

Preparando el entorno

Para la fecha en la cual estoy escribiendo esta parte del libro las versiones que utilizo son las siguientes:

NodeJS => v6.2.0

NPM => 3.10.6

Angular CLI => 1.0.0-beta.21

Consola => lterm

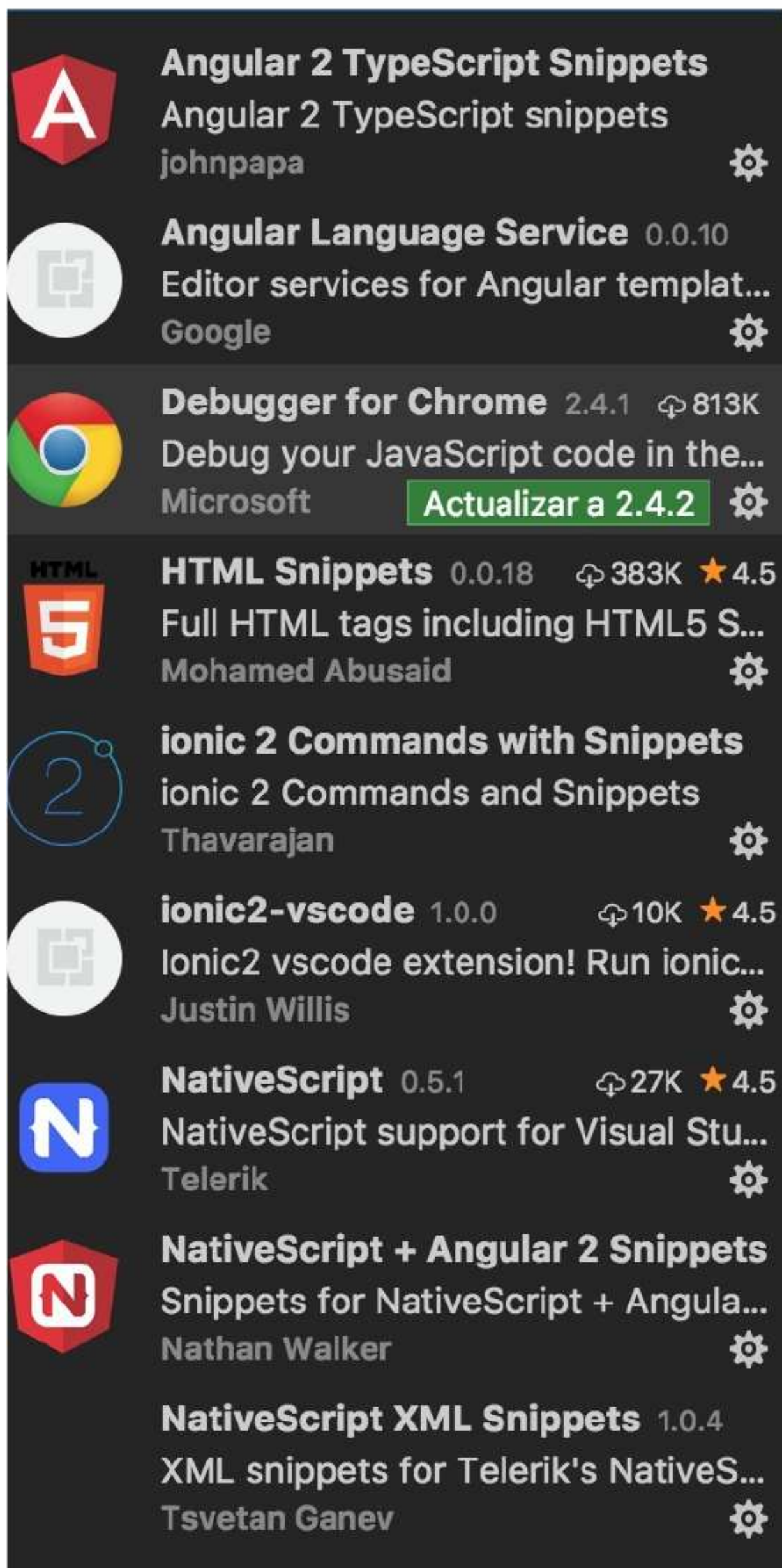
Editor de Texto => Visual Studio Code

Por lo tanto vos deberias tener estas cosas minimo para empezar, como dije anteriormente y vuelvo a repetir, el IDE o Editor de texto es a gusto de cada uno.

Personalmente utilizo WebStorm para los proyectos que son grandes y necesito de un backend (99.9% de las veces es node) y ahi puedo debuggear y conectarme al servidor tener GIT tener un manejo de versiones y cosas que me resultan comodas a la hora de tener un proyecto grande.

Ahora para los cursos, ejemplos, posts y videos suelo utilizar el Visual Studio Code, porque es relativamente mediando, me gusta el manejo que tiene y los plugins hacen que sienta que es lo que necesito de un editor de texto que no es a su vez un IDE.

Esta es una captura de mi VSC del día de hoy (2 de enero del 2017) en el cual se veran los plugins que utilizo.



The image shows a screenshot of the Visual Studio Code interface, specifically the 'Extensions' view. The background is dark grey. The list of extensions is as follows:

- Angular 2 TypeScript Snippets** by johnpapa. Icon: Red shield with a white 'A'. Description: Angular 2 TypeScript snippets. A gear icon is visible.
- Angular Language Service** 0.0.10 by Google. Icon: Grey circle with a white square. Description: Editor services for Angular templat... A gear icon is visible.
- Debugger for Chrome** 2.4.1 by Microsoft. Icon: Chrome logo. Description: Debug your JavaScript code in the... A green button says 'Actualizar a 2.4.2'. A gear icon is visible.
- HTML Snippets** 0.0.18 by Mohamed Abusaid. Icon: Orange shield with a white '5'. Description: Full HTML tags including HTML5 S... A gear icon is visible.
- ionic 2 Commands with Snippets** by Thavarajan. Icon: Blue circle with a white '2'. Description: ionic 2 Commands and Snippets. A gear icon is visible.
- ionic2-vscode** 1.0.0 by Justin Willis. Icon: Grey circle with a white square. Description: Ionic2 vscode extension! Run ionic... A gear icon is visible.
- NativeScript** 0.5.1 by Telerik. Icon: Blue square with a white 'N'. Description: NativeScript support for Visual Stu... A gear icon is visible.
- NativeScript + Angular 2 Snippets** by Nathan Walker. Icon: Red shield with a white 'N'. Description: Snippets for NativeScript + Angula... A gear icon is visible.
- NativeScript XML Snippets** 1.0.4 by Tsvetan Ganey. Description: XML snippets for Telerik's NativeS... A gear icon is visible.

Todos lo que veas en este libro lo vas a poder usar en cualquier sistema operativo que soporte NodeJS, por lo tanto si tenes Windows o Linux o Mac no vas a estar limitado a utilizar ninguna de las funcionalidades que vamos a ver a lo largo del libro, todos los comandos de consola pertenencen a NodeJS por lo tanto como dije, si soportas nodeJS soportas lo demas.

Ya elegiste tu IDE/Editor? entonces sigamos...

Mi primera App

Hay dos formas de trabajar sobre Angular, una es usar el CLI y la otra es traer todo a mano (como veniamos acostumbrados anteriormente), la verdad que ambas tienen sus beneficios y contras (como todo en la vida del desarrollador) pero vamos a ver ambas opciones y luego nos vamos a quedar por una hibrida.

SystemJS

La primer opción como comente seria una opcion "manual" por lo tanto deberiamos traer todo a mano y arrancar Angular con SystemJS.

Basandonos en el siguiente enlace de la documentacion oficial (<https://embed.plnkr.co/?show=preview&show=app%2Fapp.component.ts>) vamos a ver como seria un Hello Angular.



index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello Angular</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <style>
      body {color:#369;font-family: Arial,Helvetica,sans-serif;}
    </style>

    <!-- Polyfills for older browsers -->
    <script src="https://unpkg.com/core-js/client/shim.min.js"></script>

    <script src="https://unpkg.com/zone.js@0.7.4?main=browser"></script>
    <script src="https://unpkg.com/reflect-metadata@0.1.8"></script>
    <script src="https://unpkg.com/systemjs@0.19.39/dist/system.src.js"></script>

    <script> window.autoBootstrap = true; </script>

    <script src="https://cdn.rawgit.com/angular/angular.io/b3c65a9/public/docs/_examples/_boilerplate/systemjs.config.web.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err); });
    </script>
  </head>

  <body>
    <my-app>Loading AppComponent content here ...</my-app>
  </body>

</html>

<!--
Copyright 2016 Google Inc. All Rights Reserved.
Use of this source code is governed by an MIT-style license that
can be found in the LICENSE file at http://angular.io/license
-->
```

En nuestro index.html deberíamos poner todo esto... de esta forma traeríamos todos los JS necesarios para poder hacer nuestro primer "Hello Angular"...

Viendo un opco mas en detalle, traemos Shim, Zone, Reflect, systemjs y angular (la version que se hizo para el plnkr)...

Luego de eso ejecutamos SystemJS, para cargar la "app" ...

Y podemos observar que en el body tenemos un Componente llamado "my-app", el cual no recibe ningun tipo de parametro, pero si tiene texto adentro, si no lo adivinaste todavia, este texto esta simplemente para que la web muestre algo mientras se esta cargando el

contenido del componente, para que este cargue primero tiene que traer todas las dependencias que pusimos para que traiga la pagina y ahi pueda ejecutar SystemJS a Angular... pero esto no es todo...

app/app.component.ts

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

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

/*
Copyright 2016 Google Inc. All Rights Reserved.
Use of this source code is governed by an MIT-style license that
can be found in the LICENSE file at http://angular.io/license
*/
```

Esta era la parte que nos faltó, hasta ahora solo vimos ejemplos de componentes en su parte de HTML, pero no habíamos visto la parte que los define, que les da funcionalidad, quien muestra su estructura HTML, o quien le da la funcionalidad, o simplemente quien genera todo el componente.

Vamos a ver "muy por arriba" que es este TS (typescript).

Si nunca viste typescript, y venis de programar en algun lenguaje tipado y orientado a objetos, vas a sentir cierta comodidad... Si claro, estas viendo un import un annotation, una clase, esto parece algun tipo de magia oscura, pero no lo es.

Si venis del frontend todo esto puede llegar a hacerte mucho ruido, ya que JS me dejaba hacer lo que queria, aca veo cosas raras, no te preocupes, vamos a ver todo ...

Pero volviendo al ejemplo, este TS, importa "algo", que casualmente se llama componente, lo que quiere decir es que va a traer de "angular => core" la funcionalidad de componente.

Lo que se ve con un "@" (arroba) es un decorador (que es el componente) que lo que va a hacer en un modo simple de definir, es una serie de metadata para el navegador... Como por ejemplo como se llama el tag del componente, que es el **selector** (en nuestro caso serial <my-app> pero al annotation solo le tenemos que pasar el contenido no los < >), como así le damos un **template** osea una "pedazo" de HTML para que tenga una estructura web... y luego **exporta** una clase que simplemente crea una variable del nombre "name" y le asigna el valor "Angular"...

Muy bien, ya te diste cuenta no? El "`{{ }}`" (si doble llave, que lo podriamos llamar de muchas formas, para nosotros va a ser doble llave) es lo que une el HTML o Template con la Clase... por lo tanto, cuando escribimos en nuestro template `__` lo que va a hacer Angular es buscar una variable en la clase correspondiente con el mismo nombre (respetando mayusculas y minusculas) y unir el contenido de la variable con la vista del template.

Hasta ahi parece todo bastante facil para arrancar... pero la verdad que en este ejemplo tenemos servido algo que luego de un tiempo y de necesidad de agrandar las funcionalidades de nuestra WEBAPP vamos a tener que modificar y ahi es donde se vuelve mas complicado utilizar y es el archivo de configuracion de SystemJS.

systemjs.config.web.js

Este que tenemos nosotros en el ejemplo es uno hecho y dedicado para el ejemplo inicial de angular el cual nos serviria de base para arrancar un proyecto.

```
/**
 * WEB ANGULAR VERSION
 * (based on systemjs.config.js in angular.io)
 * System configuration for Angular samples
 * Adjust as necessary for your application needs.
 */
(function (global) {
  System.config({
    // DEMO ONLY! REAL CODE SHOULD NOT TRANSPILE IN THE BROWSER
    transpiler: 'ts',
    typescriptOptions: {
      // Copy of compiler options in standard tsconfig.json
      "target": "es5",
      "module": "commonjs",
      "moduleResolution": "node",
      "sourceMap": true,
      "emitDecoratorMetadata": true,
      "experimentalDecorators": true,
      "noImplicitAny": true,
      "suppressImplicitAnyIndexErrors": true
    },
    meta: {
      'typescript': {
        "exports": "ts"
      }
    },
    paths: {
      // paths serve as alias
      'npm:': 'https://unpkg.com/'
    },
    // map tells the System loader where to look for things
    map: {
      // our app is within the app folder
      app: 'app',
```

```
// angular bundles
'@angular/core': 'npm:@angular/core/bundles/core.umd.js',
'@angular/common': 'npm:@angular/common/bundles/common.umd.js',
'@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
'@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platform-browser.umd.js',
'@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js',
'@angular/http': 'npm:@angular/http/bundles/http.umd.js',
'@angular/router': 'npm:@angular/router/bundles/router.umd.js',
'@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
'@angular/upgrade': 'npm:@angular/upgrade/bundles/upgrade.umd.js',
'@angular/upgrade/static': 'npm:@angular/upgrade/bundles/upgrade-static.umd.js',

// other libraries
'rxjs': 'npm:rxjs',
'angular-in-memory-web-api': 'npm:angular-in-memory-web-api/bundles/in-memory-web-api.umd.js',
'ts': 'npm:plugin-typescript@4.0.10/lib/plugin.js',
'typescript': 'npm:typescript@2.0.3/lib/typescript.js',

},
// packages tells the System loader how to load when no filename and/or no extension
packages: {
  app: {
    main: './main.ts',
    defaultExtension: 'ts'
  },
  rxjs: {
    defaultExtension: 'js'
  }
}
});

if (global.autoBootstrap) { bootstrap(); }

// Bootstrap with a default `AppModule`
// ignore an `app/app.module.ts` and `app/main.ts`, even if present
// This function exists primarily (exclusively?) for the QuickStart
function bootstrap() {
  console.log('Auto-bootstrapping');

  // Stub out `app/main.ts` so System.import('app') doesn't fail if called in the index.html
  System.set(System.normalizeSync('app/main.ts'), System.newModule({ }));

  // bootstrap and launch the app (equivalent to standard main.ts)
  Promise.all([
    System.import('@angular/platform-browser-dynamic'),
    get AppModule()
  ])
}
```

```
.then(function (imports) {
  var platform = imports[0];
  var app      = imports[1];
  platform.platformBrowserDynamic().bootstrapModule(app.AppModule);
})
.catch(function(err){ console.error(err); });
}

// Make the default AppModule
// returns a promise for the AppModule
function getAppModule() {
  console.log('Making a bare-bones, default AppModule');

  return Promise.all([
    System.import('@angular/core'),
    System.import('@angular/platform-browser'),
    System.import('app/app.component')
  ])
  .then(function (imports) {

    var core    = imports[0];
    var browser = imports[1];
    var appComp = imports[2].AppComponent;

    var AppModule = function() {}

    AppModule.annotations = [
      new core.NgModule({
        imports:    [ browser.BrowserModule ],
        declarations: [ appComp ],
        bootstrap:  [ appComp ]
      })
    ]
    return {AppModule: AppModule};
  })
}
})(this);

/*
Copyright 2016 Google Inc. All Rights Reserved.
Use of this source code is governed by an MIT-style license that
can be found in the LICENSE file at http://angular.io/license
*/
```

WOW, si es un poco mas grande de lo que esperabamos, pero simplemente yo quiero utilizar el framework no quiero tener que estar configurando mil cosas, para eso vamos a tener el CLI.

_Advertencia: _este archivo de configuracion (systemjs.config.web.js) no lo deberiamos utilizar tal y como esta, ya que esta transpilando momento de ejecución

AngularCLI

No es nada nuevo que un framework tenga un CLI hoy en día para los que trabajamos en frontend o mejor dicho con JS a lo largo de los últimos años, es algo que se está volviendo común y está bueno para algunas cosas y para otras no... Desde el AngularCLI, a partir de ahora CLI, nosotros podríamos tranquilamente hacer toda la app, desde crearla, agregar funcionalidades base, arquitectura de archivos, ejecutar los test, ejecutar un servidor de prueba local, hasta hacer el build final sin tener que CREAR una carpeta o archivo dentro del proyecto a "mano", ya sea apretando click derecho en nuestro navegador de carpetas o desde nuestro editor de texto.

Pero después de tiempo de utilizarlo, realmente creo que una opción intermedia es lo mejor que podemos utilizar... para eso primero vamos por la opción 100% CLI.

Para instalar el CLI (Documentación oficial => <http://cli.angular.io/>), vamos a tener que abrir nuestra consola e hacer lo siguiente (entendiendo que tienen NodeJS instalado ya, si no lo tienes entra a <http://www.nodejs.org/es/> y descargalo)

```
npm install -g angular-cli
```

Una vez que termine vamos a poder empezar a crear proyectos directamente desde nuestra terminal, para eso vamos a tener que ejecutar el siguiente comando.

```
ng new NombreProyecto
```

este por defecto nos va a crear un proyecto desde cero en Typescript, en el caso de no quieres utilizar TypeScript (si no le diste una oportunidad es hora que lo hagas) vas a tener que utilizar inputs para que traiga el template inicial de JavaScript o de Dart.

