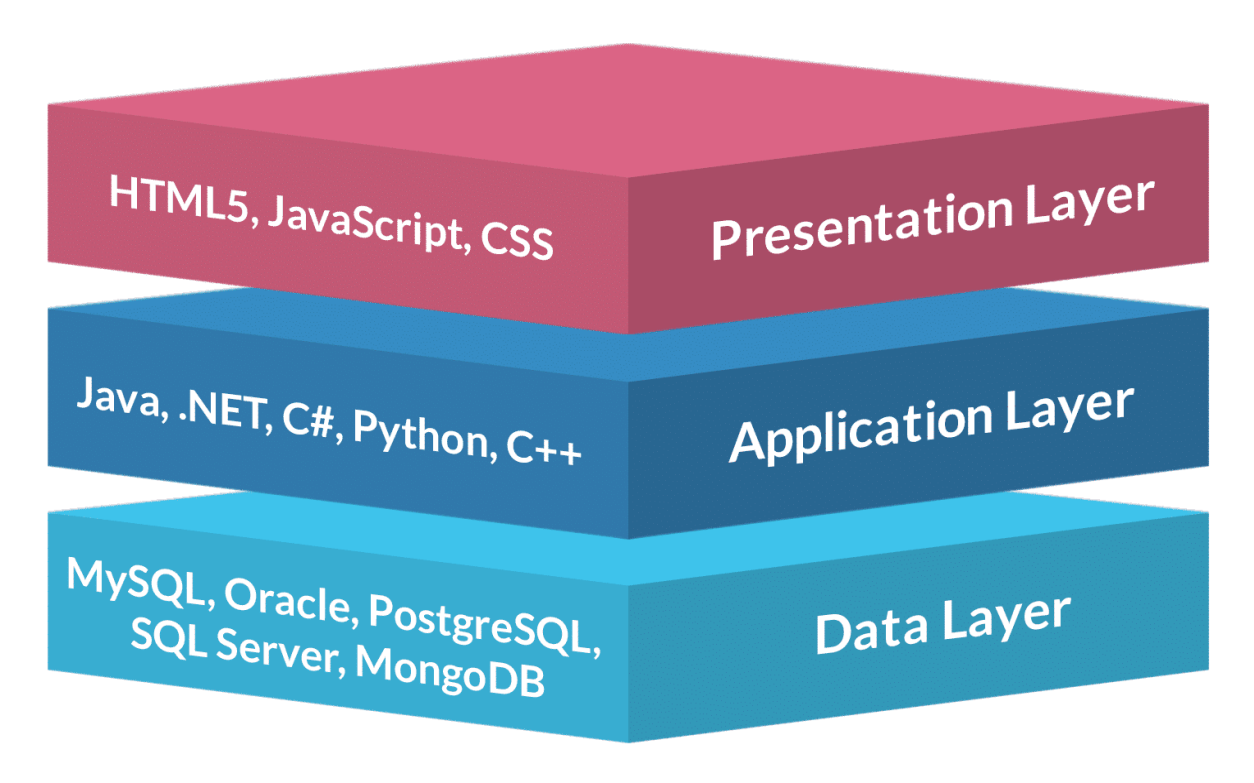
Desarrollo WEB con

ANGULAR

- Angular es un *framework* para la construcción de la interfaz de usuario, capa de presentación o vista en una arquitectura distribuida.
- Su estudio es interesante pues por las bondades que *presenta* se está convirtiendo en una tecnología estándar

Arquitectura 3 Capas



FrameWork

- Recuerda que un framework es un programa “hecho a medias”, al cual yo le puedo añadir mis piezas, adecuándome al estilo que el programa base espera y construir así mi aplicación, ajustada a mis necesidades

FrameWork

- Un Framework “gusta” porque aporta:

  Soluciones a problemas comunes

  Rapidez en el desarrollo

  Tecnología testada, robustez

  Y en general PRODUCTIVIDAD

ANGULAR

- Angular emplea internamente, el conjunto de tecnologías estándar empleadas en el mundo de los navegadores web:

