Tema 6. Desarrollo web con Angular

Programación web

Boni García

Curso 2017/2018

Índice

1. Introducción
2. Node.js
3. TypeScript
4. Angular
5. Librerías de componentes

Índice

1. Introducción
   - Motivación de Angular
   - Versiones Angular
   - Principales características de Angular
2. Node.js
3. TypeScript
4. Angular
5. Librerías de componentes

Introducción

Motivación de Angular

- Problema: las aplicaciones web con uso intensivo de tecnologías cliente como JavaScript/jQuery son muy difícil de mantener
- Angular es un framework creado por Google para el desarrollo de aplicaciones web complejas de forma modular
- Angular permite la creación de aplicaciones web **SPA** (*Single Page Application*): Aplicaciones web de página única, que permiten una experiencia muy fluida a los usuarios (de forma similar a una aplicación de escritorio)

Angular 1.x

https://angularjs.org

https://angular.io/

Angular 2+

Introducción

Versiones Angular

- **Angular 1.x**
  - Es lo que se conoce como como Angular.js
  - Basado únicamente en JavaScript
- **Angular 2**
  - Supuso una importantes refactorización con respecto a Angular.js (Angular 1 y Angular 2+ son incompatibles)
  - Se recomienda usar **TypeScript** (también se puede usar JavaScript ES5 o ES6)
- **Angular 4**
  - Los cambios con respecto a Angular 2 son mínimos (mejoras de rendimiento del *core* de Angular, manteniendo la compatibilidad hacia atrás con Angular 2)
- **Angular 5**
  - Incorpora soporte para aplicaciones web progresivas
  - Mantiene compatibilidad hacia atrás con Angular 2

Introducción

Principales características de Angular

- Las aplicaciones se construyen mediante la composición de unidades llamadas **componentes** que tienen una lógica de negocio (Typescript) asociada a una presentación (HTML)
- Angular sigue una variante del **patrón MVC** para una articulación flexible entre presentación, datos, y lógica de negocio
- Usa la de **inyección de dependencias** para fomentar la modularidad
- Promueve el diseño modular basado en **servicios**
- Promueve las pruebas (*testability*), al usar componentes aislados que pueden ser verificados de manera independiente

Introducción

Principales características de Angular

- Está diseñado siguiendo el paradigma de **programación declarativa**, que es considerado un mejor enfoque para la creación de interfaz de usuario que la programación imperativa
  - En la programación imperativa se describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución (por ejemplo: C, C++)
  - En la programación declarativa las sentencias que se utilizan lo que hacen es describir el problema que se quiere solucionar, pero no cómo (por ejemplo: SQL, HTML)

```
List<int> collection = new List<int>
          { 1, 2, 3, 4, 5 };
List<int> results = new List<int>();
foreach(var num in collection) {
    if (num % 2 != 0)
    results.Add(num);
}
```

```
var results = collection.Where(num =>
num % 2 != 0);
```

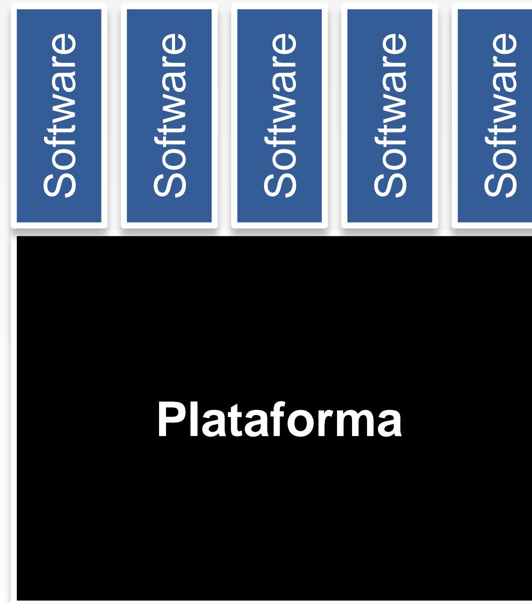
Índice

1. Introducción
2. Node.js
   - ¿Qué es Node.js?
   - Instalación
   - Stack MEAN
   - Ecosistema Node.js
3. TypeScript
4. Angular
5. Librerías de componentes
6. Ionic 2

Node.js

¿Qué es Node.js?