Volviendo al comando vamos a tener la misma solución una vez que arranquemos nuestro localhost...

```
ng serve
```

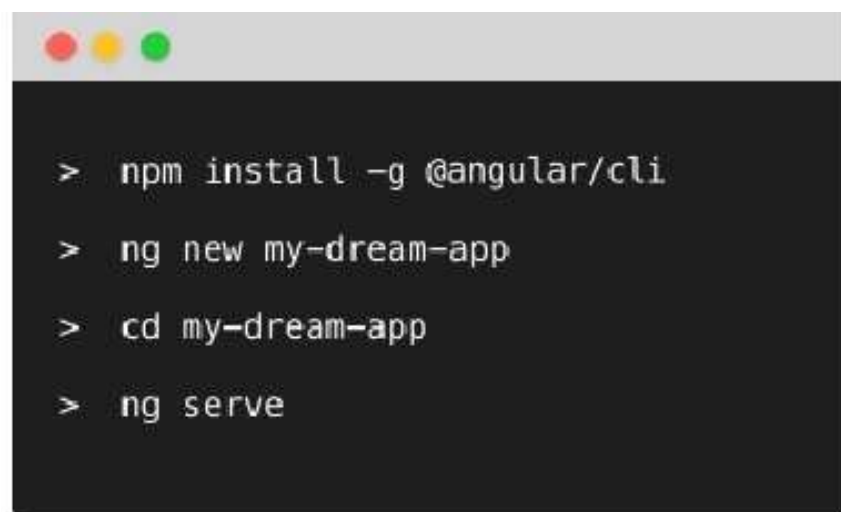
que va a hacer ? A simple modo, lo que va a hacer es levantarnos un servidor web liviano y mostrarnos el resultado de nuestra WEBAPP aunque no hicimos nada todavía...

Preview

Hello Angular

Si, la preview es la misma, es el mismo resultado pero mucho mas rapido todavia!

Angular-CLI

A terminal window with a dark background and light gray text. It shows four commands being executed in sequence, each preceded by a prompt character '>'. The commands are: 'npm install -g @angular/cli', 'ng new my-dream-app', 'cd my-dream-app', and 'ng serve'. The window has a standard macOS-style title bar with red, yellow, and green window control buttons at the top left.

```
> npm install -g @angular/cli
> ng new my-dream-app
> cd my-dream-app
> ng serve
```

Como habras visto, es bastante facil utilizar el CLI, ademas amigarse con la consola (si todavia no lo hiciste) es algo super importante para el desarrollo de hoy en día...

Pero este CLI realmente tiene una serie de ventajas importantes, pero arranquemos por el principio.

¿Que hace?

Las tareas que vamos a poder hacer con el CLI son:

- Crear una aplicación => `ng new NombreAPP`
- Crear un componente => `ng g component mi-componente`
- Crear una directiva => `ng g directive mi-directiva`
- Crear un pipe => `ng g pipe mi-pype`
- Crear un servicio => `ng g service mi-servicio`
- Cear una clase => `ng g class mi-clase`
- Crear una interfaz => `ng g interface mi-interface`
- Crear un enumerador => `ng g enum mi-enumerador`
- Crear un modulo => `ng g module mi-modulo`
- Hacer un build => `ng build`
- Ejecutar los test => `ng test`
- Ejecutar los e2e => `ng e2e`
- Deploy en githubpage => `ng github-page:deploy --message "esto es un mensaje opcional"`
- Ejecutar tsLint => `ng lint`
- Setear y procesar SASS => `ng set defaults.styleExt scss`
- Ejecutar un server local => `ng serve`
- Ejecutar un server con ssl => `ng serve --ssl true`

Pareceria que hace todo por nosotros, solo tendríamos que escribir comandos en la consola y toda la arquitectura de carpetas y archivos en la aplicacion se harian solas gracias al CLI, y luego nos tendríamos que sentar a escribir TS CSS y HTML... Seria algo muy muy muy armado (si escribi 3 veces muy) ...

En mi perspectiva y con el tiempo que llevo trabajando con Angular y el CLI, entendi que para ciertas cosas esta bueno el CLI y para otras me gusta hacerlo manualmente...

La disposición de las carpetas, o el contenido de cada una me gusta hacerlo a mano... como asi crear los componentes inyectables y todo lo demas...

Por lo tanto yo uso el CLI para => **Crear una aplicacion, haber un build, levantar un servidor y ejecutar los test.**

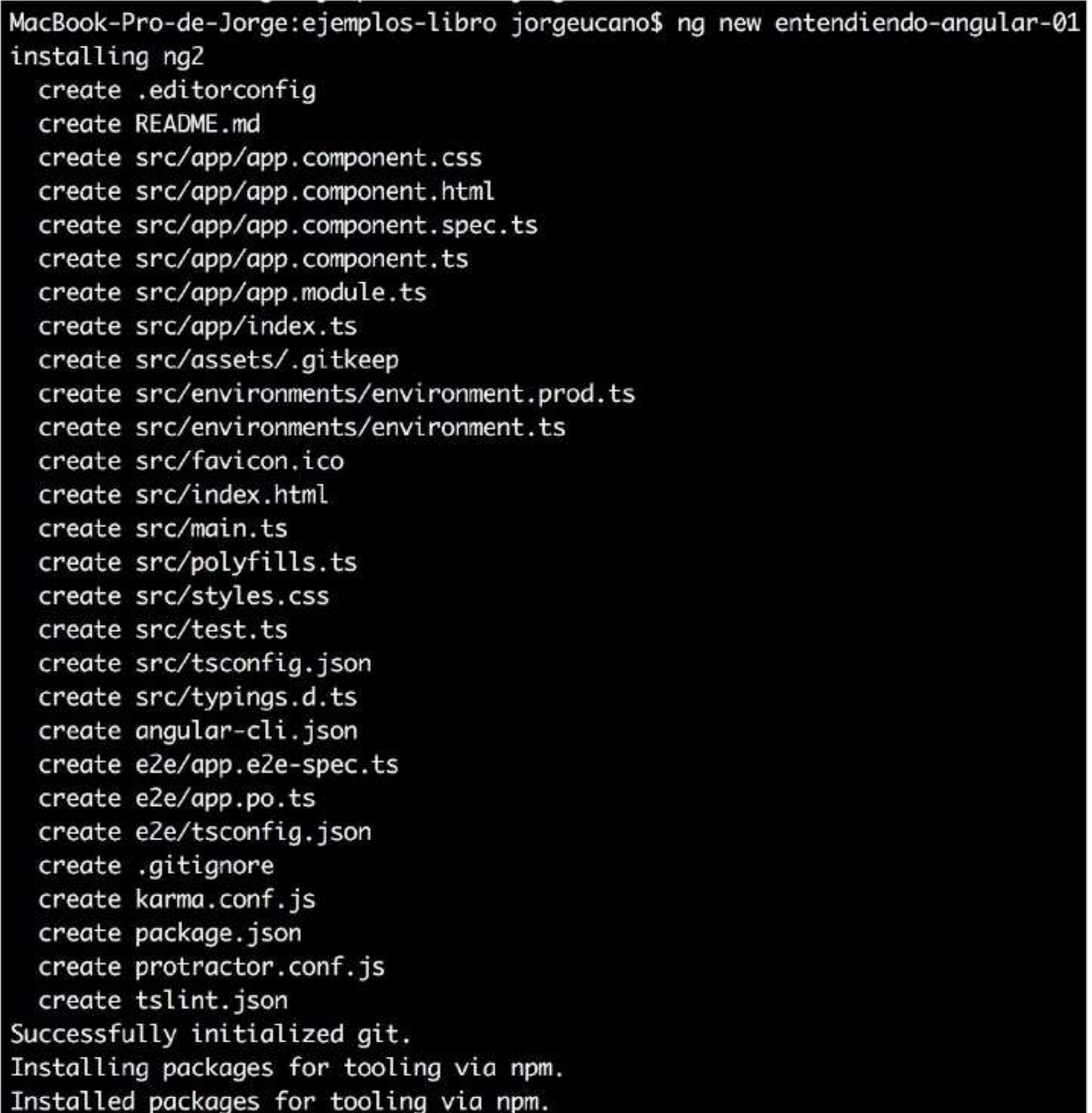
En lo demas me gusta tener un poco mas el control sobre lo que hago, por lo tanto, es como tener una solución hibrida entre, bajarte todo a mano y hacer todo con el CLI.

Algo importante a tener en cuenta, el CLI de Angular esta realizado con WebPack por lo tanto, cuando ejecutemos algunos de los comandos en realidad hay toda una configuracion de WebPack por atras que hace y genera todo lo necesario para luego ir a internet (por medio de NPM) si es necesario.

Estructura de una Angular WebAPP

Vamos a crear nuestra WebApp para empezar a ver la estructura de carpetas y contenido que tiene una app en Angular.

```
ng new entendiendo-angular-01
```

A terminal window with a black background and white text. The prompt is 'MacBook-Pro-de-Jorge:ejemplos-libro jorgeucano\$'. The command 'ng new entendiendo-angular-01' has been executed. The output shows a list of files and folders being created, including 'src/app/app.component.css', 'src/app/app.component.html', 'src/app/app.component.spec.ts', 'src/app/app.component.ts', 'src/app/app.module.ts', 'src/app/index.ts', 'src/assets/.gitkeep', 'src/environments/environment.prod.ts', 'src/environments/environment.ts', 'src/favicon.ico', 'src/index.html', 'src/main.ts', 'src/polyfills.ts', 'src/styles.css', 'src/test.ts', 'src/tsconfig.json', 'src/typings.d.ts', 'angular-cli.json', 'e2e/app.e2e-spec.ts', 'e2e/app.po.ts', 'e2e/tsconfig.json', '.gitignore', 'karma.conf.js', 'package.json', 'protractor.conf.js', and 'tslint.json'. It also shows 'Successfully initialized git.', 'Installing packages for tooling via npm.', and 'Installed packages for tooling via npm.'

```
MacBook-Pro-de-Jorge:ejemplos-libro jorgeucano$ ng new entendiendo-angular-01
installing ng2
  create .editorconfig
  create README.md
  create src/app/app.component.css
  create src/app/app.component.html
  create src/app/app.component.spec.ts
  create src/app/app.component.ts
  create src/app/app.module.ts
  create src/app/index.ts
  create src/assets/.gitkeep
  create src/environments/environment.prod.ts
  create src/environments/environment.ts
  create src/favicon.ico
  create src/index.html
  create src/main.ts
  create src/polyfills.ts
  create src/styles.css
  create src/test.ts
  create src/tsconfig.json
  create src/typings.d.ts
  create angular-cli.json
  create e2e/app.e2e-spec.ts
  create e2e/app.po.ts
  create e2e/tsconfig.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tslint.json
Successfully initialized git.
Installing packages for tooling via npm.
Installed packages for tooling via npm.
```

Ahi podemos observar en la imagen (podria variar algo con el paso del tiempo, ire actualizando), que crear una serie de archivos y carpetas, y luego ejecuta via NPM la descarga de los archivos que necesita... lo hace porque genera un package.json para poder descargar todas las dependencias que tiene un proyecto basico de Angular.

2.4 - Estructura de una Angular WebAPP

Antes de ver las carpetas y archivos, ejecutemos el servidor para ver que pasa...

```
ng serve
```

```

MacBook-Pro-de-Jorge:entendiendo-angular-01 jorgeucano$ ng serve
clear
** NG Live Development Server is running on http://localhost:4200. **
5529ms building modules
41ms sealing
0ms optimizing
0ms basic module optimization
174ms module optimization
0ms advanced module optimization
7ms basic chunk optimization
0ms chunk optimization
0ms advanced chunk optimization
1ms module and chunk tree optimization
76ms module reviving
8ms module order optimization
4ms module id optimization
4ms chunk reviving
0ms chunk order optimization
15ms chunk id optimization
78ms hashing
0ms module assets processing
189ms chunk assets processing
4ms additional chunk assets processing
0ms recording
1ms additional asset processing
1518ms chunk asset optimization
75ms asset optimization
43ms emitting
Hash: 2674cb8605acdea80a46
Version: webpack 2.1.0-beta.25
Time: 7790ms

      Asset      Size  Chunks             Chunk Names
  main.bundle.js  2.72 MB    0, 2  [emitted]  main
  styles.bundle.js 10.3 kB    1, 2  [emitted]  styles
  inline.bundle.js  5.54 kB      2  [emitted]  inline
  main.bundle.map  2.86 MB    0, 2  [emitted]  main
  styles.bundle.map 14.4 kB    1, 2  [emitted]  styles
  inline.bundle.map  5.6 kB      2  [emitted]  inline
  index.html      494 bytes             [emitted]

Child html-webpack-plugin for "index.html":
      Asset      Size  Chunks             Chunk Names
  index.html  2.82 kB      0

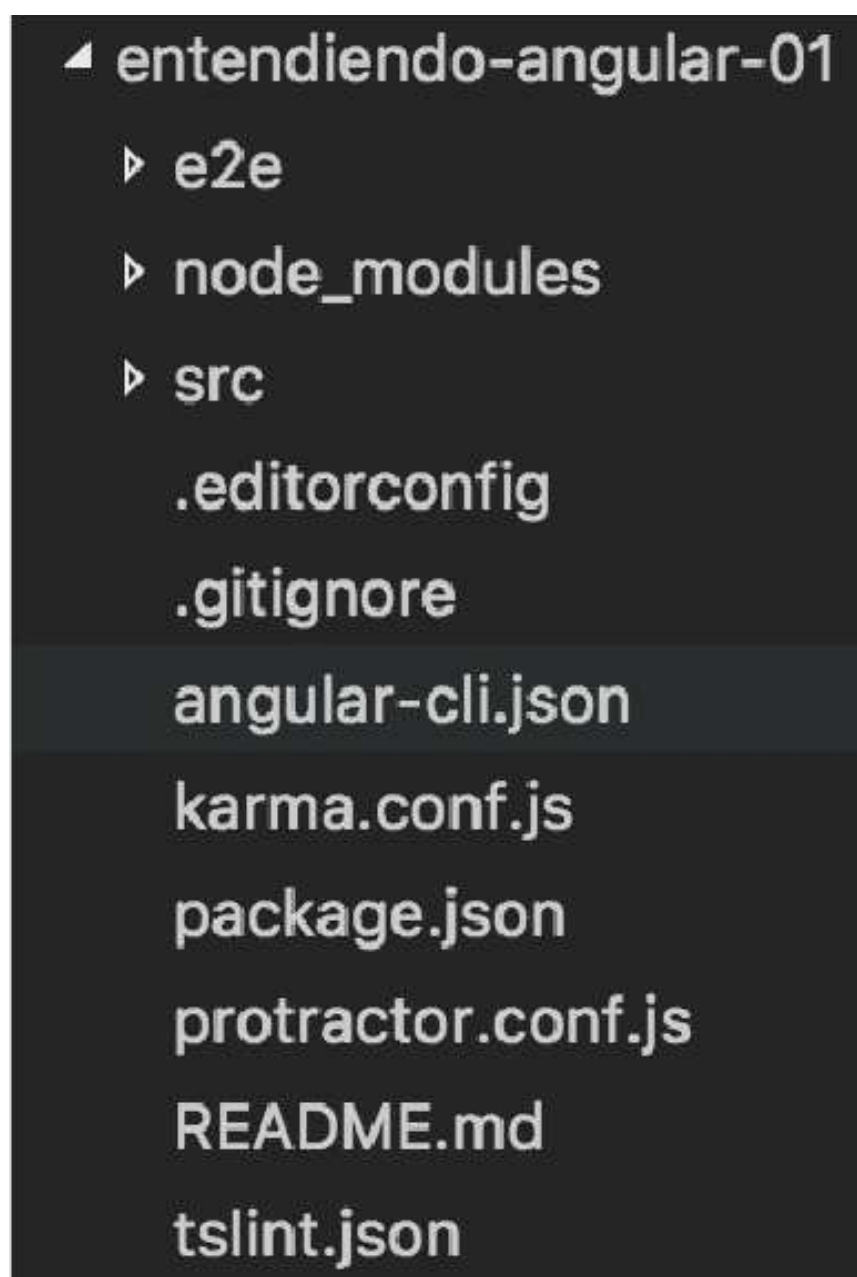
webpack: bundle is now VALID.
[default] Checking started in a separate process...
[default] Ok, 1.561 sec.

```

Okey, ahora que tenemos nuestro servidor en el `http://localhost:4200/` podemos entrar a ver que tiene...

app works!

Si, un titulo que simplemente dice "app works!" ... vamos a ver como se genera en las carpetas...



Vamos a arrancar por los archivos de la carpeta raiz.

.editorconfig => es de VSC es el archivo de configuracion.. no haremos nada sobre el..