  HTML

  CSS

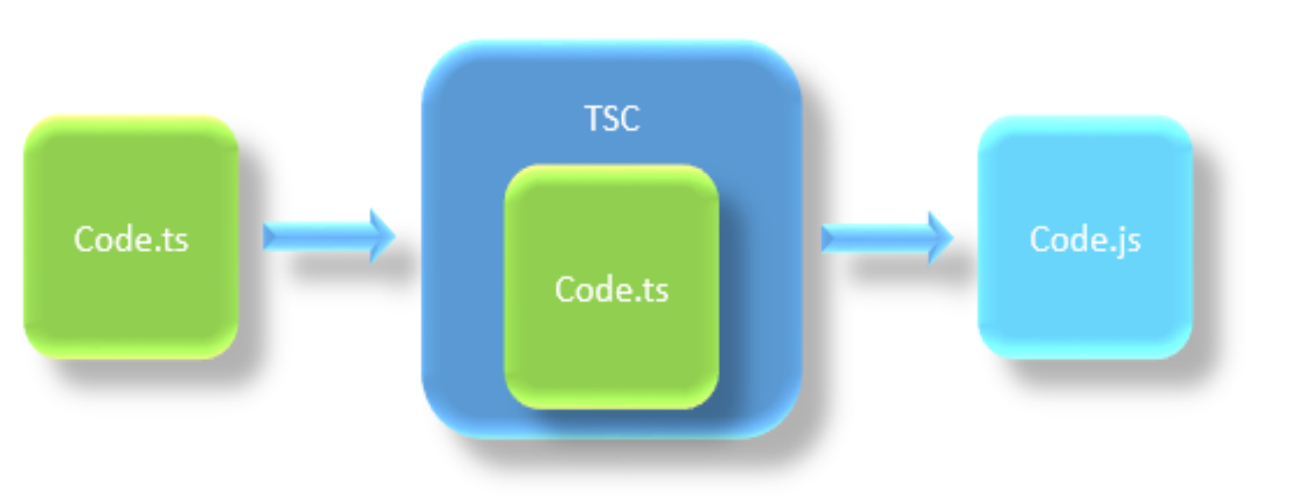
  ECMA-Script (JavaScript)

ANGULAR

- Pero para su uso, añade una capa nueva que consiste en un nuevo lenguaje denominado “TypeScritp” creado por Microsoft

- Todo mi código fuente será con extensión .ts y acabará traduciéndose en HTML, CSS y JavaScript

TRANSPILAR



La transpilación es un caso especial de compilación donde el lenguaje traducido y el resultante son ambos de alto nivel

Por qué ANGULAR

- Llegados a este punto, uno podría o más bien debería preguntarse:

  Si ya sé JavaScript, Html y CSS, que además son estándar, ¿para qué usar ahora TypeScript con Angular?

Por qué ANGULAR

- TypeScript, al contrario que JS, es un lenguaje que pertenece al paradigma de la programación orientación a objetos.

Todo (variables, conceptos, relaciones funcionalidad) tiene su tipo y una estructura bien definida de antemano.

Por qué ANGULAR

- La solución que representa una aplicación, queda expresada sin ambigüedades, lo que promueve la legibilidad, la mantenibilidad y la reutilización del código (características últimas deseables en todo software)

Por qué ANGULAR

- Características más mundanas, pero también destacables, del uso de Angular son:

  Definición y ejecución de Tests

  Gestión de dependencias

  Disminución de errores de tipo

Por qué ANGULAR

- En resumen: la amalgama que puede constituir la mezcla de código HTML, CSS y JavaScript en una misma app, queda “normalizada” al ser incluida en contexto común y representada en clases, bajo una estructura estándar (un proyecto Angular)

ENTORNO

Node.js Intérprete JS fuera del navegador

NPM Node Package Manager. Repositorio de librerías/módulos JS (node C npm)

Angular Cli Paquete básico que construye la estructura y dependencias iniciales de todo proyecto Angular)