- **Node.js** es un *entorno de ejecución* de aplicaciones **JavaScript** basada en V8 (el motor de JavaScript de Google)

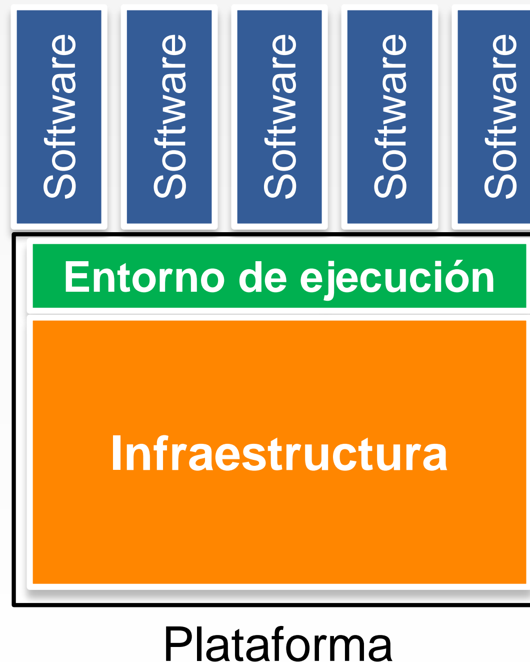


En general, una ***plataforma*** es el nombre que se le suele dar a un sistema que permite ejecutar aplicaciones

Node.js

¿Qué es Node.js?

- **Node.js** es un *entorno de ejecución* de aplicaciones **JavaScript** basada en V8 (el motor de JavaScript de Google)

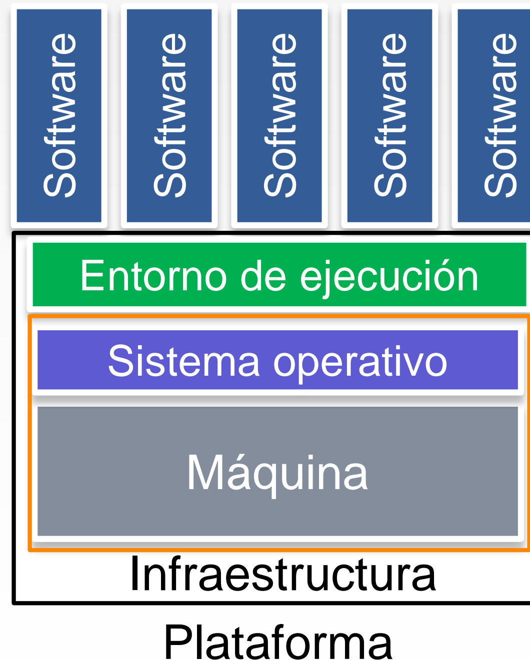


Una plataforma está compuesta por una ***infraestructura*** que puede estar provista de un ***entorno de ejecución***

Node.js

¿Qué es Node.js?

- **Node.js** es un *entorno de ejecución* de aplicaciones **JavaScript** basada en V8 (el motor de JavaScript de Google)



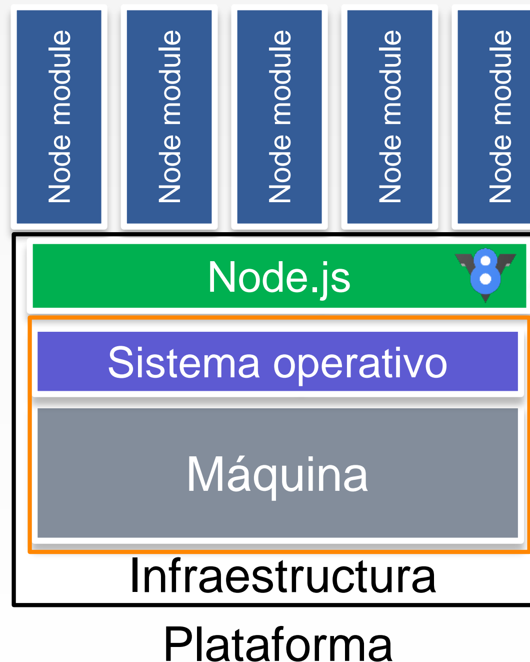
El concepto de *infraestructura* suele estar ligado a una máquina (física o virtual) con un sistema operativo

Node.js

¿Qué es Node.js?