.gitignore => El CLI nos genera un git ya para que podamos subir nuestro codigo a algun repositorio de GIT, puede ser GITHUB o cualquiera que tenga soporte sobre GIT. Y ya que estan nos dejan todo preparado para ignorar las dependencias que no tiene sentido que subamos.

angular-cli.json => Es este archivo vamos a ver la configuración que genero el CLI para el proyecto.

karma.conf.js => Es el archivo de la configuracion de Karma para los tests.

package.json => Es la configuracion para descargar las dependencias de NodeJS.

protractor.conf.js => Es la configuracion de jasmine para los test e2e.

README.md => es el readme para git, que te explica algunos comandos del CLI para quien se baje el proyecto.

tslint.json => es la configuracion de lint para typescript.

/e2e => aqui tendremos los test e2e para ejecutar, por defecto ya viene con uno, que verifica si en la pantalla existe un H1 que tenga el texto "app works!".

/node_modules => aqui estan todas las dependencias del proyecto

/src => aca es donde mas vamos a estar , diria yo que el 99.9% del tiempo (el 0.01% seria para los test y configuración, capaz que exagere y es un poco mas para test, SI!!!! tenes que hacer test, ya hacer frontend es de profesionales! =D) , en esta carpeta estara todo el codigo que programemos, asi que esta carpeta la vamos a ver mas en detalle y la vamos a utilizar todo en todo el libro.

src

En esta carpeta como bien imaginaron va a estar toda la programación que hagamos, y donde vamos a trabajar... por lo tanto vamos a ver un poco los archivos de adentro para entender la estructura basica para poder entender todo bien.

/app=> aquí y en sus subcarpetas vamos a crear todos los componentes, directivas, servicios y todo lo que refiera con la vista y su funcionamiento.

/assets=> (creo que no hace falta decirlo) pero aca van todos los assets que tengamos

/environment=> aquí vamos a definir en el entorno que estamos trabajando y que variables se exportan en el mismo.

typings.d.ts => en este archivo vamos a tener las declaraciones globales de typescript

tsconfig.json=> configuracion de TypeScript

test.ts=> Karma

style.css=> nuestra hoja de estilos global

polyfills.ts=> Este archivo incluye polyfills necesarios para Angular 2 y se carga antes de la aplicación.

main.ts=> nuestro archivo de entrada a Angular

index.html => En esta html se renderizaran las paginas.

app

Dentro de app, es donde vamos a programar, por lo tanto vamos a ver los archivos.

app.component.ts => aca tenemos definido el root component

app.module.ts => aca tenemos la entrada del bootstrap del proyecto y manejaremos todos los modulos que contenga

app.component.html => estructura html del componente

index.ts => export de todos los componentes y modulos

app.module.ts

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

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

app.component.html

```
<h1>
  {{title}}
</h1>
```

index.ts

```
export * from './app.component';
export * from './app.module';
```

app.module.ts

Este archivo es muy importante mientras desarrollemos en Angular, porque aqui vamos a manejar los hilos principales.

Es importante saber que este archivo es tan importante (tiene el decorador `ngModule`) que hay todo un capitulo dedicado a su funcionamiento.

Directivas

Si alguna vez utilizaste AngularJS muchas cosas te serán conocidas, ahora si nunca utilizaste AngularJS no te hagas drama, también te parecerán conocidas.

Angular nos ofrece una serie de directivas para poder trabajar de una forma más directa en nuestros templates, estas no tienen que ser agregadas en nuestro componente, ni importadas, ni nada por el estilo, simplemente tenemos que utilizarlas donde las necesitamos y listo.

Las directivas son palabras "reservadas" que ejecutan funcionalidad en base a la directiva misma, o a datos que le enviemos.

1. NgIf
2. NgSwitch
3. NgStyle
4. NgClass
5. NgFor

NgIf

En esta directiva, vamos a poner un valor (puede ser una operación, una variable, una función) y en el caso de que sea VERDADERO va a mostrar el contenido del tag al que se lo hayamos puesto, en caso contrario lo escondera.

<code><div *ngIf="false"> Hola </div></code>	<code><<< nunca se mostrara</code>
<code><div *ngIf="a < b"> A es menor a b </div></code>	<code><<< se mostrara si y solo si A es mayor a B</code>
<code><div *ngIf=" str == 'yes'"> YES </div></code>	<code><<< Se mostrara si str es 'yes'</code>
<code><div *ngIf= "myFunct()"> HOLA </div></code>	<code><<< Se ejecuta si el retorno de la funcion es true</code>

Si te diste cuenta ya, para las directivas de Angular vamos a tener que ponerle el '*' (asterisco) adelante.

NgSwitch

Algunas veces necesitamos renderizar diferentes elementos dependiendo de una condición.

Cuando nos pase esto tendríamos con ngIf una forma de hacerlo =>


```
<div class="container">
  <span *ngIf="variable == 'A'"> la variable es A </span>
  <span *ngIf="variable == 'B'"> la variable es B </span>
  <span *ngIf="variable != 'A' && variable != 'B'"> La variable no es ni A ni B </span>
</div>
```

Y se pone peor cuando tenemos mas posibilidades, como si tendríamos que comparar por C, D, E ...

```
<div class="container">
  <span *ngIf="variable == 'A'"> la variable es A </span>
  <span *ngIf="variable == 'B'"> la variable es B </span>
  <span *ngIf="variable == 'C'"> la variable es C </span>
  <span *ngIf="variable == 'D'"> la variable es D </span>
  <span *ngIf="variable == 'E'"> la variable es E </span>
  <span *ngIf="variable != 'A' &&
    variable != 'B' &&
    variable != 'C' &&
    variable != 'D' &&
    variable != 'E'"> La variable no es ni A ni B </span>
</div>
```

Para este tipo de casos, Angular nos ofrece ngSwitch, si tal y como el nombre lo indica, es algo que vamos a poder entender facilmente ya que se basa en la funcionalidad del switch

```
<div class="container" [ngSwitch]="variable">
  <span *ngSwitchCase="'A'"> la variable es A </span>
  <span *ngSwitchCase="'B'"> la variable es B </span>
  <span *ngSwitchDefault> La variable no es ni A ni B </span>
</div>
```

En el caso de que tengamos que agregar nuevas posibilidades, nuestro código va a seguir siendo limpio:

```
<div class="container" [ngSwitch]="variable">
  <span *ngSwitchCase="'A'"> la variable es A </span>
  <span *ngSwitchCase="'B'"> la variable es B </span>
  <span *ngSwitchCase="'C'"> la variable es C </span>
  <span *ngSwitchCase="'D'"> la variable es D </span>
  <span *ngSwitchCase="'E'"> la variable es E </span>
  <span *ngSwitchDefault> La variable no es ni A ni B </span>
</div>
```

NgStyle

Con `ngstyle` podemos obtener el elemento del DOM y agregarle las propiedades que queramos por medio de una expresion de Angular.

```
<div [style.background-color]='red'>
  el background color es rojo
</div>
```

En este caso al `style` (del tag en el que nos encontramos, osea al elemento del dom que pertenece) le agrega un background color red.

Hasta ahi no tendria mucho sentido hacerlo, pero vamos a avanzar un poco mas con las funcionalidades de `ngstyle`.

```
<div [NgStyle]="{color: 'white', 'background': 'red' }">
  Texto blanco, fondo rojo
</div>
```

Pero esto parece muy "sucio" es como volver a los 90 cuando haciamos "style in line", pareceria mucho mejor agregar una clase con esos estilos, ahora que pasa cuando esto lo "bindeamos" a una variable, aps!! ahora cambia todo no?

```
<h4>
  ngStyle con el objeto y su propiedad desde una variable
</h4>

<div>
  <span [ngStyle]="{color: color}">
    Hola, soy la variable {{color}}
  </span>
</div>

<h4>
  Style desde una variable directa
</h4>

<div>
  <span [style.background-color]="color"
    style="color: white;">
    el background es {{color}}
  </span>
</div>
```

Al cual podriamos poner un boton para que cambie el color

```
aplicarColor(color:string):void{
    this.color = color;
}
```

Por lo tanto, cuando le ponemos una variable, ya podemos jugar con los estilos dependiendo de las necesidades de la webApp

esto nos lleva a la proxima directiva

NgClass

NgClass es la directiva que representa al atributo de la clase en el HTML (template) y esto puede generar que cambien los estilos dinamicamente en nuestro DOM.

Lo primero que podemos hacer con NgClass es pasaer un literal (string). El objeto espera una clase y un valor (verdadero o falso) para poder aplicar o no el estilo.

```
.fondo-rojo{
    background: red;
}
```

Ahora vamos a ver como impactaria en distintos divs.

```
<div [ngClass]="{ fondo-rojo : false }"> Este fondo NO es rojo nunca</div>
<div [ngClass]="{ fondo-rojo : true }"> Este ondo siempre sera rojo </div>
```

Otra alternativa es definir un objeto en nuestro componente:

```
export class NgClassApp{
    esRojo: boolean;
    classesObj: Object;
    classList: string[];
}
```

Y en nuestro HTML:

```
<div [ngClass]="classesObj">
    Usando un objeto {{ classesObj.background ? "ON":"OFF" }}
</div>
```

Otro caso seria poniendole el estilo directo

```
<div [ngClass]="['background-red', 'redondeado']">
  background red y bordes redondeados
</div>
```

O podriamos declararlo en un array y ponerlo directo

```
this.classListStyle = ['background-red', 'redondeado'];
```

```
<div [ngClass]="classListStyle">
  Este background {{classListStyle.indexOf('roja-background') > -1 "" : "no"}} es ro
  jo
  Este div {{classListStyle.indexOf('redondeado') > -1 "" : "no"}} es redondeado
</div>
```

De esta forma vamos a poder ir jugando con nuestro estilos directamente.

NgFor

NgFor es nuestra directiva iteradora, es como un foreach, pero con la diferencia que va a iterar en nuestro template, y generar variables con 'let' para que sean unicas y se liberen en el momento

Si tenemos un array de colores:

```
this.colorArray = ['rojo', 'blanco', 'verde', 'azul'];
```

Lo que deberiamos hacer para iterarlos en nuestro Template es lo siguiente:

```
<ul>
  <li *ngFor="let color of colorArray">
    {{color}}
  </li>
</ul>
```

De esta forma vamos a tener una serie colores iterados... si tan simple como poner esa linea y los "" se repetiran.

Ahora, que pasa si queremos iterar un array de objetos que vinieron de nuestro backend por ejemplo:

```
this.tickets = [
  { id: 1, name: 'no me funciona la impresora', state:'in progress'},
  { id: 2, name: 'no me funciona el mouse', state:'open'},
  { id: 3, name: 'no me funciona el office', state:'close'},
  { id: 4, name: 'no me funciona la impresora', state:'in progress'}
];
```

Ahora que tenemos nuestro array de objetos, vamos a iterarlo.

```
<div *ngFor="let ticket of tickets">
  <div class="container">
    <span class="col-md-3"> ID: {{ticket.id}} </span>
    <span class="col-md-6"> {{ticket.name}} </span>
    <span class="col-md-3"> {{ticket.state }} </span>
  </div>
</div>
```

Lo que vemos ahora, que "ticket" pasa a ser una variable que tiene el objeto que estamos iterando.. cada pasada va a obtener 1 de los objetos y lo va a iterar como el mismo, por lo tanto con el "." vamos a poder acceder a sus variables internas.

Componentes

Como bien mencione anteriormente, Angular es un arbol de Componentes, por lo tanto, una de las cosas mas importantes para entender y poder desarrollar con este fantastico framework es entender los componentes.

Un componente consta de 3 partes:

- Estructura (HTML)
- Estilos (CSS)
- Funcionalidad (TS /JS)

Vamos a pasar por todos los pasos de este Componente basico:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
}
```

(<https://gist.github.com/jorgeucano/68910a54c717f470e5923790544f346b>)

En la primera linea, vemos que hace un "import", lo que quiere decir esto es que va a traer el "Component" que esta en '@angular/core'... Sigamos bajando esto mas a la realidad...

En `@angular/core` => estan parte de las funcionalidades y objetos que tiene el framework, por lo tanto de todas las que tiene va a traer "Component".

Para los que vienen de programación orientada a objetos, esto les parece algo comun (importar una funcionalidad) ahora los que vienen mas del frontend (epoca de jquery o demas) estan mas acostumbrados que con traer algo en un `<script>` ya se podia utilizar en cualquier lado.

Pero en Angular no es asi, lo que vamos a hacer en nuestros componentes es solamente importar lo que necesitamos para poder utilizarlo en ese momento.

@Component() es un annotation, es lo que hace es enriquecer a la metadata de la clase que creamos para que termine siendo un componente web que el navegador entienda. Este recibe un json con datos, donde los principales son:

selector => es quien se encarga del "nombre" del tag del componente... por ejemplo este es <app-root>, pero solo vamos a poner el nombre, ya que de los tags se encarga angular.

template => en template (no esta en el ejemplo), podemos poner una porcion de html directo en nuestro componente, con los apostrofes frances ` (son de arriba a abajo, de izquierda a derecha) podemos utilizar el multilinea para nuestro html.

templateUrl => a diferencia del "template" aquí vamos a poner la dirección y el archivo donde se encuentra la porcion de HTML que dara la estructura .

StyleUrls => aquí vamos a tener que poner un array de estilos para el componente... como bien comente anteriormente tenemos un css de estilos globales, que afectan a todo el proyecto, los estilos que pongas en este componente van a estar y ser utilizados solo para el mismo.

Aclaración: vamos a ver mas cosas sobre este annotation pero por ahora con esto estamos bien!

Y por ultimo tenemos la clase exportada, que le va a dar **vida** a nuestro componente.

Esta clase no tiene que tener absolutamente nada para que funcione el componente, pero obviamente nosotros necesitamos que tenga funcionalidad...

Esta clase va a tener el binding con el html para dar funcionalidad en la vista, por lo tanto, las variables sin declarar, van a ser las variables que encontremos en el html, como asi las funciones tambien.

En este ejemplo podemos ver que tiene

```
title = ' app works!';
```

y en el html vamos a ver

```
{{title}}
```

que es el "binding" o la "conexion" entre la clase y el html.

Creo que es hora de ver el HTML de este componente.

```
<h1>
  {{title}}
</h1>
```

Si, lo se , es super basico... pero estamos viendo lo principal de un componente para poder ir avanzando poco a poco. Entonces como les dije... tenemos el binding gracias a las doble llaves, y eso hace el resultado final!

En un par de capitulos mas adelante, vamos a ver "componentes avanzados" en donde le sacaremos el jugo maximo al componente, pero creo que es importante ver algunas cosas antes de avanzar muy a fondo con nuestro componente, por eso este capitulo va a "no ser tan grande" capaz que como el de componentes avanzados u otros capitulos pero seria como un poquito mas del capitulo de primeros pasos...

Pero espera!! Todavia no termino !!

Cuando la aplicacion empieza a crecer los componentes se vuelven mas complicados y van a necesitar de otras cosas... como por ejemplo conectarse con un servicio rest o algun otro servicio.

Como tambien podrian estar calculando alguna funcion, o tienen que reaccionar mediante algun cambio fuerte... por lo tanto vamos a necesitar que ciertas funcionalidades sean mas "Common" con el resto del proyecto... el tema es que los componentes son "shadows" o "encapsulados" las funcionalidades de cada componente le pertenecen solo y nada mas que al componente y eso hace que no se pueda compartir la funcion, a pesar de

Funciones

Los componentes pueden terminar quedando super completos, y para eso podemos ir viendo distintos temas que deberíamos aprender.

Una parte son las funciones, ya vimos que los componentes tienen html, css y ts... la parte de TS, es la que genera el componente en si....

Y vimos que si creamos una variable se "bindea" si queremos con la parte html... ahora veremos un poco sobre las funciones.

Para crear una función, simplemente tenemos que indicar el nombre, que parametros reciben y que devuelve, pero mejor veamos en un ejemplo esto

```
miFuncion(variable:string):void{
    console.log("la variable que ingreso en nuestra funcion es :" + variable);
}

miFuncionConDevolucion():string{
    return "Que bueno saber que tipo va a devolver";
}
```

En el momento de transpilar esto quedara como funciones mas parecidos a "javascript", pero en el momento de transpilar, typescript va a chequear que lo que devuelvas sea un string en el segundo caso, o que quien llame a "miFuncion" le envíe un string.

Esto seria la parte en donde utilizamos las funciones dentro de lo que seria "interno" del componente, si quisieramos llamarlo desde el HTML (o la "vista") deberíamos ejecutar otros tipos de funciones.