npm install -g @angular/cli

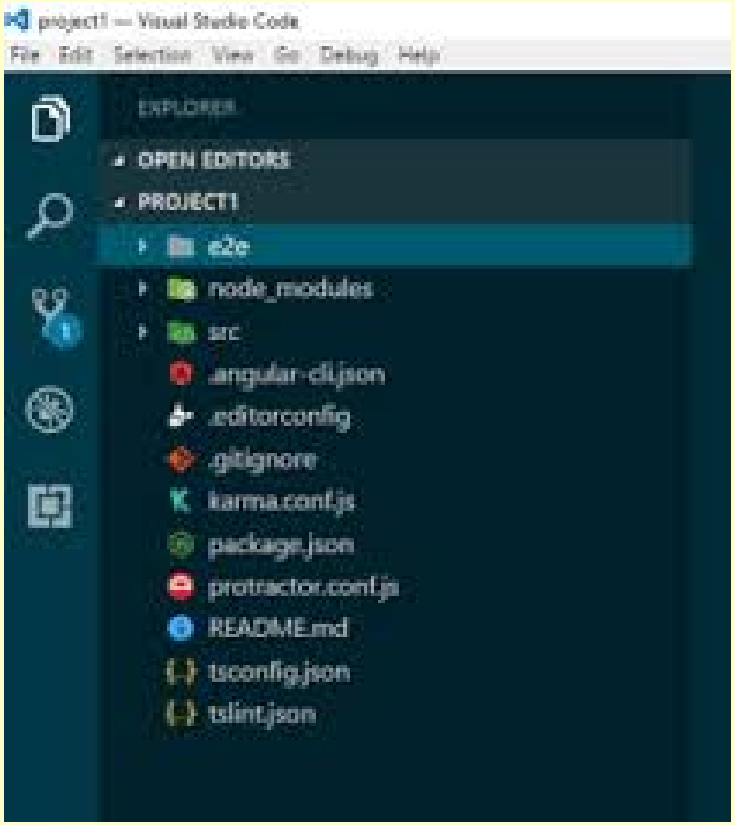
APLICACIÓN ANGULAR

ng new my-app //creo una nueva app

cd my-app //me introduzco en el directorio

ng serve --open // lanzo mi app en el navegador

ESTRUCTURA



Cualquier aplicación Angular tiene una estructura común

ESTRUCTURA

my-app .- carpeta raíz: engloba el proyecto

my-app/e2e .- carpeta que contiene los ficheros de prueba integrales o test *end to end* (Protactor)

my-app/package.json .- fichero con las dependencias de mi proyecto (npm)

my-app/nodemodules .- donde se descargan las dependencias que mi proyecto referencia

ESTRUCTURA

src .- carpeta con el código fuente

src/index.html .- página de inicio de mi aplicación

src/app/*.spec.ts .- ficheros empleados en los test unitarios (Karma Jasmine)

src/app/module.ts - El módulo raíz. “mi main”

src/assets .- Para los archivos de imagen, y otros de importancia y uso en mi app

ESTRUCTURA

src/main.ts .- El punto de entrada en la aplicación. Empezará ejecutando el módulo raíz

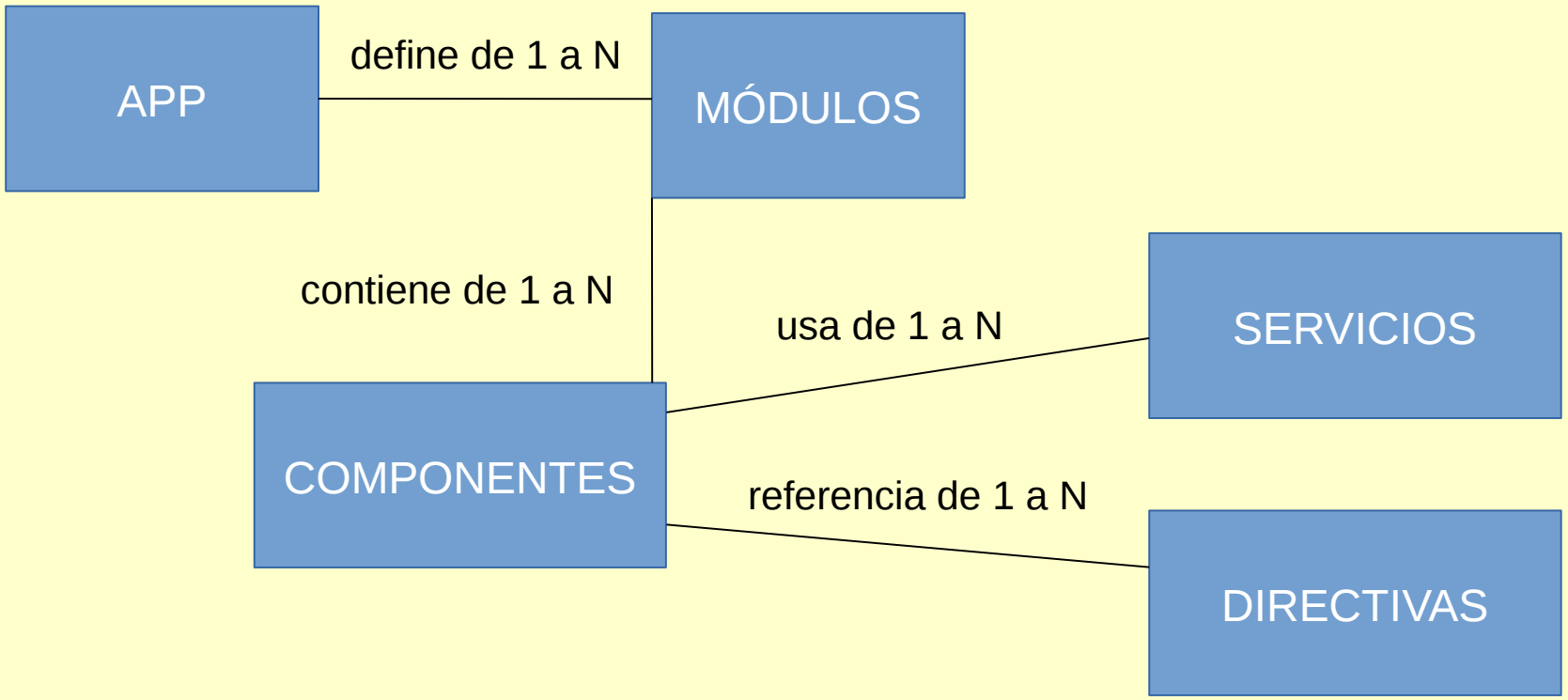
src/polyfills.ts .- normaliza el comportamiento de diferentes navegadores (uso avanzado)