- **Node.js** es un *entorno de ejecución* de aplicaciones **JavaScript** basada en V8 (el motor de JavaScript de Google)

https://nodejs.org/



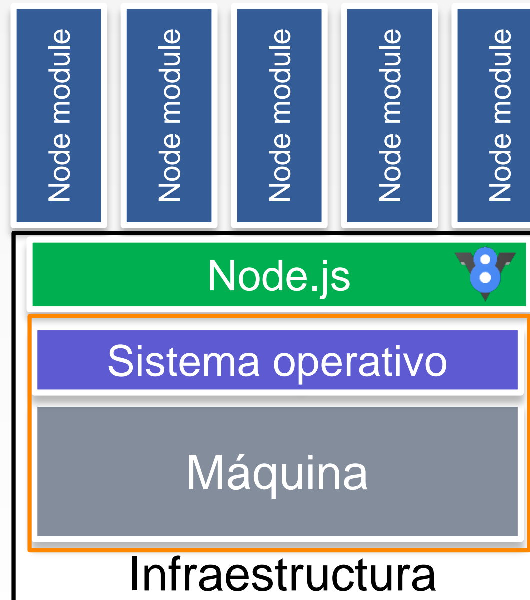
Podemos ver Node.js como una capa de software que permite la ejecución de aplicaciones desarrolladas con JavaScript

Node.js

¿Qué es Node.js?

- **Node.js** es un *entorno de ejecución* de aplicaciones **JavaScript** basada en V8 (el motor de JavaScript de Google)