```
<a (click)="miFuncion('le paso una variable')"> Click Me </a>
```

Con este "HTML" lo que vamos a hacer es ejecutar una funcion "click" que va a ejecutar una funcion "miFuncion" , de esa forma vamos a conectar el HTML con el TS ...

Para entender un poco mas, los atributos de los "TAGS" que tengan los parentesis () , van a estar "bindeados" con el TS , por lo tanto por ejemplo si el es TAP o CLICK o alguno otro, van a ser funciones que se ejecuten dentro de Angular, seria como un equivalente a `onClick=""` y que eso ejecute una función.

Vale aclarar que nuestras funciones pueden ser lo mas complejo posible, y eso dependeria de nosotros, y de que tan funciona queramos hacer nuestras funciones, recuerden que mientras mas atomicas sean las funciones mas reutilizables van a ser.

INPUT

Hay varios tipos de componentes, el que vimos seria un "Componente Principal" lo que quiero decir con esto, es que ese componente, tiene una interacción inicial, y eso seria entrar como "HomeComponent", o que entra por el router (que vamos a ver mas adelante) por lo tanto esos componentes lo que hacen es directamente tener un init como "principal".

Ahora podriamos tener componentes dentro de componentes, y si nos ponemos a pensar es como super normal, imaginemos que tenemos unos datos iterables y esos tengan un componentes que podriamos "reutilizar" en distintos lugares de la aplicación, ya sea por su vista o su funcionalidad...

Estos componentes tienen interacciones y una de esas interacciones es el annotation `@Input`.

Un componente input super basico seria asi =>

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

@Component({
  selector: 'input-component',
  template: `
    <li> {{entrada}} </li>
  `,
  styleUrls: ['input.component.css']
})
export class InputComponent{
  @Input() entrada:string;

  constructor(){
    console.log(this.entrada);
  }

}
```

Y el componente " padre " o que instancie a este componente seria de este tipo

```
import { Component } from '@angular/core';
import { InputComponent } from 'input.component';
@Component({
  selector: 'home-component',
  template: `
    <h1> Hola </h1>
    <ul *ngFor="let id of ids">
      <input-component entrada="id"></input-component>
    </ul>
  `
})

export class HomeComponent{

  ids = [
    id : 1,
    id : 2,
    id : 3,
    id : 4,
    id : 5
  ];

  constructor(){}
}
```

El resultado de esto, va a ser un html con una lista del 1 al 5, y en la consola vamos a ver lo mismo, un console log por cada iteración del ngFor.

Es importante saber que podemos poner tantos Inputs como NECESITEMOS.

El annotation Input puede traer cualquier tipo de cosa, desde un *string* un *any* o un *objeto* completo.

Output

Dentro de los componentes "Hijos" existen entradas (los inputs), y tambien existen eventos de salida, que al pasar algo, el componente hijo, le enviara datos al componente "Padre".

Para hacer esto vamos a hablar de los Outputs, estos outputs son "EventEmmitter" que van a ejecutar una funcion del componente "padre" para enviarle un dato al mismo, tal y como hablamos en los Inputs, los parametros que pasemos, pueden pasar desde un **any** hasta un **objeto**.

Vamos a ver como funciona, para entender un mejor

Componente Hijo:

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

@Component({
  selector : 'output-component',
  template: `
    <span (click)="me()">Click ME<span>
  `
})
export class OutputComponent{

  @Output() outputEmitter = new EventEmitter<string>();

  me():void{
    this.ouputEmitter.emit("Hey te estoy enviando un msj");
  }
}
```

Componente Padre:

3.3 Output

```
Import { Component } from '@angular/core';
@Component({
  selector: 'padre-component',
  template: `
    <output-component (ouputEmiter)="ouputEmiter($event)"></output-component>
  `
})
export class PadreComponent{

  ouputEmiter(miVar:string){
    console.log("el valor que vino es: ", miVar);
  }

}
```

Entonces para recapitular, cuando creas un "Output" lo que vas a generar es un event emitter que lo que hace es enviar un valor del hijo al padre... Recuerda que este valor puede ser desde un string , un any, hasta un objeto completo.

Constructor

El constructor es Algo muy importante sobre un componente, es quien lo va a crear, esto viene de la mano de la teoria de los objetos... pero vamos a repasar un poco y si no la conoces vamos a entenderla.

Un Objeto es una Clase instanciada ... Osea que un "class" cuando le hacemos un "new" genera un objeto de esa clase, entonces con los componentes es algo parecido... cuando arrancamos nuestro proyecto Angular instancia un Componente para renderizar la vista (eso se indica en el NgModule).

Entonces una clase cuando es instanciada, lo primero que se ejecuta es el constructor, quien "construye" lo que genera el formato inicial del objeto.

Para esto lo vamos a ver en un ejemplo.

```
class MyPerson(){  
  
    var nombre:string;  
    var apellido:string;  
  
    constructor(_nombre:string, _apellido:string){  
        this.nombre = _nombre;  
        this.apellido = _apellido;  
    }  
}  
  
var Jorge = new MyPerson("Jorge", "Cano");  
  
var JorgeDos = new MyPerson(); => //da error, ya que el constructor pide DOS parametro  
s
```

Entonces ahora que entendimos que un objeto se instancia a base del contructor, vamos a entender un poco mejor como se generan con los constructores de los componentes de angular.

Esto mismo pasa en los componentes de Angular, lo que quiere decir, es que Angular entiende que si importamos ciertos modulos, y los ponemos en nuestros constructores.

Vamos a ver un ejemplo en un componente de Angular con el modulo Router

3.4 Constructor

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

@Component({
  selector: 'app-component',
  templateUrl: 'app.component.html'
})

export class AppComponent{

  constructor(private router:Router){
  }
}
```

Ahora cuando Angular cree "AppComponent" lo que va a pasar es que va a tener que enviarle si o si la funcionalidad Router (que importamos arriba) que a su vez tuvimos que tener importada en el NgModule.

Entonces cuando crea el componente lo que va a pasar es que le va a pasar ese parametro, ahora veamos que en el codigo a diferencia de las funciones normales le agregue un "private" adelante.

Lo que va a hacer un "private" o "public" en el constructor es declarar una variable privada o publica dentro del componete, en el caso de que no la hagamos tendríamos que asignarla a una variable que nosotros hayamos creado antes.

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

@Component({
  selector: 'app-component',
  templateUrl: 'app.component.html'
})

export class AppComponent{
  router:Router;

  constructor(_router:Router){
    this.router = _router;
  }
}
```

En ambos codigos logramos lo mismo, la costumbre y la documentación indica que lo deberíamos hacer como en el primer código indicado.

Con esto ya entendemos como Angular declara por dentro cada uno de los componentes, como utiliza sus constructores y como pasar parametros a la creación.

Ahead-of-time(AOT) vs Just-in-time(JIT)

Cuando trabajamos con Angular podemos compilar la aplicación en el navegador, en tiempo de ejecución, a medida que se carga la aplicación, utilizando el compilador Just-in-Time (JIT). Este es el enfoque de desarrollo estándar que se muestra en toda la documentación de la web de Angular (angular.io).

La compilación JIT tiene una penalización de ejecución. Las vistas tardan más en procesarse debido al paso de compilación dentro del navegador. La aplicación es más grande porque incluye el compilador angular y un montón de código de biblioteca que la aplicación realmente no necesita. Las aplicaciones más grandes tardan más en transmitir y son más lentas de cargar.

La compilación puede descubrir muchos errores de vinculación de plantilla de componentes. JIT compilación los descubre en tiempo de ejecución que es más tarde de lo que nos gustaría.

El compilador AOT (Ahead-of-Time) puede detectar errores de plantilla con anticipación y mejorar el rendimiento al compilar en el momento de la compilación.

Con AOT, el compilador se ejecuta una vez en el tiempo de construcción utilizando un conjunto de bibliotecas.

Con JIT se ejecuta cada vez para cada usuario en tiempo de ejecución utilizando un conjunto diferente de bibliotecas.

¿Por qué AOT compilación?

Rendimiento más rápido

- El navegador descarga una versión **pre-compilada** de la aplicación. El navegador carga código ejecutable para que pueda procesar la aplicación inmediatamente, sin esperar a compilar la aplicación primero.

Menos solicitudes asíncronas

- El compilador "inlines external" plantillas de html y hojas de estilo css dentro de la aplicación JavaScript, eliminando las solicitudes separadas de ajax para esos archivos de origen.

Menor tamaño de descarga de Angular

- No es necesario descargar el compilador Angular si la aplicación ya está compilada. El compilador es aproximadamente la mitad de Angular, por lo que omitirlo reduce drásticamente la carga útil de la aplicación.
- **Detectar errores de plantilla con anterioridad**
- El compilador AOT detecta e informa errores de enlace de plantilla durante el paso de compilación antes de que los usuarios puedan verlos.

Mejor seguridad

- AOT compila plantillas HTML y componentes en archivos JavaScript mucho antes de que se sirvan al cliente. Sin plantillas para leer y ninguna evaluación de HTML o JavaScript cara al cliente, hay menos oportunidades para ataques de inyección.

Como funcionan los navegadores



By @addyosmani

← En respuesta a Addy Osmani



Addy Osmani @addyosmani · 9 h

@addyosmani ^ this was also my 1st attempt at channeling my inner @kosamari to explain how V8 will work. My drawing skills aren't great :P pic.twitter.com/bHSf3uLgG3

<https://twitter.com/addyosmani/status/829728691798188034>

Si, **WOW** es valido... hace muchas mas cosas de las que pensabas no ??? hace todo eso para levantar nuestro JS ... entonces hacer la pre-pasada (mas la parte de compilación de angular) nos ahorra tiempo y espacio... de hecho, en el CLI por defecto va a tratar de hacer la compilación con AOT.

Aclaración: Este capitulo es casi un clon de la documentación original de angular, ya que es complejo y esta muy bien explicado

(<https://angular.io/docs/ts/latest/cookbook/aot-compiler.html>)

Aclaración 2: En mi caso siempre que pueda elijo hacer AOT pero no siempre va a pasar eso, hay ciertos modulos que no soportan AOT, lo que si, cada vez que utilices el CLI va a tratar de compilar por defecto en AOT.

NgModule

Capaz que una de las grandes mejoras en las betas y RC, fue el manejo del NgModule ... Este es quien maneja los hilos de la aplicación, es quien lleva la batuta , basicamente es quien orquesta a nuestra aplicación en Angular.

NgModule es la unidad de compilación y distribución de los componentes y pipes de Angular. En muchas cosas es similar a los modulos de ES6, porque tenemos declaraciones, imports, y exports.

Para ello vamos a ver como esta compuesta y para que sirve cada uno de los datos que recibe:

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    RouterModule.forRoot(APPROUTER)
  ],
  providers: [myService],
  bootstrap: [InitComponent]
})
```

Esta declaración que vemos arriba es lo mas "comun" o inicial que podriamos ver...

NgModule es un decorador que se utiliza en **app.module.ts** @NgModule toma un objeto de metadatos que indica a Angular cómo compilar y ejecutar código de módulo.

Vamos a ir viendo paso a paso como funciona y que requiere cada función.

- **Declarations:** este arreglo sirve para incluir todos los **Componentes** que creamos a lo largo de nuestro desarrollo para poder utilizarlo, si no declaramos nuestro componente, no vamos a poder utilizarlo, ya sea por el router o por llamarlo como un tag especifico. Tambien vamos a declarar nuestras **Directivas**
- **Imports:** este arreglo va a tener "normalmente" lo que contiene *Module , lo que quiere


decir es que normalmente todo lo que termina con Module va a estar ahí, normalmente son librerías y funcionalidades CORE, acá va a estar la composición de la funcionalidad que vamos a utilizar... Por ejemplo, RouterModule lo importamos para poder utilizar el router de Angular, y con el forRoot le indicamos un tipo de objeto que tiene las rutas de cada uno de los componentes.

- **Providers:** aquí tendremos otro arreglo en el cual vamos a poner todos nuestros servicios, nuestros providers, todo lo que contenga "@Injectable", estos servicios se levantan en el momento que se crea la APP en el navegador y son singletons, esto quiere decir que se instancian en una porción de memoria y se mantienen ahí hasta que se termine la app.
- **Bootstrap:** Este arreglo va a tener nuestro componente inicial, nuestro "Init" por decirlo de alguna manera... donde vamos a tener el router por ejemplo... quien sea el contenedor inicial para nuestra aplicación.

El **NgModule**, cuando entro fue algo raro... antes estábamos acostumbrados a hacer otro tipo de cosas...

(Aquí está el código del bootcamp que hicimos antes de que apareciera el NgModule <https://github.com/jorgeucano/bootcamp>)

Pero esto sirve para organizarnos un poco... realmente si no funciona los errores se entienden, y el NgModule (y los IDE's y Editores de texto) ya saben que te falta y te lo marcan antes de que hagas nada.. te avisan =>



```
@Component({
  [Angular] Component 'MyComponent' is not included in a module and w
  ill not be available inside a template. Consider adding it to a NgM
}
odule declaration
e
(alias) Component(obj: Component): TypeDecorator
}
import Component
```

Entonces de esta forma vamos a estar mas "contenidos" a la hora de escribir nuestro código y sentir que podemos avanzar sin tener tantos problemas con el NgModule, digo esta claro el "error" que nos está mostrando... **Consider adding it to a NgModule declaration** ... hace falta mas ???

Angular Modules help organize an application into cohesive blocks of functionality

"... ayudan a organizar una aplicación en bloques cohesivos de funcionalidad" .. esta claro, nos ayudan a ser cohesivos y funcionales...

Por lo tanto, este archivo lo vamos a tener abierto a lo largo de nuestro desarrollo (algunas veces nos dara dolor de cabeza) pero sabemos que si este archivo no esta actualizado la app no funciona :-D ... primer punto para ver ...

Directivas

Las directivas son "atributos" que vamos a agregar a nuestro código...

Alguna vez usaron algún plugin o librería que simplemente con agregar una palabra a nuestros input validaba algo, o creaba las letras de distinto color ???

NO? ENSERIO?

Bueno, supongamos que nunca nadie utilizó una librería media rara que hiciera cosas por nosotros... y vamos a ver que es una directiva y cómo crear "directivas custom".

Para ello vamos a ver que es una directiva... recuerden que todo el código lo tenemos en el Git.

Primero les voy a mostrar el resultado (si es llamativo a propósito) :



Ahora vamos de a poco.. arranquemos por el html

```
<h1 [style.background]='lime' title-blue>
  {{title}}
</h1>
```

Bien, acá vemos claramente que "cambio" un poco nuestra UI, agregamos con unos **corchetes unos parametros que cambiaron nuestro background ...**

En realidad, los corchetes nos sirven para "bindear" lo que ponemos adentro con Angular... lo que hicimos es una "directiva" común.. con un "Atributo" cambiamos nuestro background... lo que hacemos con esto es pasarle los parametros a nuestra webApp y cambiarle los parametros a nuestro gusto...

[] => le genera una referencia al elemento completo

style.background => hace referencia a lo que quiere obtener del elemento que obtuvimos.

`=" 'lime' "` => le asigna el color "lime" ... aca podriamos tener una variable bindeada para que haga el cambio que queramos dependiendo que vamos haciendo... pero con esto podemos ver que la directiva, obtiene una referencia del elemento en el que esta y luego modifica sus parametros para hacer un "custom" de lo que necesitamos.

No queda super lindo hacerlo así, ademas imaginate si lo tuvieras que hacer constantemente, se vuelve algo super tedioso... por lo tanto ahi aparece otra función que podemos utilizar...

Directivas personalizadas (CUSTOM Directives)

Las **directivas personalizadas** lo que hacen es generar directo una funcionalidad en base a un "atributo" ... en el caso de nuestro ejemplo es **title-blue**, pero claro nos falta un poco de código para poder entender esto...

title.blue.directive.ts

```
import { Directive, ElementRef, Input } from '@angular/core';
@Directive({ selector: '[title-blue]' })
export class TitleBlueDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.fontSize = '50px';
    el.nativeElement.style.color = 'blue';
  }
}
```

Ahora si!! nos faltaba lo mas importante... ver porque tenia el tamaño y el color... aca esta, SI!!! lo se, es parecido al otro cierta parte... pero vamos a ver paso a paso.

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

Vamos a importar las cosas que necesitamos para trabajar...

Directive lo necesitamos para el Decoration, es quien va a definir que esto es una directiva (tal como el de component, define que es un componente)...

ElementRef : es la refencia del elemento completo .. algo asi como hacer un "document.getElementById('ID')" y asignar eso a una variable...

Input : Permite que los datos fluyan de la expresión vinculante a la directiva...

```
@Directive({ selector: '[title-blue]' })
```

Aquí tenemos la directiva y el selector que decidimos darle.. esto luego se convierte en **title-blue** dentro del html

La declaración de la clase la voy a dejar de lado (ya deberias entender 100% que hace).

```
constructor(el: ElementRef)
```

En el constructor estamos trayendo la referencia del elemento, de esto se encarga Angular por nosotros, para que podamos manipularlo a nuestro gusto.

```
el.nativeElement.style.fontSize = '50px';  
el.nativeElement.style.color = 'blue';
```

Y por ultimo, la manipulación final... aquí tenemos como vamos a modificar el element.. con **nativeElement** vamos a generar lo mismo que hicimos con los **[]** (los corchetes) en el html directo.

Luego el nativeElement es la asignación de la que hablamos... que va a tomar los "atributos" nativos del elemento... para entenderlo de una forma mas "facil" serian "los que podemos modificar por CSS" por ejemplo.

Y por ultimo tomamos el Style, para hacer la modificación del font-size y el color ...

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

import { AppComponent } from './app.component';

//directivas custom
import { TitleBlueDirective } from './directives/title.blue.directive';

@NgModule({
  declarations: [
    AppComponent,
    // directives
    TitleBlueDirective
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Por ultimo tenemos nuestro ya "amado" **app.module.ts** en el cual **importamos** nuestra directiva y la agregamos a **declarations**.

De esta forma vamos a poder generar Directivas a nuestra necesidad... recuerden que no tiene sentido escribir muchas reglas o cosas que vayamos replicando por todos lados... nuestros componentes son "encapsulados" por lo tanto o vamos a tener que agregar css a nuestro archivo global y luego copiar lo que necesitamos en el "codigo" .. o directamente podemos crear nuestra directiva para poder **aprovechar el maximo poder de Angular**.