ELEMENTOS

Tipos de clases TypeScript en Angular:

- Módulo
- Componente
- Servicio
- Directiva

ELEMENTOS


APP
define de 1 a N
MÓDULOS
contiene de 1 a N
usa de 1 a N
SERVICIOS
COMPONENTES
referencia de 1 a N
DIRECTIVAS

ELEMENTOS

Otros tipos de menor entidad:

- Pipes
- Clases e interfaces (propias)

Nomenclatura

La mayoría de los archivos TypeScritp acaban en extension .ts. Pero su nombre, debe indicarnos qué tipo de elemento es y hay un estándar que debemos seguir

Nomenclatura

module.ts → módulos

component.ts → componentes

component.html → plantilla

component.css → estilo

model.ts / interface.ts → interfaces

service.ts → servicios

Decoradores

Todos los elementos son clases TypeScritp. Para distinguir entre las distintos tipos de clases se emplean los llamados decoradores.

No es más que una información que acompaña a la definición de la clase para ayudar a Angular a identificarlo y configurarlo como tal.

Decoradores

En otras tecnologías, un decorador vendría a ser una directiva

@NgModule para Módulos

@Component para Componentes

@Directive para Directivas

@Injectable para Servicios

@Pipe para Tuberías/formateadores

Componentes

```
1  import { Component } from '@angular/core';
2  import { JsonPipe } from '@angular/common';
3  import { Login } from './app.login.model'
4  import { LoginService } from './login.service';
5  import { HttpResponse } from '@angular/common/http';
6
7  @Component({
8    selector: 'app-root',
9    templateUrl: './app.component.html',
10   styleUrls: ['./app.component.css'],
11   providers: [LoginService]
12 })
13 export class AppComponent {
14   login: Login;
15
16   constructor(private logservice: LoginService) {
17     this.login = new Login();
18   }
19
20   static informa (datos)
```

Componentes

Un componente es una vista, o un trozo de una Página. Está asociado a ella por una etiqueta.

Respalda los datos que pueden enviarse y recibirse en esa sección web y define los métodos y operaciones que pueden derivarse de eventos en la plantilla asociada