https://www.npmjs.com/

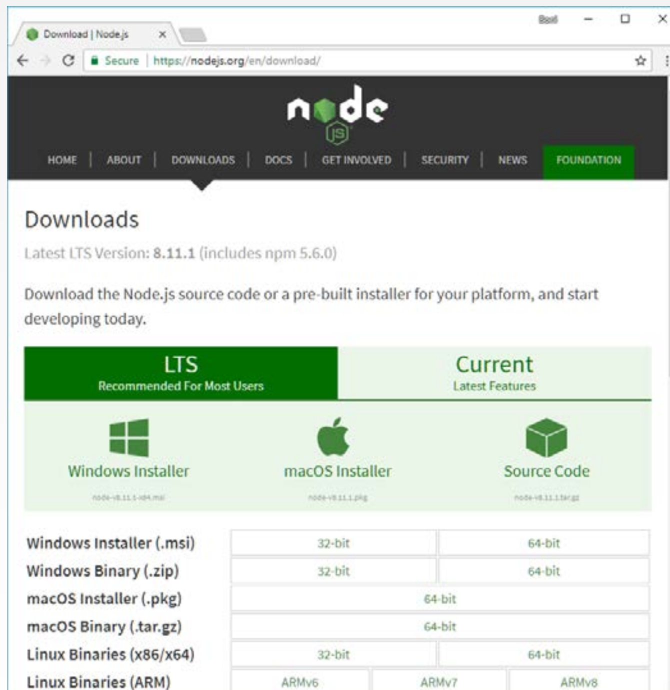


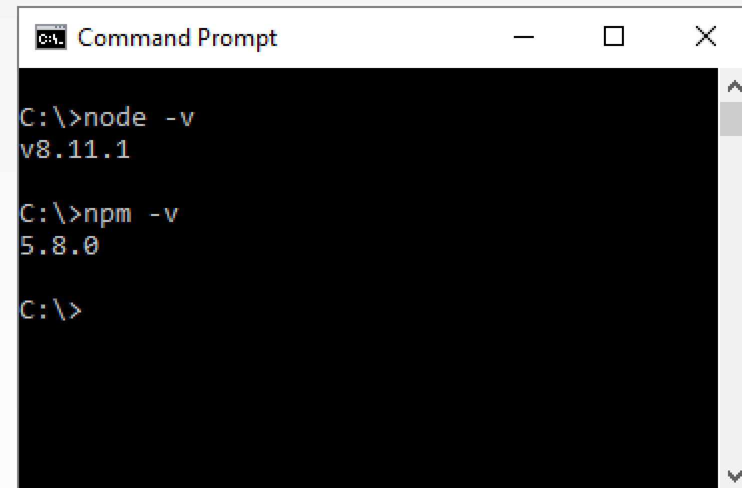
El gestor de paquetes en Node.js se llama **NPM** (*Node.js Package Manager*)

Node.js

Instalación

- La versión LTS (*Long Term Support*) actual de Node.js es la v8





https://nodejs.org/es/download/

Node.js

Stack MEAN

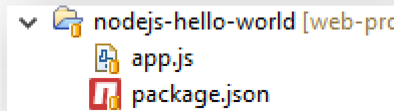
- En el mundo Node.js, se puede usar **MEAN** para desarrollo *full-stack*
  - M=MongoDB (base de datos no-sql)
  - E=Express.js (back-end de facto en el mundo Node.js)
  - A=Angular (framework front-end)
  - N=Node.js (plataforma)

Node.js

Fork me on GitHub

Stack MEAN

- Vamos a ver el ejemplo más sencillo posible (*hello world*) usando Node.js y Express.js:



app.js

```
var express = require('express');
var app = express();

app.get('/', function(req, res) {
    res.send('Hello World!');
});

app.listen(3000, function() {
    console.log('Example app listening on port 3000!');
});
```

Node.js

Fork me on GitHub

Stack MEAN

- Vamos a ver el ejemplo más sencillo posible (*hello world*) usando Node.js y Express.js:

package.json

```
{
   "name": "nodejs-hello-world",
   "version": "1.0.0",

   "description": "Node.js simplest app ever",
   "license": "Apache-2.0",
   "repository": {
      "type": "git",
      "url": "git+https://github.com/bonigarcia/web-programming-examples.git"
   },

   "dependencies": {
      "express": "^4.15.0"
   }
}
```

El fichero `package.json` sirve para especificar las características y dependencias de una aplicación Node.js (*Node module*)

Las versiones en Node.js pueden usar estos símbolos:

- tilde (~) : Asegura la versión *minor*
- caret (^): Asegura la versión *major*

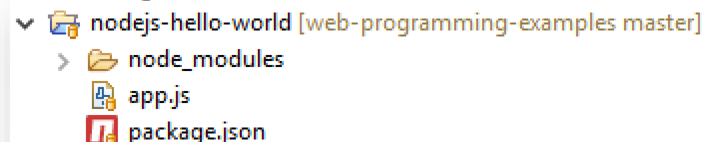
Node.js

Stack MEAN

- Vamos a ver el ejemplo más sencillo posible (*hello world*) usando Node.js y Express.js:

```
> npm install
nodejs-hello-world@1.0.0 D:\projects\web-programming-examples\nodejs-hello-world
npm notice created a lockfile as package-lock.json. You should commit this file.
up to date in 0.383s
```

Después de ejecutar el comando `npm install`, las dependencias se descargan en la carpeta `node_modules`