Componentes avanzados

Ya estuvimos hablando de los componentes, y sabemos que son la gran mayoría de lo que vamos a escribir... porque deberíamos pensar en componentes.. Angular es un arbol de componentes, por lo tanto vamos a ver un poco mas a fondo todavia los componentes y entender un poco mas su integración.

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

@Component({
  selector: 'login',
  templateUrl: 'login.component.html'
})
export class LoginComponent{

  //constructor es quien recibe la inyeccion de las dependencias => capítulo 9
  constructor(){

  }

  // lifecycle => capítulo 10
  ngOnInit(){
  }
  ngOnDestroy() {
  }

}
```

Vimos como crear un componente, agregarle entradas, salidas... pero hay mucho mas todavía.. vamos a ver/pensar todo lo que podemos hacer con un Componente...

Primero , el componente principal va a ser quien cargue la aplicación,

- Por medio de los lifecycle vamos a poder modificar cosas (o determinar si se abre o no)
- Por medio de router vamos a poder abrir componentes dentro de otros (sin que sean childcomponents, expecificamente)
- Vamos a poder pasar datos entre componentes padres e hijos (cuantos tengamos en la rama)
- Vamos a poder manejar distintos tipos de estados
- Vamos a poder pasar datos por medio de la Inyección de dependencias (servicios)
- Generar funcionalidad encapsulada

- manejo de formularios
- Manejo de eventos en la UI
- Detección de cambios (AngularZone)

Claro podemos hacer muchisimas cosas y todo eso gracias a toda la funcionalidad de Angular, y para eso vamos a tener que avazar con los capitulos ... pero queria mostrar como quedaria un componente super completo, mas alla que podrias no entender/saber algunas cosas de las que vamos a ver ahora, pero si podrias identificarlas.

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

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

import { Store } from '@ngrx/store';
import { INCREMENT, DECREMENT, RESET } from '../services/counter';
import { Observable } from 'rxjs/Observable';

import { Router } from '@angular/router';

import { AngularFire, FirebaseListObservable } from 'angularfire2';
import { AuthProviders, AuthMethods } from 'angularfire2';

interface AppState{
  counter: number;
}

@Component({
  selector: 'app-component',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';
  votacion = '';
  data:any;
  myForm : FormGroup;
  counter: Observable<number>;
  dataFirebase:any;
  errorMessage:any;

  constructor(
    private fb: FormBuilder,
    private store: Store<AppState>,
    private _ngZone:NgZone,
    private router: Router,
    private af : AngularFire
  ){

    this.callTicketsMongo();

    this.ticketFirebase = af.database.list('/ticket');
```

```
this.dataFirebase.push({
  'id': 1, 'texto': 'no me funciona nada', 'estado': 'enojado'
});

this.af.auth.login({
  provider: AuthProviders.Google,
  method: AuthMethods.Popup
});

this.counter = store.select('counter');

this.tickets = ticketService.getTickets();

this.myForm = fb.group({
  'name': ['Jorge']
});
}

callTicketsMongo():void{
  this.ticketService.getTicketsMongo()
    .then(
      tickets => this.ticketMongo = tickets,
      error => this.errorMessage = <any>error
    );
}

votos = [
  {title : 'Opción 1'},
  {title : 'Opción 2'},
  {title : 'Opción 3'},
  {title : 'Opción 4'}
];

addVoto(response: string) {
  this.votacion = "Usted eligio: " + response;
}

cantidad = 5;
factor = 1;

onSubmit(value:string):void{
  console.log("El formulario tiene", value);
}

increment(){
  this.store.dispatch({type: INCREMENT});
}

decrement(){
  this.store.dispatch({type: DECREMENT});
}
```

```
}
reset(){
  this.store.dispatch({type: RESET});
}

progress: number = 0;
label:string;

processWithinAngularZone(){
  this.label="inside";
  this.progress = 0;
  this._ngZone.runOutsideAngular(()=>{
    this._increaseProgress(()=>{
      this._ngZone.run(()=>{console.log("Finalizado con ngzone")})
    })
  })
}
_increaseProgress(doneCallBack: ()=>void){
  this.progress+=1;
  console.log(`Progreso: ${this.progress}%`);
  if(this.progress<100){
    window.setTimeout(()=>{
      this._increaseProgress(doneCallBack);
    },10);
  }
  else{
    doneCallBack();
  }
}

verDatos(id:number):void{
  this.router.navigate(['/ticket', id ]);
}

updateDatos(key):void{
  console.log(key);
  this.datosFirebase.update(key, {estado: 'avanzado'});
}

removeDatos(key):void{
  console.log(key);
  this.datosFirebase.remove(key);
}

removeAllTicket():void{
  this.datosFirebase.remove();
}
}
```

Claro esto se esta poniendo interesante (a decir verdad, este componente hasta podria ser "chico" comparado con otros), hay muchas mas funcionalidades en esta porción de código, pero no siempre es necesario tener todo ahi... **seguramente mucho lo podriamos abstraer en otro componente**... por lo tanto es importante entender lo siguiente:

Los componentes deben cumplir una función única, si tiene mas de una se pueden hacer dos componentes (uno podria ser hijo del otro)

Importante: Este capítulo va a estar siempre en modo WIP (work in progress) ya que angular va a ir avanzado y este capítulo va a estar para esos avances, la idea de este capitulo es que entiendan que apartir de ahora vamos a agregar un monton de funcionalidad que puede estar en un solo componente o entender que podemos utilizar la "atomicidad" y poder reutilizar todos los componentes que realicemos para nuestros desarrollos.

Router

Si hay algo a lo que se le dio mucha vida y lineas de código , y reescritura... Sin NINGUNA duda fue el Router... Angular paso por 3 versiones de router, mas una de NgRX, para poder tener la que tenemos ahora ... de hecho si lo recuerdan Angular en Marzo del 2017 pasa de la version 2.x.x a la 4... para mantener un "hilo" con el router...

Ahora si, luego de tantas "idas y vueltas" con el router y de aprender unos cuantos, yo me habia quedado con el router de NgRX, tenia componentes reactivos y para todo se utilizaban observables... era super entretenido... pero con la version del router final, que creo Victor Savkin (@victorsavkin), cambiaron muchas cosas, y hizo que vuelva a darle un vistazo al "nuevo" router, que estaba dentro de la capa del "core" (lo pongo entre comillas porque esta en otro paquete, mas alla que es como parte del core) entonces encontré que con el "nuevo" router se podían hacer muchas mas cosas que simplemente "si cambias a esta url => carga este componente".

Realmente parecia poco decir que al cambiar la URL cambia el Componente, entre cada una de las cosas que hace el router para ver sus funcionalidades un poco mas a fondo.

Entendiendo los pasos del router:

- El router de Angular toma la URL y "ejecuta" las siguientes "funcionalidades":
- 1. Aplica la redirección
 2. Reconoce el estado del router
 3. Ejecuta "guards" y resuelve datos
 4. Activa todos los componentes necesarios
 5. Administra la navegación para un futuro cambio.

SI!!! hace estas cosas cada vez que cambiamos una URL interna (dentro de la aplicación) ... Claro que uno "tiene idea" de lo que pasa .. pero no sabe todo lo que va ejecutando por atras el router cada vez que cambiamos la url... por lo general nosotros no nos preocupamos por esto (esta bien) pero no me parece mal que se sepa. Recuerden que el proposito es que entiendan como funciona angular, al entender como funciona, van a poder programar mas comodis y entender rapidamente como resolver ese bug o como realizar esa feature que se acerca.

Formato de la URL:

/usuarios/10(popup:compose)

/usuarios/

/usuarios/10;open:true/tickets/resueltos

Como se puede ver, el router utiliza paréntesis para serializar segmentos secundarios (por ejemplo, popup: compose), la sintaxis de dos puntos para especificar la salida y la sintaxis 'parameter = value' (por ejemplo, open = true) para especificar parámetros específicos de ruta .

Configuración:

Lo primero que vamos a hacer cuando queramos programar nuestras funcionalidades dentro del router va a ser su configuración y lo primero (siempre se puede ir modificando) es crear nuestras rutas.

```
export const ROUTERCONFIG = [
  { path: '', pathMatch: 'full', redirectTo: '/home' },
  {path: 'home', component: HomeComponent },
  {
    path: ':id',
    children: [
      { path: '', component: UsuariosComponent },
      {
        path: ':id',
        component: UsuarioComponent,
        children: [

          { path: 'tickets', component: TicketsComponent },

          { path: 'tickets/:status', component: TicketComponent }

        ]
      }
    ]
  },
  {
    path: 'compose',
    component: ComposeTicketComponent,
    outlet: 'popup'
  },
  {
    path: 'tickets/:status',
    component: PopupTicketComponent,
    outlet: 'popup'
  }
];
```

Como funciona una redirección:

Redirección: Un redireccionamiento es una sustitución de una "parte" de la URL. Los redireccionamientos pueden ser **locales o absolutos**. Los redireccionamientos locales reemplazan una sola "parte" por una diferente. Los redireccionamientos absolutos reemplazan toda la URL. **Los redireccionamientos son locales a menos que prefijemos una url con una barra.**

Modificar URL: Cuando un usuario o función cambian la URL, el router obtiene la modificación, lo PRIMERO que hacer el router es modificar el texto de la URL para aplicar el funcionamiento de la redirección.

Reconocer el estado: para esto el router va a tener que reconocer la URL que acaba de modificar, entendiendo desde donde arranca y que contiene que "checkear" en la configuración si existe tal y que debe hacer (o que debe hacer en el caso de que no exista) para esto deberiamos entender que es cada parte de las URLS (rutas).

Que es cada una de las "cosas" que tiene una URL para el router y para que las configuramos, para ello el Router hace un "forEach" (recorre todos las posibilidades de las configuraciones del router cuando se modifica la url).

":id" => todo lo que contenga dos puntos adentro es un "segmento variable" , lo que quiere decir es que el router ahi sabe que puede venir algo directamente (y que luego tiene que seguir haciendo match con lo que continua de la configuración)

Parametros fijos => son aquellos que estan "fijos" por ejemplo en nuestro caso ... luego de recibir :id esperamos que tenga un "tickets" o "ticket/:status" o sea que si viniera "tickets" seguramente seria un "listado" con todos los tickets... entonces el parametros "tickets" o "ticket" son fijos, y estos hacen que el router pueda entender para donde debe ir.

Si el camino tomado a través de la configuración no "consume" toda la URL, el router retrocede para intentar una ruta alternativa. Si es imposible emparejar toda la URL, la navegación fallará. Pero si funciona, se construirá el estado del enrutador que representa el estado futuro de la aplicación.

Router State: Un estado de enrutador consiste en rutas activadas. Y cada ruta activada puede estar asociada con un componente. Además, tenga en cuenta que siempre tenemos una ruta activada asociada con el componente raíz de la aplicación. O sea basicamente,

son las rutas que existen y se pueden acceder, osea es la conexion entre la URL y el Componente, con la aclaracion que vamos a poder darle un componente a nuestra URL raiz.. que podria ser por **ejemplo** `www.ejemplo.com` o `www.ejemplo.com/myAngularApp/`

Ejecuta los "Guards": el router chequeara que el estado actual de la aplicación tiene "permisos" para poder pasar al siguiente estado... Esto lo hara con los **Guards** (con estos Guards podremos manejar los "permisos" de la navegación, se puede usar para seguridad, autorización o propósitos de monitoreo entre otras cosas).

Resuelve la "data": ahora por ultimo lo que le queda es resolver los datos que obtiene

```
constructor(route: ActivatedRoute)
```

En nuestro constructor vamos a tener que inyectar (si ! lo se, no lo vimos todavía, es justo el siguiente capítulo) el "ActivatedRoute" y con el vamos a poder obtener los parametros/datos enviados

```
this.tickets = route.data.pluck('tickets');
```

Activación del componente: ahora que ya ejecuto todo lo anterior esta en posición para activar el componente que necesita, para ello recordemos que ya creamos todo lo necesario para poder utilizarlo.

hasta aca todo muy lindo pero todo esto en donde se va a encontrar ?? donde se va a ejecutar ?

Bien, para poder ejecutar el router vamos a tener que hacer algunas cosas mas que simplemente tener un Array de objetos que entiende el router..

Para ello voy a explicar aquí en el codigo como hacer ...

Primero vamos a configurar bien nuestro "config":

/commos/router.ts

```

import { HomeComponent } from '../components/home/home.component';
import { UsuariosComponent } from '../components/usuarios/usuarios.component';
import { UsuarioComponent } from '../components/usuarios/usuario.component';
import { TicketsComponent } from '../components/tickets/tickets.component';
import { TicketComponent } from '../components/tickets/ticket.component';
import { ComposeTicketComponent } from '../components/tickets/compose.ticket.component';
import { PopupTicketComponent } from '../components/tickets/popup.ticket.component';

export const ROUTERCONFIG = [
  { path: '', pathMatch: 'full', redirectTo: '/home' },
  { path: 'home', component: HomeComponent },
  {
    path: ':id',
    children: [
      { path: '', component: UsuariosComponent },
      {
        path: ':id',
        component: UsuarioComponent,
        children: [
          { path: 'tickets', component: TicketsComponent },
          { path: 'tickets/:status', component: TicketComponent }
        ]
      }
    ]
  },
  {
    path: 'compose',
    component: ComposeTicketComponent,
    outlet: 'popup'
  },
  {
    path: 'tickets/:status',
    component: PopupTicketComponent,
    outlet: 'popup'
  }
];

```

Aquí como vimos tenemos la configuración, vamos a tener que importar los componentes para que lo podamos utilizar en nuestra config.

Por otro lado vamos a tener que configurar nuestros modulos, Angular Router, es una dependencia dentro de angular que esta "separada" dentro de las dependencias de core.. por lo tanto la vamos a llamar como **@angular/router**.

app.module.ts

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

import { AppComponent } from './app.component';

// components
import { HomeComponent } from './components/home/home.component';
import { UsuariosComponent } from './components/usuarios/usuarios.component';
import { UsuarioComponent } from './components/usuarios/usuario.component';
import { TicketsComponent } from './components/tickets/tickets.component';
import { TicketComponent } from './components/tickets/ticket.component';
import { ComposeTicketComponent } from './components/tickets/compose.ticket.component';
;
import { PopupTicketComponent } from './components/tickets/popup.ticket.component';

//router
import { ROUTERCONFIG } from './commons/router';
import { RouterModule } from '@angular/router';

//directivas custom
import { TitleBlueDirective } from './directives/title.blue.directive';

@NgModule({
  declarations: [
    AppComponent,
    // directives
    TitleBlueDirective,
    // coponents
    HomeComponent,
    UsuariosComponent,
    UsuarioComponent,
    TicketsComponent,
    TicketComponent,
    ComposeTicketComponent,
    PopupTicketComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot(ROUTERCONFIG)
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Ahora si, ya importamos nuestro modulo , y con "forRoot" vamos a darle una configuración inicial, que es la que creamos anteriormente... lo que nos queda es crear los componentes, como son dummies todos, voy a mostrar donde muestra nuestro ruteador.

Para eso tenemos que tener algun lado donde se genere el mismo, el router corre bajo un tag, que en nuestro caso vamos a llamar dentro de nuestro componente principal:

app.component.html

```
<h1 [style.background]='lime' title=blue>
  {{title}}
</h1>
<router-outlet></router-outlet>
```

En este nuevo tag "router-outlet" vamos a poder ver los que cargue nuestro Router... entonces ya con todo esto, cuando arranquemos nuestra app, debería levantar el componente que utilizamos como Home ...