Componentes atributos

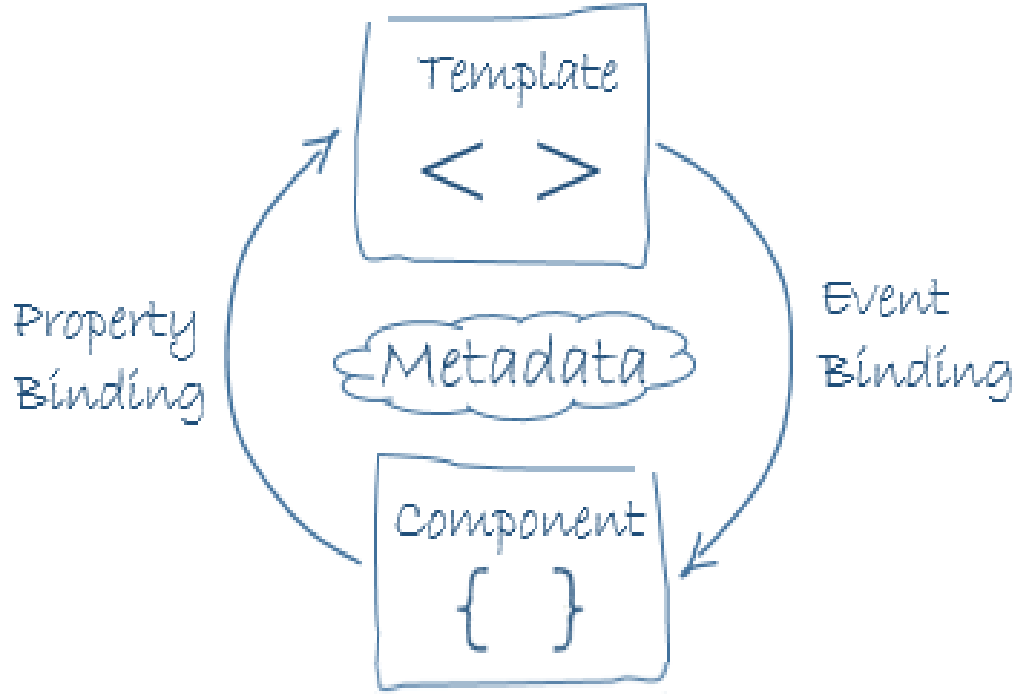
selector: la etiqueta html que enlaza con la plantilla del componente

templateUrl: la plantillla, el html propio del

stylesUrl:el css asociado a la plantilla

providers: los servicios usados por el componente

Com C's → Plantilla



Com C's → Plantilla

El intercambio de información entre la plantilla y su componente recibe el anglicismo de *binding*

Interpolación: (Hacia el DOM)

Property binding: (Desde el DOM)

Event binding: (Desde el DOM)

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

Com C's → Plantilla

Interpolación: {{todo.subject}}

Un atributo todo llamado en el componente tiene una propiedad subject que se representa en la página

Property binding: [todo]="selectedTodo"

Una propiedad todo en el componente, es seteada con el valor del selectedTodo

Com C's → Plantilla

Event binding: (click)="selectTodo(todo)"

El evento click sobre un elemento provocará la llamada del método selectTodo

Two-way binding: [(ngModel)]="todo.subject"

Si el atributo subject cambia, cambiará el componente ngModel. Y viceversa

Com entre C's

Puede ser necesario compartir info entre Componentes Para ello, hay cuatro métodos previstos:

@Output + EventEmitter – Del hijo al padre

@Input – Del padre al hijo

@ViewChild – Del hijo a padre

Via servicio – entre cualquiera -

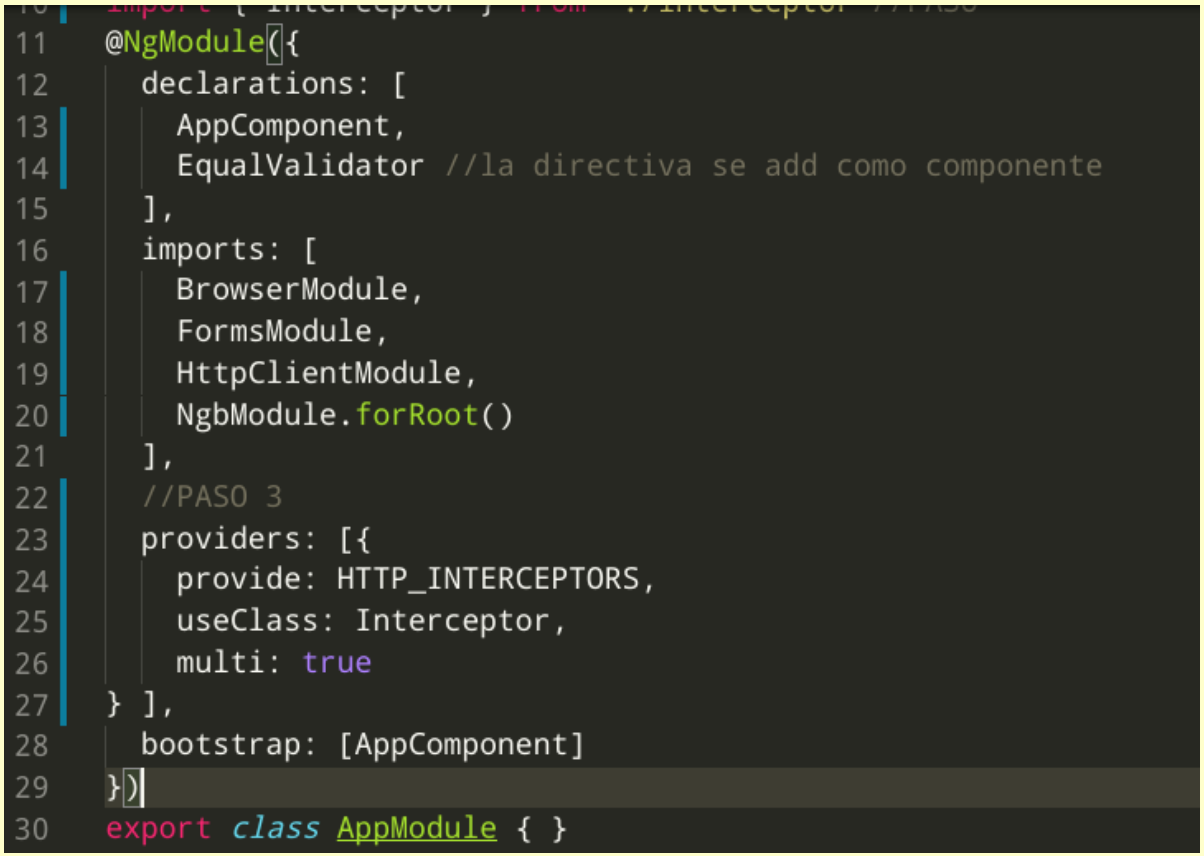
Módulos

Un módulo Angular, es una clase que viene identificada por el decorador @NgModule

Un módulo representa una agrupación Funcional, que referencia a una serie de Componentes y Servicios

Toda app Angular tiene al menos un módulo

Módulos



```
11  @NgModule({
12    declarations: [
13      AppComponent,
14      EqualValidator //la directiva se add como componente
15    ],
16    imports: [
17      BrowserModule,
18      FormsModule,
19      HttpClientModule,
20      NgbModule.forRoot()
21    ],
22    //PASO 3
23    providers: [{
24      provide: HTTP_INTERCEPTORS,
25      useClass: Interceptor,
26      multi: true
27  } ],
28    bootstrap: [AppComponent]
29  })
30  export class AppModule { }
```

Módulos Atributos

declarations: enumera a los componentes y directivas empleados en el módulo

imports: enumera las librerías de js empleadas

providers: los servicios usados en el módulo

bootstrap: el componente que empieza a interpretarse en primer lugar

Módulos Angular vs JS

Un módulo JavaScript es una librería. Un conjunto de tipos, métodos y atributos listos para referenciarse y emplearse en mi proyecto

Un módulo Angular es un componente reutilizable propio de Angular, que agrupa una serie de componentes y servicios. Una entidad lógica y funcional a nivel de Angular

TypeScript Tipos

En TypeScritp, todo debe tiparse. Así que hay que añadir el tipo a :

Variables

Atributos

Parámetros formales

Tipos devueltos por los métodos

TypeScript Tipos

enum: rango valores predefinido

number: cualquier número entero o real

string: cadenas

any: lo que sea (no comprueba tipos)

object: ninguno de los primitivos

void: no devuelve nada

TypeScript Casting

```
var objetoA: TypeA;
var objetoX: any;

objetoA = <TypeA> objetoX;//ambos son
objetoA = objetoX as TypeA;//válidos

let idn = +ids; //Casting de String a número
```

Servicios

El decorador @Injectable acompaña a los Servicios en su definición y ayuda a Angular a identificarlos como tal.

La inyección de dependencias es el mecanismo previsto para cuando un Componente emplea un Servicio.

Servicios

A efectos prácticos, los servicios son usados para enviar y recibir datos del servidor al cliente.