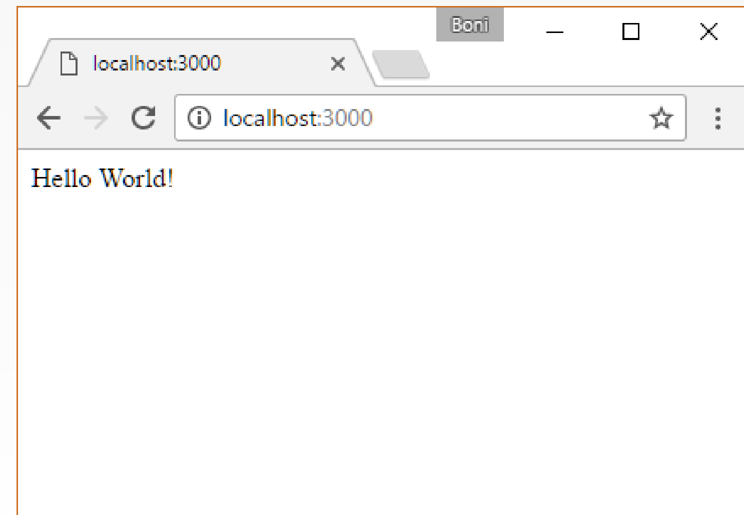
Node.js

Stack MEAN

- Vamos a ver el ejemplo más sencillo posible (*hello world*) usando Node.js y Express.js:

```
> node app.js
Example app listening on port 3000!
```

Una vez resultas las dependencias, podemos ejecutar la aplicación con el comando `node app.js`



Node.js

Ecosistema Node.js

- Existe todo un ecosistema de módulos Node.js (sobre todo para desarrollo de *front-end*). Por ejemplo:
  - **Bower**: Gestiona las dependencias (librerías) de las aplicaciones front-end
  - **Grunt/Gulp**: Ejecutan tareas relacionadas con la ejecución y construcción de proyectos front-end
  - **Yeoman**: Generador de código inicial (*scaffolding*) de proyectos front-end
  - **Angular-cli**: Herramienta de consola de comandos que facilita el desarrollo de aplicaciones Angular
  - …

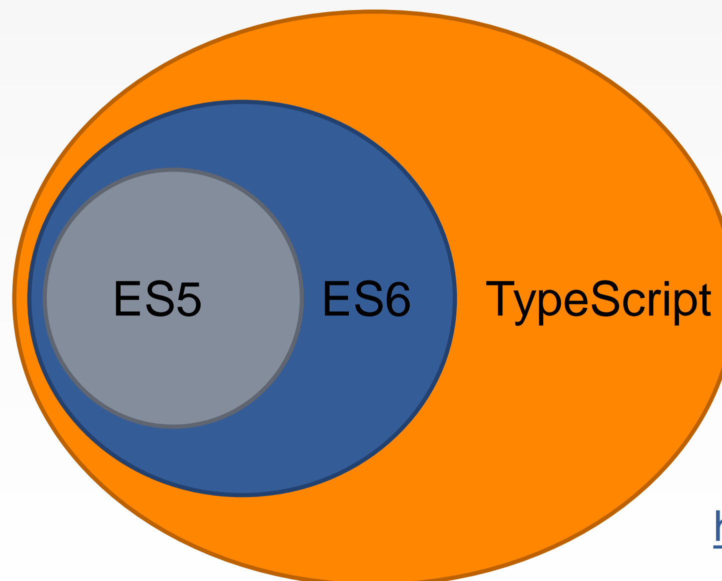
Índice

1. Introducción
2. Node.js
3. TypeScript
   - Introducción
   - Sintaxis TypeScript
4. Angular
5. Librerías de componentes

TypeScript

Introducción

- TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft
- Es un superconjunto de ECMAScript 6