/components/home/home.component.ts

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

@Component({
  selector: 'home',
  template: '<h1> Home </h1>'
})
export class HomeComponent {}
```

Y como resultado tenemos:



Home

Para ver todo el código pueden verlo en github, en el paso "router-01".

Inyección de Dependencias

Uno de las grandes novedades de Angular (entre otras cosas) es la inyección de dependencias, este patron (https://es.wikipedia.org/wiki/Inyección_de_dependencias) es un patron de diseño orientado a objetos.

La IDEA de la ID (Inyección de dependencias) es muy simple , si tenes un componente que depende de un servicio, no creas ese servicio (en el componente) si no que en el constructor lo solicitas y Angular te lo trae.

De esta forma, el codigo esta desacoplado, ya que el que el servicio se encarga de lo que debe, y el componente solamente lo consume.

Esto nos sirve para poder testear como se debe y que sea mas facil.

Angular viene con un sistema de inyección de dependencia. Para ver cómo se puede usar, veamos el siguiente componente, que muestra una lista de tickets usando la directiva for:

```
@Component({
  selector: 'ticket-component',
  template: `
    <h2>Tickets:</h2>
    <ticket *ngFor="let t of tickets" [ticket]="t"></ticket>
  `
})
class TicketCmp {
  constructor() { //..obtenemos la data }
}
```

Ahora crearemos un mock up para los datos que tenemos que iterar

```
class TicketsMockUp {
  fetchTickets() {
    return [
      { id: 1, text: 'No me funciona la impresora', state: 'open' },
      { id: 2, text: 'No me funciona el mouse', state: 'close' }
    ];
  }
}
```

Ahora lo que necesitamos en nuestro componente es obtener los datos de este servicio...

¿Como lo hacemos?

Creamos una instancia del mismo en nuestro componente, de la siguiente forma:

```
constructor() {  
    const mockUp = new TicketsMockUp();  
    this.tickets = mockUp.fetchTickets();  
}
```

Hasta ahí todo genial para "entender" o hacer una mini demo, pero realmente NO es para producción, para ello deberíamos utilizar lo siguiente =>

```
class TicketCmp {  
    constructor(mockUp: TicketsMockUp) {  
        this.tickets = mockUp.fetchTickets();  
    }  
}
```

Lo que hicimos, fue utilizar la DI para traer nuestro "servicio" (mejor dicho mockup), Angular lo que hizo fue crear una instancia de TicketsMockUp.

Este "provider"/"servicio"/"mockup" que creamos (un obtenedor de datos) tenemos que declararlo, y tenemos dos formas de hacerlo.

La primera es declararlo en nuestro componente

```
@Component({  
    selector: 'ticket-component',  
    template: `  
        <h2>Tickets:</h2>  
        <ticket *ngFor="let t of tickets" [ticket]="t"></ticket>  
    `,  
    providers: [TicketsMockUp]  
})
```

Y la segunda opción es en nuestro NgModule:

```
@NgModule({  
    ....  
    providers: [TicketsMockUp]  
    ....  
})
```

¿Cuál es la diferencia y cuál deberías preferir?

Generalmente, recomiendo registrar proveedores en el nivel de módulo cuando no dependen del DOM, componentes o directivas. Y sólo los proveedores relacionados con la interfaz de usuario que tienen que estar asignados a un componente en particular deben

registrarse a nivel de componente. Como `TicketsMockUp` no tiene nada que ver con la interfaz de usuario, deberías registrarla en el nivel del módulo.

Arbol de Inyección

La inyección de dependencias tiene dos partes:

Registro de providers: Cómo y dónde debe crearse un objeto

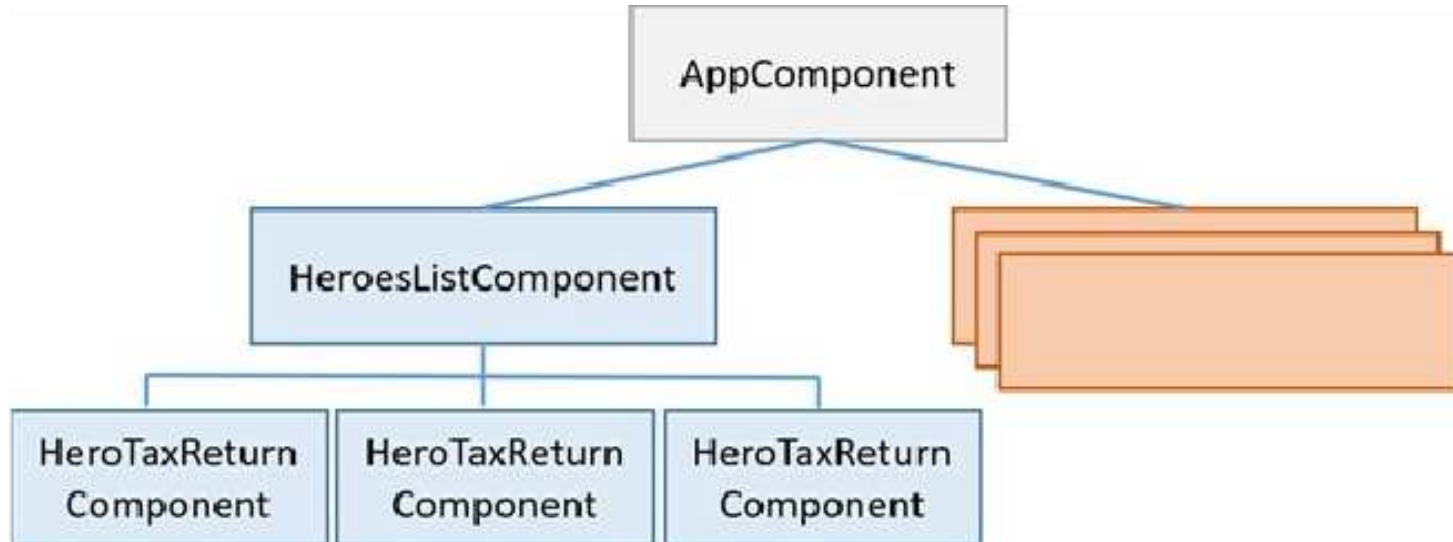
Injectar dependencias: De qué depende un objeto.

Y todo lo que un objeto depende (servicios, directivas y elementos) se inyecta en su constructor. Para hacer este trabajo Angular construye un árbol de inyectores.

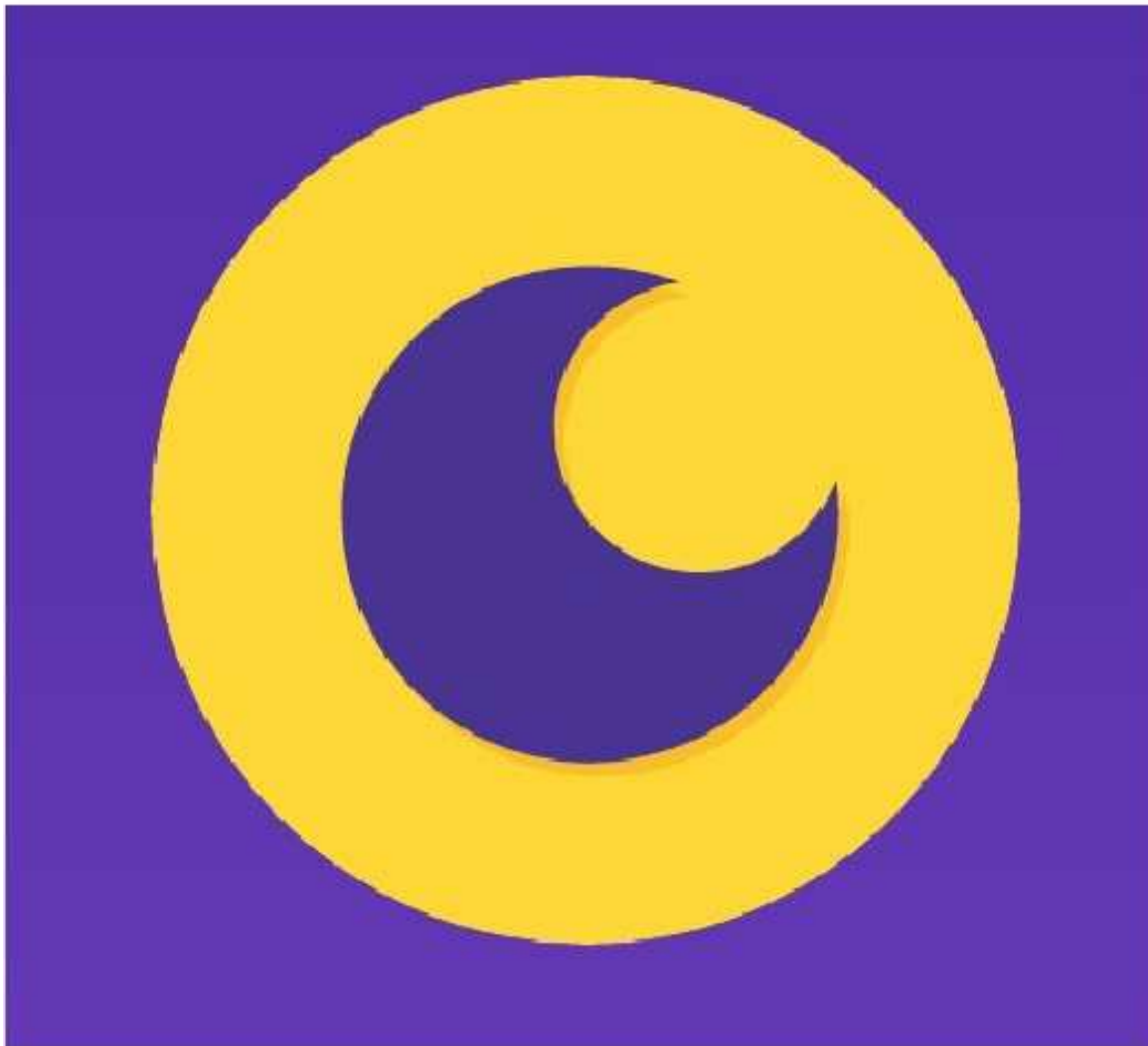
En primer lugar, cada elemento DOM con un componente o una directiva en él recibe un inyector. Este inyector contiene la instancia de componente, todos los proveedores registrados por el componente y algunos objetos "locales" (por ejemplo, el elemento).

En segundo lugar, al iniciar un `NgModule`, Angular crea un inyector usando el módulo y los proveedores definidos allí.

Así, el árbol inyector de la aplicación se verá así:



O también lo podrías ver en el navegador gracias a Augury



(<https://augury.angular.io/>)

Esta herramienta para el chrome nos mostrara el arbol de componentes y dependencias que tengamos en nuestra aplicación.

Ciclos de vida

Los ciclos de vida (LifeCycle) de Angular se manejan a traves de cada uno de los componentes que creemos, para tener un poco mas de idea son HOOKS (un hook es una funcion que se ejecuta en un determinado momento por medio de una acción).

Los hooks que nos ofrece Angular son:

- OnInit
- OnDestroy
- DoCheck
- OnChanges
- AfterContentInit
- AfterContentChecked
- AfterViewInit
- AfterViewChecked

Para utilizarlos vamos a tener que usar un "patron" parecido:

Para ser "notificado" sobre los eventos hay que hacer:

1 - Declarar la directiva o componente e implementar la interfaz

2 - Declarar el metodo (ej ngOnInit)

Todos los metodos contienen el prefix de ng... por ejemplo ngOnInit, ngAfterContentInit ... etc...

Cuando Angular sabe que implementamos en el componente esa función (por los hooks), las invoca en el momento que deben ser ejecutadas.

Ahora no es mandatorio poner el implements, si ya ingresas en tu componente la función angular entiende que la tiene que ejecutar directamente.

OnInit and OnDestroy

El hook OnInit se ejecuta cuando las propiedades de la/el directiva/componente ya han inicializado, y antes de que cualquier de las propiedades de la directiva/componente inicialicen.

Ejemplo

```
@Component({
  selector: 'on-init',
  template: `
    <div>
      Init / Destroy
    </div>
  `,
})
class OnInitComponent{

  ngOnInit():void{
    console.log('on init ejecutado');
  }

  ngOnDestroy():void{
    console.log('on destroy ejecutado');
  }

}
```

Si a esto le agregamos funcionalidad y ruteo vamos a ver en que momento ejecuta los console log.

Esto lo puedes ver en el tag : 'ng-on-init-destroy'

OnChanges

OnChanges es un hook que se llama cuando una o mas propiedades del componente cambian.

El metodo ngOnChanges recibe los parametros que tiene que mirar si cambian.

Para eso se puede ver en el ejemplo dentro del git "ejemplo02"

crea un proyecto nuevo con "Angular 4" en el cli ... e hice las siguientes modificaciones:

En el html del componente principal:

```
<h1>
  {{title}}
</h1>
<button (click)="changeTxt()"> Cambiar texto </button>
<home [textoNuevo]="textChange"></home>
```

En el componente de componente principal

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app works!';

  textChange = 'Hola soy un texto';

  changeTxt(){
    this.textChange = 'Acabo de cambiarme';
  }
}
```

Claro, ya sabes que hice, tengo un componente que es input y lo vamos a usar para cambiar el texto y ver las diferencias... simplemente en el componente padre vamos a cambiar el texto con el boton.

Ahora veamos donde esta la magia, el componente hijo:

```

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

import { SimpleChanges } from '@angular/core';

import { Input } from '@angular/core';

import { OnChanges } from '@angular/core';

@Component({
  selector: 'home',
  templateUrl: 'home.cmp.html'
})
export class HomeComponent implements OnChanges{

  @Input() textoNuevo:string;

  firstTxt:string = '';
  secondTxt:string = '';

  ngOnInit(){
    this.firstTxt = textoNuevo;
  }

  changeLog = [];

  ngOnChanges(changes: SimpleChanges) {
    for (let propName in changes) {
      let chng = changes[propName];
      this.secondTxt = JSON.stringify(chng.currentValue);
    }
  }

}

```

Ahora si, entendamos un poco el código.. El hook es la funcion "ngOnChanges" quien es la que ejecuta el cambio, luego la funcion recibe un parametro (usaremos change) que es del tipo SimpleChanges => este lo importamos arriba, lo que va a hacer eso, es tomar el cambio y darnos el contenido del cambio para que podamos trabajar en el.

Luego recorreremos los cambios para mostrar el contenido anterior y el contenido actual, en nuestro html vamos a ver lo siguiente:

```

<h1> Home Component </h1>

<p *ngFor="let item of changeLog">
  {{item}}
</p>

```


Nada que tengamos que explicar a esta altura, ahora veamos lo que obtenemos:

app works!

Cambiar texto

Home Component

**textoNuevo: currentValue = "Hola soy un texto",
previousValue = {}**

Y luego de apretar el boton:

app works!

Cambiar todo

Home Component

**textoNuevo: currentValue = "Hola soy un texto",
previousValue = {}**

**textoNuevo: currentValue = "Acabo de
cambiarme", previousValue = "Hola soy un texto"**

DoCheck

El sistema de notificación predeterminado implementado por OnChanges se activa cada vez que el mecanismo de detección de cambios en angular advierte que hubo un cambio en cualquiera de las propiedades de directiva.

Sin embargo, puede haber momentos en que la sobrecarga agregada por esta notificación de cambio puede ser demasiado, especialmente si el rendimiento es una preocupación.

Puede haber momentos en los que sólo queremos hacer algo en caso de que un elemento haya sido eliminado o agregado, o si sólo una propiedad particular cambió.

AfterContentInit

Este Hook se llama después de OnInit, justo después de la inicialización del contenido del componente o directiva ha terminado

AfterContentChecked

Funciona de manera similar, pero se llama después de la verificación de la directiva ha terminado. La comprobación, en este contexto, es la comprobación del sistema de detección de cambios.

Los otros dos hooks: **AfterViewInit** y **AfterViewChecked** se activan justo después del contenido de arriba, justo después de que la vista se haya inicializado completamente. Estos dos hooks sólo son aplicables a los componentes y no a las directivas.

```
@Component({  
  selector: 'afters',  
  template: `  
    <div class="ui label">  
      <i class="list icon"></i> Counter: {{ counter }}  
    </div>  
  
    <button class="ui primary button" (click)="inc()">Increment
```

```
</button>
```

```
,
```

```
})
```

```
class OtroCmp implements OnInit, OnDestroy, DoCheck, OnChange, AfterContentInit, After
ContentChecked,
    AfterViewInit, AfterViewChecked){

    countador: number;
    constructor() {
        console.log('constructor');
        this.counter = 1;
    }

    inc(){
        console.log('inc');
        this.counter += 1;
    }

    ngOnInit(){
        console.log('ng on init');
    }

    ngOnDestroy(){
        console.log('ngOnDestroy');
    }

    ngDoCheck(){
        console.log('ngDoCheck');
    }

    ngOnChanges(){
        console.log('ngOnChanges');
    }

    ngAfterContentInit(){
        console.log('ngAfterContentInit');
    }

    ngAfterContentChecked(){
        console.log('ngAfterContentChecked');
    }

    ngAfterViewInit(){
        console.log('ngAfterViewInit');
    }
    ngAfterViewChecked(){
        console.log('ngAfterViewChecked');
    }
}
```

De esta forma veremos y sabremos que hacer con cada uno de nuestros hooks dependiendo de que necesitamos en nuestra aplicación.

PIPES

Los pipes en Angular sirven para ejecutar funcionalidades sobre nuestra vista.

¿Qué quiero decir con esto?

Angular tiene una serie de pipes que ya estan estan definidos por ejemplo :

UPPERCASE, LOWERCASE, CURRENCY, DATE , entre otros...

Con los nombres simplemente te daras cuenta que pueden modificar una variable para que este en mayuscula, minuscula, poner el tipo de moneda, etc ...

Ahora es muy interesante el manejo del mismo... porque nos solo nos hace obviar funciones para manejar los datos, si no que tambien nos deja crear nuestros propios pipes para nuestro proyecto, pero primero vamos a ver como funcionan los base:

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

@Component({
  selector: 'pipe-cmp',
  template: `
    Date : {{ cumpleaños | date }} </br>
    nombre: {{ nombre | uppercase }} </br>
    moneda: {{ diez | currency }}
  `
})
export class OtroCmp{
  cumpleaños = new Date(1989, 1, 13);
  nombre = 'Jorge Cano';
  diez = 10;
}
```



Date : Feb 13, 1989
nombre: JORGE CANO
moneda: USD10.00

La variable cumpleaños realmente se ve asi => Mon Feb 13 1989 00:00:00 GMT-0200 (-02), pero con el manejo de date se pasa directo al location que tiene y pone la fecha tal y como deberia verse!

Así mismo con el nombre, lo pone completo en mayuscula y el numero 10 al agregarle al currency le puso los dolares directo, mas alla de eso podemos indicarle el currency que queramos.

Pero no tiene sentido si fuera esto solo, porque podria hacerlo en cualquier lado... ahora lo importante es que nosotros podemos hacer nuestros propios PIPES.

Vamos a crear nuestro propio pipe!