Los componentes muestran los datos, pero estos son ofrecidos por el servicio correspondiente

Servicios

Si un C usa un Servicio, éste puede declararse en el decorador del C o del Módulo que lo engloba. Depende del ámbito

```
export class HeroListComponent {
  heroes: Hero[];

  Constructor( /*private*/ heroService: HeroService) {
    this.heroes = heroService.getHeroes();
  }
```

Servicios DI

La inyección de dependencias es un mecanismo por el cual, en el constructor defino un parámetro que quiero usar. Como por arte de magia, Angular lo instancia y me lo pasa en la llamada

```
constructor( private heroService: HeroService) {
    this.heroes = heroService.getHeroes();
  }
```

Interfaces

La interfaz es un tipo propio de TypeScript, sin equivalente en JavaScritp.

```
export interface SearchItemReducido {

    artistName : string,
    artworkUrl100 : string,
    previewUrl : string

}
```

Interfaces

Con él puedo definir tipos o métodos abstractos

```
export interface BusquedaInterface {

    busca (termino : string):any;

}
```

Interfaz o Clase

Emplearé una interfaz cuando quiera abstraer servicios con diferentes implementaciones o cuando quiera detallar el tipo de datos de intercambio de mi ciente con el servidor.

Si además del mero tipo, veo la necesidad de definir métodos asociados a él, debo modelarlo como una clase

Directivas

Una directiva puede entenderse como un atributo de un elemento HTML.

Las hay predefinidas o también Puedo definir mis propias directivas con el decorador @Directive

Directivas

Las hay de dos tipos:

Estructurales: Para hacer bucles de los elementos o condicionar su aparición

<div ***ngFor**="let todo of todos">
<todo-detail ***ngIf**="selectedTodo"></todo-detail>

Directivas

Atributo: Para validar, asignar el estilo dinámicamente, modificar su apariencia, comunicar componente-plantilla, etc..

<div **[ngClass]**="setTodoClasses()">Este ToDo es importante y está pendiente

Observable

Observable es el sustituto de Promises

A un objeto Observable, puedo asociarle un Observador, que será invocado ante cambios de estado

Es el mecanismo previsto para consumir servicios remotos

Observable

```
this.itunes.busca(term).subscribe //busca devuelve un Observable
    ( ok => this.gestionRespuestaOk (ok),
    error =>this.gestionRespuestaError (error),
completado=>this.gestionCompleta (completado)  );
```

El orden importa, pues programa los callbacks en caso de que 1) haya ido bien, 2 mal o 3 se haya completado

Formularios

El formulario es un elemento habitual en las aplicaciones web. Angular le da soporte en dos versiones:

1 Formularios Template driven

2 Formularios Model driven o Reactivos

Formularios Template

Más sencillos

Validación declarativa (no test)

Dirty/pristine valid touched para atributos

Value valid submitted para formulario

Formularios Reactivos

Más complejos (anidados)

FormGroup, FormControl, FormArray

Validadores son clases (test programático)

Atributos control, valid

Listeners validación y cambio de valores

No tenemos acceso a submitted

Rutas

Pulsar un enlace en un entorno web estándar, provoca en el envío de una petición Get por parte del navegador al servidor.

En una aplicación Angular, en realidad siempre estamos en una misma página, de modo que el artificio de la navegación, se hace mediante el módulo Router

Rutas

1) Defino la sección en la plantilla donde irán los enlaces de navegación

```
<nav>
    <a routerLink="/alumnos">Asistentes</a>
    <a routerLink="/minsait">Minsait</a>
    <a routerLink="/">Inicio</a>
</nav>
<router-outlet></router-outlet>
```

Rutas

2) Defino un módulo donde hago corresponder a cada ruta con su componente destino

```
const routes: Routes = [
{ path: 'alumnos', component: AlumnosComponent },
{ path: 'minsait', component: MinsaitComponent },
{ path: 'alumno/:id', component: AlumnoDetailComponent }
];
```

Rutas

...

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

Aspectos avanzados

- I18n
- Testing unitario Karma + Jasmine
- Ciclo de vida de un componente
- Test de integración E2E con Protactor
- Diferencias entre modo desarrollo producción

ENLACES

Página web oficial de Angular

Página web oficial de Node.js

Referencia oficial TypeScritp

Comunicación entre C's

Definición de directivas propias

Diagrama de Observables