TypeScript

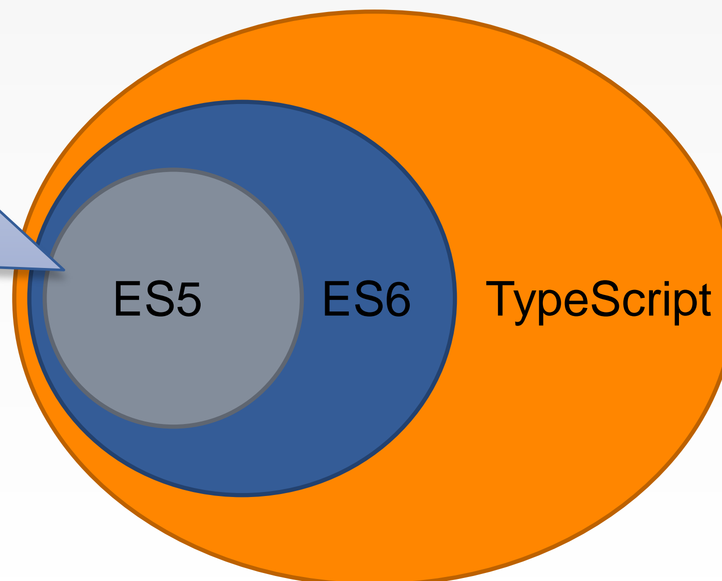
https://www.typescriptlang.org/

TypeScript

Introducción

- TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft
- Es un superconjunto de ECMAScript 6

ECMAScript 5 define la versión de JavaScript que todos los navegadores implementan en la actualidad

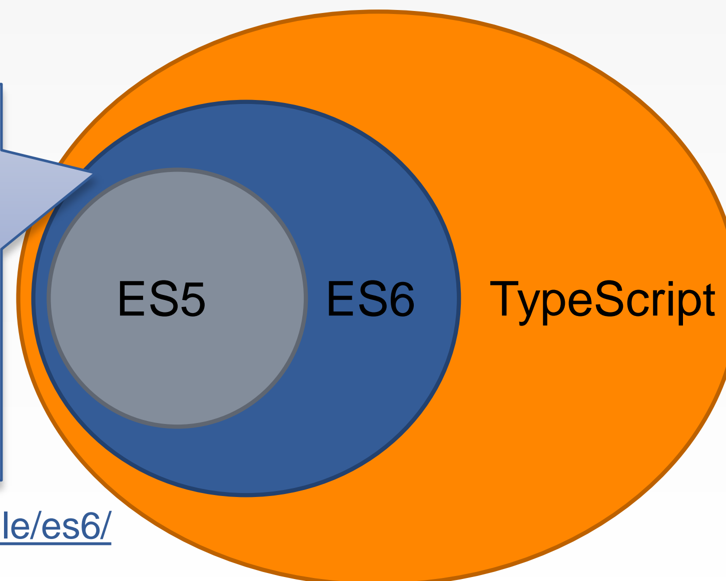


TypeScript

Introducción

- TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft
- Es un superconjunto de ECMAScript 6

ECMAScript 6 define la siguiente versión de JavaScript. Está siendo implementada paulatinamente en los navegadores



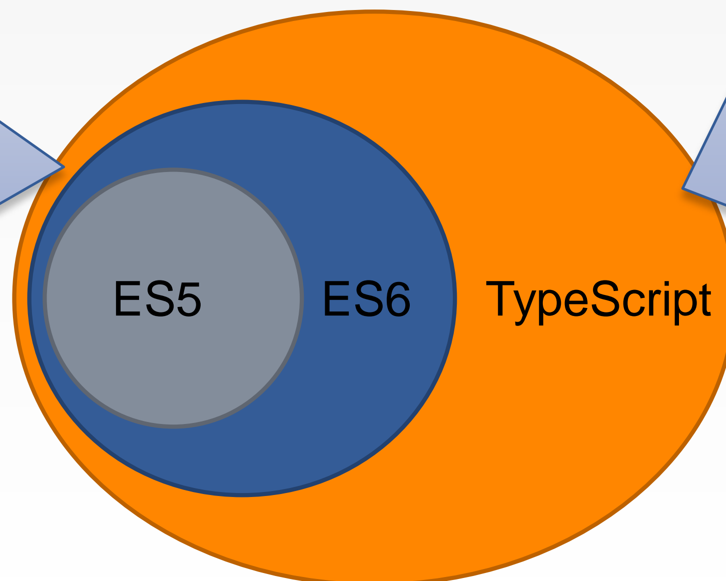
http://kangax.github.io/compat-table/es6/

TypeScript

Introducción

- TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft
- Es un superconjunto de ECMAScript 6

TypeScript reutiliza ECMAScript 6 y añade otras funciones más (modificadores de acceso, interfaces, tipado estático, etc)