```
@Component({
  selector: 'my-app',
  template: `
    <h2>Conversor</h2>
    <div>Cantidad: <input [(ngModel)]="cantidad" /></div>
    <div>1 dolar es : <input [(ngModel)]="factor" /></div>
    <p>
      Oh: {{cantidad | conversorPipe: factor}}
    </p>
  `,
})
export class App {

}

@Pipe({name: 'conversorPipe'})
export class ConversorPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return value * (isNaN(exp) ? 1 : exp);
  }
}
```

Nuestro PIPE es un conversor, lo que hace es tomar la cantidad y el factor, para poder hacer la multiplicación.

Entonces, recibe cantidad como value y como el valor del pipe lo que esta despues con los dos puntos, o sea el factor, claro que hacer una simple multiplicación capaz que no tenga mucho sentido.. pero si tenemos calculos mas complejos o funcionalidad mas generica que tenemos que repetir varias veces, por lo tanto dependiendo de lo que hacemos nos va a servir para que los pipes hagan funcionalidades.

HTTP

Uno de los modulos mas importantes de Angular tal vez sea el HTTP, como Angular esta dividido en modulos y para no hacer pesado el core, dividiendo todos por modulos de funcionalidad y este es el HTTP.

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

Como ya sabemos este modulo se ingresa en el NgModule y una vez que lo tenemos vamos a poder trabajar con peticiones a servidores.

Tenemos varias formas de consumir data para nuestra app, la primera seria que tengamos en algun lado un json para poder consumirlo:

Esta constante tendra nuestros datos '**FAKES**'

```
import { Heroe } from '../heroe';
export const HEROES: Heroe[] = [
  {id: 11, nombre: 'Mr. Nice'},
  {id: 12, nombre: 'Narco'},
  {id: 13, nombre: 'Bombasto'},
  {id: 14, nombre: 'Celeritas'},
  {id: 15, nombre: 'Magneta'},
  {id: 16, nombre: 'RubberMan'},
  {id: 17, nombre: 'Dynamia'},
  {id: 18, nombre: 'Dr IQ'},
  {id: 19, nombre: 'Magma'},
  {id: 20, nombre: 'Tornado'}
];
```

Ahora desde el servicio vamos a consumir estos datos:

```
import { Injectable } from '@angular/core';
import { Heroe } from './heroe';
import { HEROES } from './mockups/mock-heroes';

@Injectable()
export class HeroeService {

  constructor() { }

  // retorno comun
  getHeroes(): Heroe[] {
    return HEROES;
  }
}
```

De esta forma cada vez que llegamos a consumir el `getHeroes`, lo que va a pasar es que directamente se va a pasar esta Constante... o sea este array de datos, se asignará directo al quien lo llame.

La segunda forma de consumir los datos es por medio del `HttpModule`.

Para ello más allá de importarlo vamos a tener que hacer algunas cosas más.

Ahora vamos a ver un servicio, que ya tiene todo preparado para hacer todas las llamadas necesarias.

Lo primero que voy a mencionar, es que tenemos dos formas de llamar a nuestra API, la primera es a través de **PROMISE** (si no sabes lo que es promise, te recomiendo que leas un poco sobre promise) y la otra forma es por medio de los **Observables**.

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/toPromise';

import { TICKETS } from './mocks/tickets.mock';

@Injectable()
export class TicketService {

  urlBackend = "http://localhost:3000/";

  constructor(private http: Http){}

  getTicketsMongo (): Promise<any[]> {
    return this.http.get(this.urlBackend + 'tickets')
  }
}
```



```

        .toPromise()
        .then(this.extractData)
        .catch(this.handleError);
    }

    getTicketMongo(id:number):Promise<any[]>{
        return this.http.post(this.urlBackEnd+'ticket', {'id' : id})
            .toPromise()
            .then(this.extractData)
            .catch(this.handleError);
    }

    private extractData(res: Response) {
        let body = res.json();
        console.log("body", body);
        if (body.status == 200){
            return body.result;
        }
        else{
            return { };
        }
    }

    private handleError(error: Response | any){
        let errMsg:string;
        if(error instanceof Response){
            const body = error.json() || '';
            const err = body.error || JSON.stringify(body);
            errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
        }
        else{
            errMsg = error.message ? error.message : error.toString();
        }
        console.error(errMsg);
        return Observable.throw(errMsg);
    }

    getTicketObserver(id){
        return Observable.create(observer=>{
            setTimeout(()=>{
                observer.next(
                    TICKETS.find(
                        (ticket)=>ticket.id == id
                    )
                )
            },3000);
        });
    }
}

```

Bien vamos a dividir el ejemplo por partes:

Imports:

Injectable => este import lo usamos para poder hacer este servicio un singleton y que sobreviva alocado en la memoria mientras nuestra webapp este abierta.

Http, Response => estos imports los necesitamos para hacer las llamadas al mundo exterior, **http** nos sirve para hacer el llamado y **response** es el formato con el cual vamos a manejar el retorno de los datos.

Observable => es parte de RXJS y lo vamos a usar para las llamadas sobre los componentes reactivos, osea los observers, lo mismo asi con todos los imports de RXJS.

Constructor:

En el constructor lo que hacemos es crear la variable HTTP, para poder utilizarla a lo largo de nuestra clase, y poder ir llamando al backend en el momento que queramos.

Funciones:

Vamos a ver las funciones, para ver lo que hacen y como lo hacen:

```
getTicketsMongo (): Promise<any[]> {  
    return this.http.get(this.urlBackEnd + 'tickets')  
        .toPromise()  
        .then(this.extractData)  
        .catch(this.handleError);  
}
```

En esta función podemos ver que hacemos un get por medio de la variable http que creamos y que la devolución de todo esto va a ser un Promise.

Claro http, tiene => get, post, put, delete y maneja todo, le pasamos la url y tenemos la devolución.

```
private extractData(res: Response) {  
    let body = res.json();  
    console.log("body", body);  
    if (body.status == 200){  
        return body.result;  
    }  
    else{  
        return { };  
    }  
}
```

El contenido de la respuesta no es solo el contenido del JSON si no que tambien el status y todo lo que contiene la devolución de una petición HTTP.

Entonces lo que hacemos con extractData es obtener la respuesta, chequear que este todo bien (el status 200) y si es asi retornar el resultado, si no retornar un json vacio.. claro que esto depende de lo que queramos hacer por las especificaciones de nuestra app.

```
private handleError(error: Response | any){  
    let errMsg:string;  
    if(error instanceof Response){  
        const body = error.json() || '';  
        const err = body.error || JSON.stringify(body);  
        errMsg = `${error.status} - ${error.statusText || ''} ${err}`;  
    }  
    else{  
        errMsg = error.message ? error.message : error.toString();  
    }  
    console.error(errMsg);  
    return Observable.throw(errMsg);  
}
```

Ahora en el caso de que salga mal la llamada (no que devuelva un error 500 por ejemplo) si no que falle algo, lo que podemos hacer es manejar el error, en esta funcion lo que hacemos es justamente esto... si el error viene como el tipo **Response** lo que vamos a hacer es directamente parsearlo para traer el error y si viene solo un texto de error , lo vamos a mostrar directamente.

```
getTicketMongo(id:number):Promise<any[]>{  
  return this.http.post(this.urlBackend+'ticket', {'id' : id})  
    .toPromise()  
    .then(this.extractData)  
    .catch(this.handleError);  
}
```

Por otro lado tenemos el post... para mostrar un ejemplo de pasaje de variables, en esta funcion lo que hacemos es un post y le enviamos los datos que necesitamos.

```
getTicketObserver(id){  
  return Observable.create(observer=>{  
    setTimeout(()=>{  
      observer.next(  
        TICKETS.find(  
          (ticket)=>ticket.id == id  
        )  
      )  
    }, 3000);  
  });  
}
```

Por ultimo tenemos la función que se maneja por medio de los observables, pero la gran diferencia es que esta directamente crea un Observable y busca lo que necesita sobre nuestro fake... mas adelante con firebase veremos como funciona directamente... pero ahora es interesante saber que lo podemos manejar tranquilamente...

Por si no lo notaron todas las llamadas HTTP tiene un ".toPromise()" esto es porque en realidad el modulo HTTP de Angular se basa en observables, y si nosotros necesitamos transformarlo a promise tenemos que usar esa función.

Angular Material

<https://material.angular.io/>

Por lo tanto seria como un **AngularMaterialSeed**

ng new probando-material

```
MacBook-Pro-de-Jorge:Documents jorgeucano$ npm install --save @angular/material
```

```
npm install --save @angular/material
```

```
MacBook-Pro-de-Jorge:probando-material jorgeucano$ npm install --save hammerjs
```

```
npm install --save hammerjs
```

```
MacBook-Pro-de-Jorge:probando-material jorgeucano$ npm install --save-dev @types/hammerjs
```

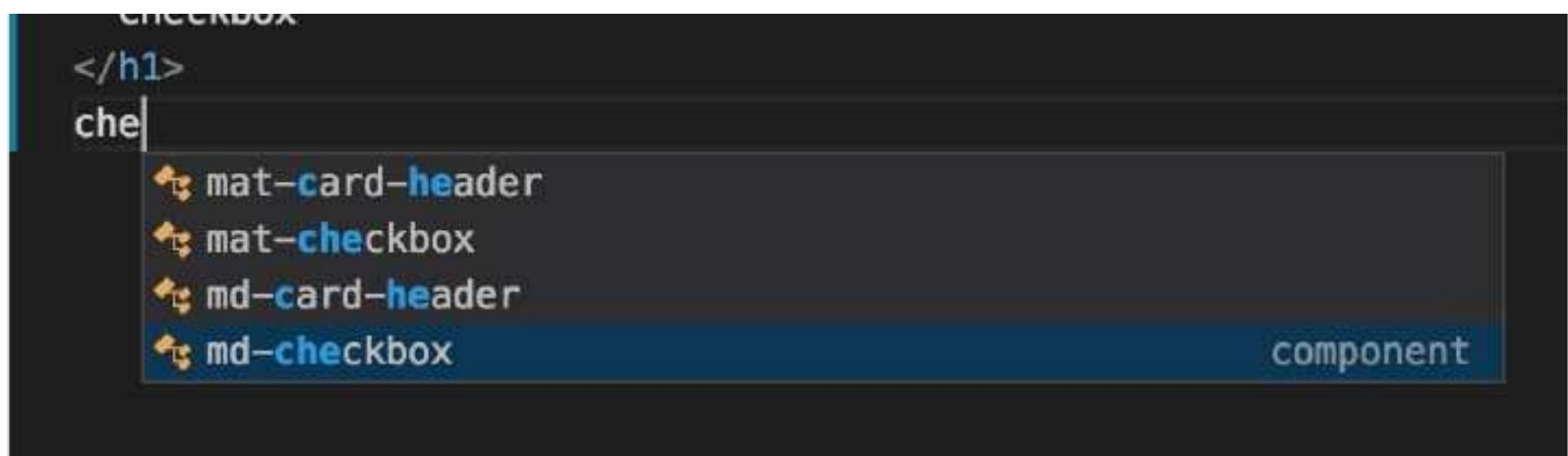
```
npm install --save-dev @types/hammerjs
```

Ahora si, tenemos las dependencias que necesitamos por lo tanto vamos a empezar a probar todo en el componente principal ... Si, claro, vamos a probar lo que nos trae material y ponerlo todo en un solo componente, para que puedan ver que trae cada cosa y como utilizarla :D !

Arrancamos a codear

Lo primero que vamos a hacer, ahora que ya tenemos la dependencia, vamos a agregar el modulo

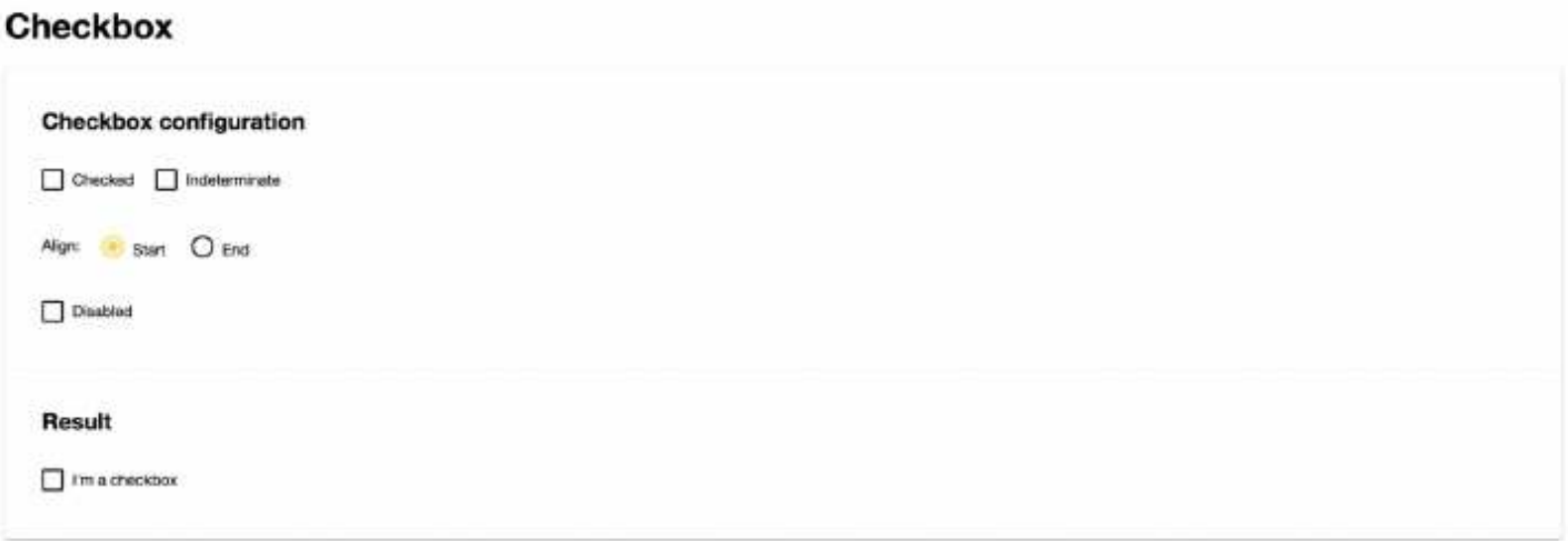
Ahora si, una vez importado el modulo y con la referencia de hammer y icon de material, el beneficio que tenemos con el autocomplete es que ya nos van a aparecer los componentes de material.



Uno de los grandes principios de material design, son las “cards”, estos componentes lo que tienen es la idea de centralizar todo el contenido que queremos mostrar sobre eso... por ejemplo, mostrar un titulo con contenido... pensemos que es un div con formato mas fijo...



CheckBox



Como todo lo parece es bastante fácil de aprender... la verdad que seria todo mas que nada un copy and paste... pero vamos a la realidad...

Que es lo que necesitamos entender para poder trabajar mejor con AngularMaterial =>

Cada uno de los nuevos tags que ponemos es un componente... por lo tanto seguimos con la vida de componentes creada por angular... Algunos de estos componentes tienen inputs y outputs y los vamos a tener que utilizar... tal y como esta en el ejemplo de arriba... md-checkbox tiene de entrada checked align disable... y esto los podemos manejar obviamente desde nuestro componente...

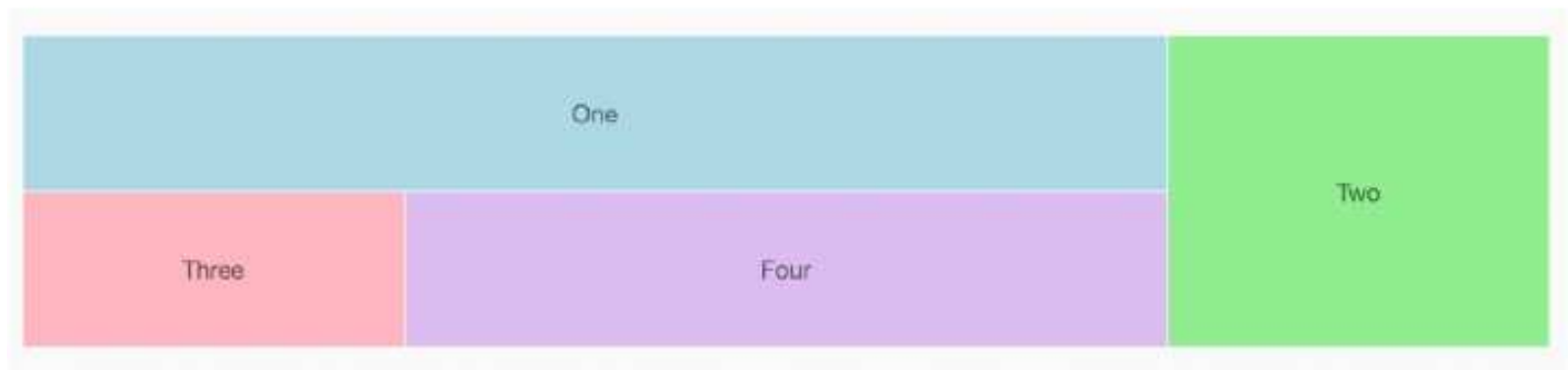
Sigamos un poco con el tema de los componentes...

En el seed van a ver que hay algunos importados... pero creo que una parte muy importante es entender la grilla que se creo para esto... ya que es un componente del tipo

Grid List

Como funciona

Este grid en este caso es dinámico ... y cada uno de los inputs se referencia al titles que esta creado... como queda =>

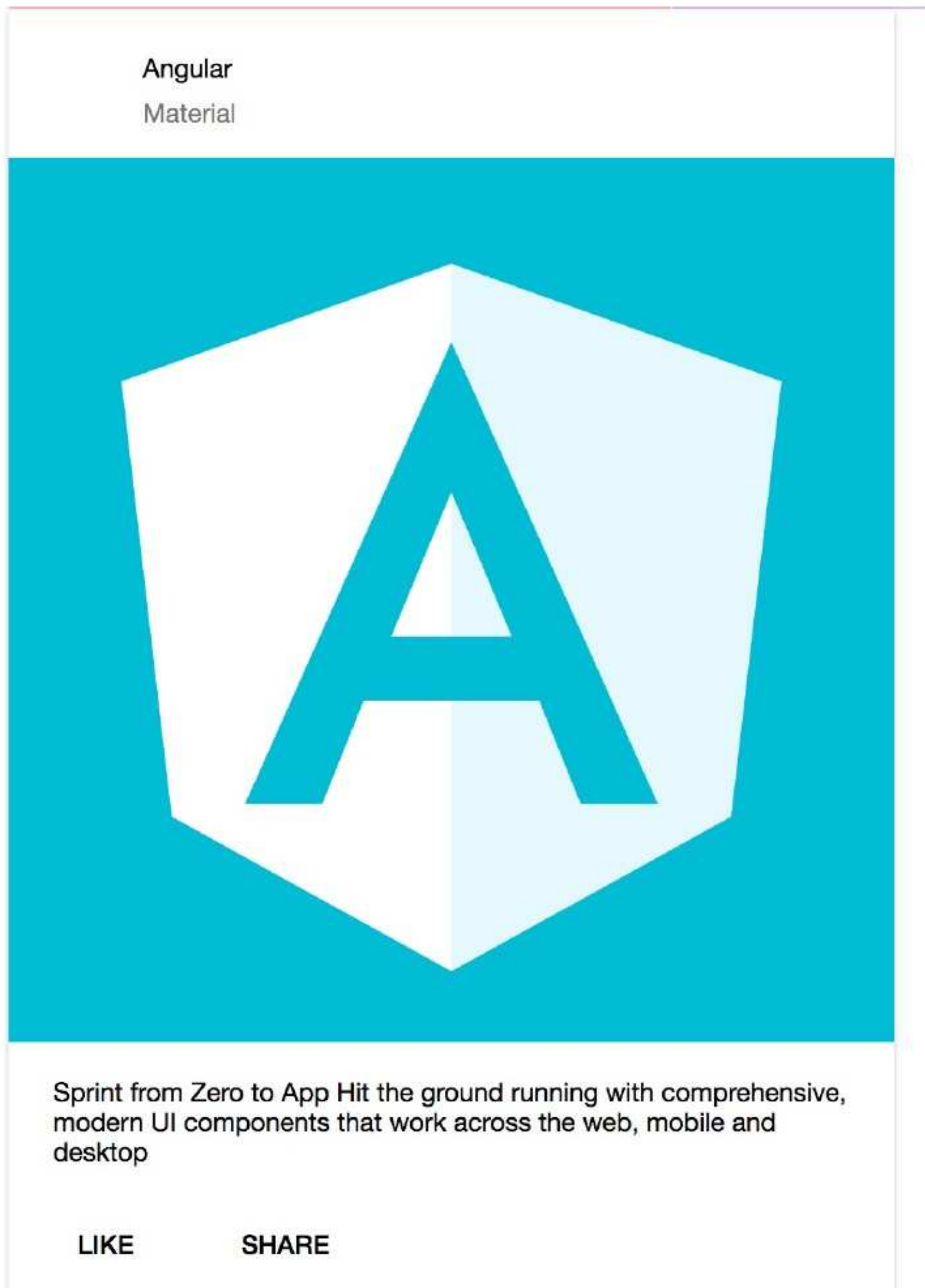


Para entender un poco mejor ... pongamos el código como si no iterara..

en **md-grid-list** vamos a indicar la cantidad de columnas y el height de cada row ...

en **md-grid-tile** , nuestro iterado, vamos a poner la cantidad de columnas y filas que va a ocupar el nuestro “div” y luego simplemente le damos un color y ponemos texto adentro... mas allá del texto podríamos poner cualquier cosa ahí adentro.. como una card por ejemplo...

Y por ultimo vamos a ver como armar una card simple... para ver el funcionamiento de las mismas...



Exactamente tenemos una serie de componentes que generan la card... pero es bastante simple como pueden ver... un contenedor, un header , una imagen, el contenido y por ultimo los botones tal y como los queremos... y gracias al theme que pusimos queda todo

de un color adecuado..

Por lo tanto utilizar AngularMaterial es super facil de usar... simplemente hay que sentarse a mirar como funciona los componentes o copiar y pegar para arrancar a utilizarlos y cambiar el contenido.. al estar dentro de angular como componentes podemos utilizar los inputs para poder darle mas funcionalidad ... y esto nos genera tener algo completo...

Para los developers (como yo) que no son muy buenos con el tema del diseño o la combinación de colores, con esto vamos a olvidarnos... y tampoco nos tenemos que poner a agregar cosas de terceros que podrían hacer perder tiempo... simplemente importamos y empezamos a utilizar...

El ejemplo de todo esto esta en : <https://github.com/jorgeucano/probando-angular-material>

vamos a bajar primeNG

```
MacBook-Pro-de-Jorge:proyect-primeng jorgeucano$ npm install primeng --save
```

npm install primeng—save

Vamos a agregar los css de primeng al CLI

Agregamos

```
“../node_modules/primeng/resources/themes/omega/theme.css”,  
“../node_modules/font-awesome/css/font-awesome.min.css”,  
“../node_modules/primeng/resources/primeng.min.css”
```

Ahora si podemos empezar a programar

Lo primero que vamos a hacer es entrar a la web de primeng para ver que vamos a utilizar

PrimeNG

[_Edit description_www.primefaces.org](#)

Supongamos que queremos agregar un boton (esto pasa con casi todos los componentes de primeng por lo tanto es importante)

Para utilizarlo no basta con importarlo y usar los css.. si no que vamos a tener que importar los modulos que queramos utilizar...

Si señor ... app.module.ts :D

y luego lo podemos utilizar directamente en nuestro html

app works!



app.component.html

Y ya lo tenemos funcionando... pero si vemos un poco el html que ingresamos el texto de adentro del boton esta en el atributo label...

Exacto lo que pensabas... si leiste el post o miraste algo de angularmaterial, en realidad lo que estamos usando son componentes custom de primeng.. por eso tuvimos que importar el modulo y luego poder utilizarlo...

Algo sumamente importante que a veces no lo vemos en la documentación es el tema de los import de los modules... todos los módulos de primeng es el nombre+module ... y se deben importar para poder utilizar... capaz que es algo que le falta a la documentación y a veces uno no entiende porque no le funciona nada...

Juguemos un poco mas ..

Tenemos unos cuantas series de componentes para poder utilizar y no volvernos tan locos con algunas cosas....

En nuestro caso vamos a utilizar algunos para ver su buen funcionamiento... mas que nada vamos a ver:

El sistema grilla (cosa muy importante)

Los inputs para los formularios

Una “tabla” para poder mostrar datos filtrar paginas y demas..

Grillas

app works!



Las grillas son realmente algo fácil de utilizar... Solo tenemos que indicar que vamos a tener un contenedor de grillas y después tenemos el modelo de 12 columnas

Como pueden ver en el código... el ui-g indica lo que seria el “iniciador” de la grilla... y luego los ui-{TAMAÑO}-{NUMERO} indica lo que van a ocupar en cada tamaño de la pantalla... el resto del css es para el ejemplo.

Inputs

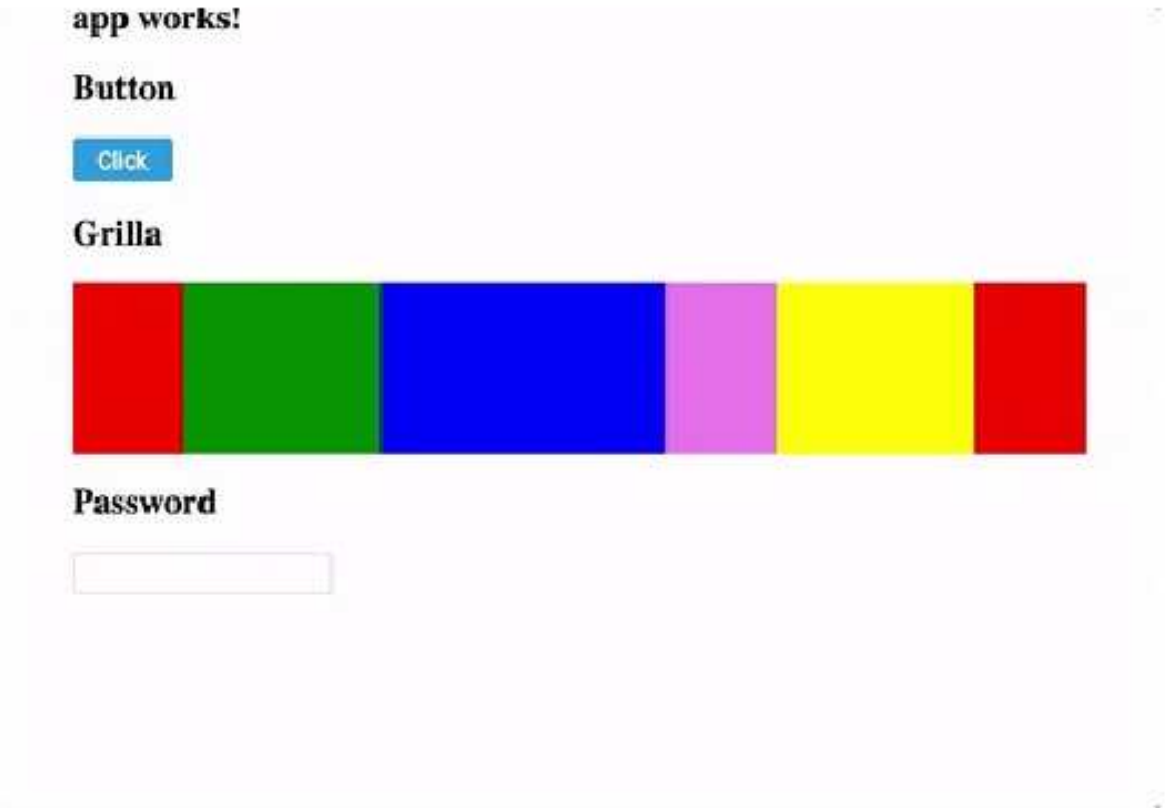
- AutoComplete
- Calendar
- Checkbox
- Chips
- Dropdown
- Editor
- InputSwitch
- InputText
- InputTextArea
- Listbox
- Mask
- MultiSelect
- Password
- RadioButton
- Rating
- Slider
- Spinner
- SelectButton
- ToggleButton
- TriStateCheckbox

Tenemos una gran cantidad de componentes para inputs... los cuales están muy buenos para utilizar... por una cuestión de muestra ... vamos a utilizar el password y el calendar...

Password

Como bien les comente al principio del post, vamos a tener que importar los modulos que vayamos a utilizar... por lo tanto

Y luego lo podemos utilizar tranquilamente!



No le puse el type password para que se pueda observar con que voy llenando y como muestra la “fuerza” del password.

Y por ultimo algo que es muy interesante es el manejo de las tablas responsive, con filtros y otras cosas...

Para eso voy a generar una json con datos ..

Y vamos a necesitar importar las tablas en nuestros modulos, tanto en el app.module.ts como en el componente

y ahora si la podemos empezar a utilizar

y esta parte de la tabla queda así

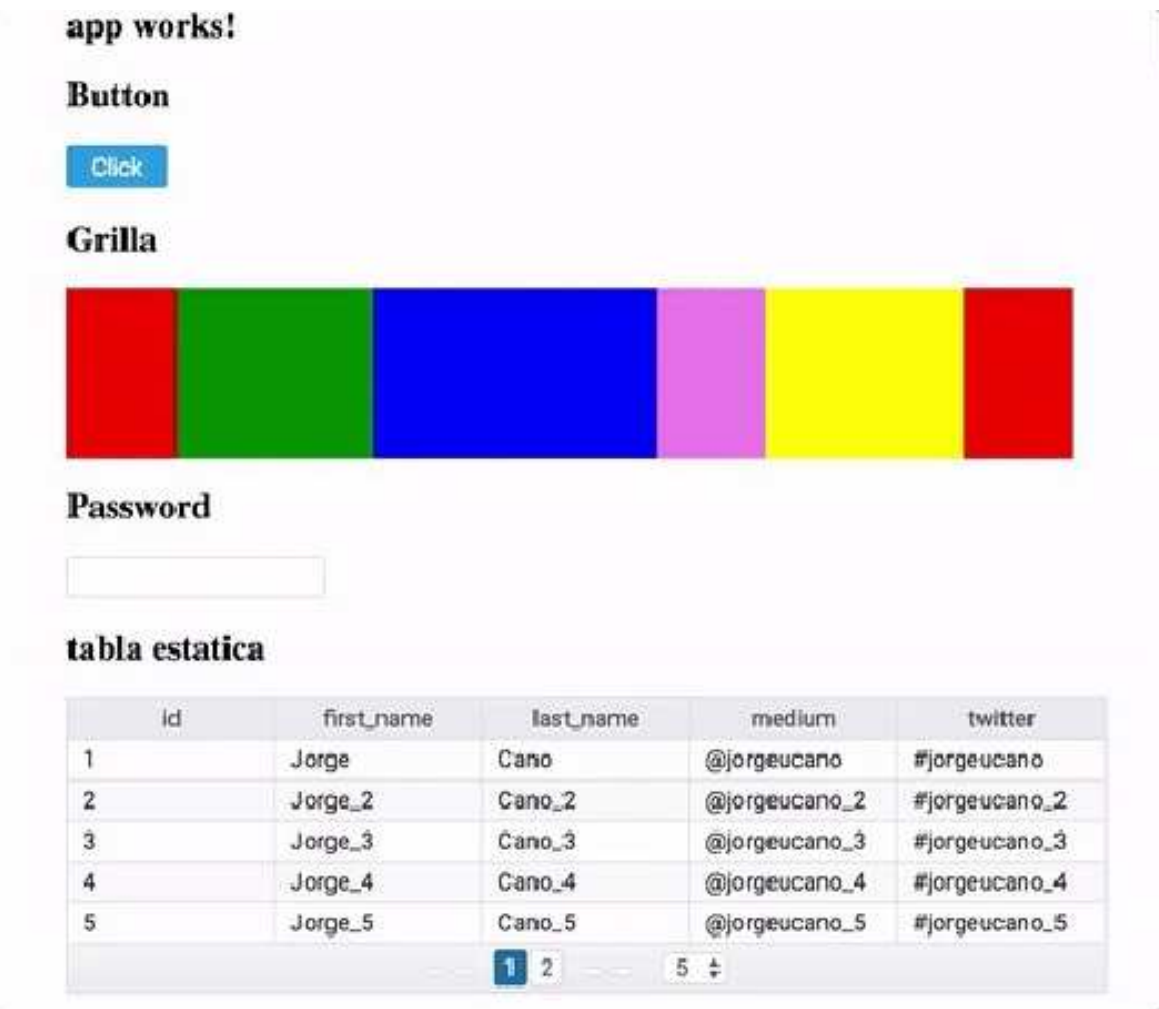
id	first_name	last_name	medium	twitter
1	Jorge	Cano	@jorgeucano	#jorgeucano
2	Jorge_2	Cano_2	@jorgeucano_2	#jorgeucano_2
3	Jorge_3	Cano_3	@jorgeucano_3	#jorgeucano_3
4	Jorge_4	Cano_4	@jorgeucano_4	#jorgeucano_4
5	Jorge_5	Cano_5	@jorgeucano_5	#jorgeucano_5
6	Jorge_6	Cano_6	@jorgeucano_6	#jorgeucano_6
7	Jorge_7	Cano_7	@jorgeucano_7	#jorgeucano_7
8	Jorge_8	Cano_8	@jorgeucano_8	#jorgeucano_8
9	Jorge_9	Cano_9	@jorgeucano_9	#jorgeucano_9
10	Jorge_10	Cano_10	@jorgeucano_10	#jorgeucano_10

Ahora vamos a ponerle complejidad... por ahora lo único que hace es iterar el objeto y ponerlo ...

Agregando una serie de inputs a nuestra tabla vamos a tener muchos beneficios...

```
[rows]="10" [paginator]="true" [pageLinks]="3" [rowsPerPageOptions]="[5,10,20]"
```

Con este código simplemente en nuestro html vamos a lograr esto...



le indicamos la cantidad de rows inicial, que vamos a paginar, cuantos “paginas” queremos ver y luego cuantas opciones de rows por pagina queremos y ya funciona.

Otra cosa muy importante en esta iteración de datos y tablas, es que sea responsive... y tenemos la solución para esto...

```
[responsive]="true" [stacked]="stacked"
```