Ventajas TypeScript:
- Código más robusto (menos propenso a errores)
- Facilidad de desarrollo (detección de errores en tiempo de compilación, IDEs permiten autocompletar, refactorizar, navegar)

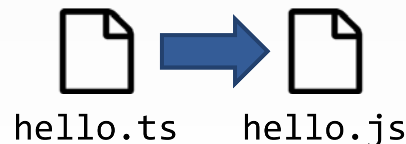
TypeScript

Introducción

- El código TypeScript se almacena en ficheros con extensión `.ts`
- Dado que los navegadores no entienden ECMAScript 6 (ni TypeScript), para que el código TypeScript pueda ser ejecutado en navegadores, es necesario transformar los ficheros `.ts` a ECMAScript 5. Este proceso recibe el nombre de ***transpilación***
- Podemos instalar el transpilador de TypeScript (`tsc`) mediante NPM:

```
> npm install -g typescript
```

```
> tsc hello.ts
```



TypeScript

Sintaxis TypeScript

- TypeScript es un superconjunto de ES6, o sea, añade funcionalidad extra a ES6
- En primer lugar, TypeScript permite la declaración explicita de las propiedades de una clase mediante **modificadores de acceso**

```
class Person {
   private firstName = "";
   private lastName = "";

   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }
}
```

Los modificadores que se pueden usar (tanto en propiedades como en métodos) son:

- `public` : miembro visible desde fuera de la clase (modificador por defecto)
- `private` : miembro no visible desde fuera
- `protected` : miembro sólo visible desde dentro de la clases y en clases hija

https://www.typescriptlang.org/docs/tutorial.html

TypeScript

Sintaxis TypeScript

- Hay un tercer caso en el cual se pueden usar modificadores de acceso: en la declaración de atributos de constructor
- De esta forma, los parámetros del constructor se convierten de forma automática en propiedades de clase

```
class Person {
   private firstName = "";
   private lastName = "";

   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }
}
```

```
class Person {
   constructor(private firstName, private lastName) {
   }
}
```

TypeScript

Sintaxis TypeScript

- TypeScript permite la definición de interfaces mediante la palabra reservada `interface`
- En los interfaces, es posible añadir el símbolo ? junto al nombre de las propiedades para indicar que son opcionales

```
interface Human {
   firstName: string;
   lastName: string;
   name?: Function;
   isLate?(time: Date): Function;
}

class Person implements Human {
   constructor(public firstName, public lastName) {
   }

   public name() {
     return `${this.firstName} ${this.lastName}`;
   }

   protected whoAreYou() {
     return `Hi i'm ${this.name()}`;
   }
}

let john = new Student("John", "Doe");
console.log(john.whoAreYou());
```

TypeScript

Sintaxis TypeScript

- TypeScript permite la declaración de lo que llaman ***decoradores***
- Su función es la misma que las anotación en Java (añadir metadatos a una clase o función)
- Los decoradores se definen usando el símbolo @

```
function Student(config) {
   return function (target) {
      Object.defineProperty(target.prototype,
         'course', {value: () => config.course})
   }
}

@Student({
   course: "angular3"
})
class Person {
   constructor(private firstName, private lastName) {
   }

   public name() {
      return `${this.firstName} ${this.lastName}`;
   }

   protected whoAreYou() {
     return `Hi i'm ${this.name()}`;
   }
}

let john = new Person("John", "Doe");
console.log(john.whoAreYou());
```

TypeScript

Sintaxis TypeScript

- ES6 define la palabra clave `export` para exportar e `import` para importar funciones
- Se puede usar la palabra clave `default` a la hora exportar para simplificar la importación (se evita el uso de { })

```
// export.ts

function square(x) {
   return Math.pow(x,2)
}

function cow() {
   console.log("Mooooo!!!")
}

export {square, cow};
```

```
// import.ts

import {square, cow} from './export';
console.log(square(2));
cow();

import {square as sqr} from './utils';
sqr(2);

import * as utils from './utils';
console.log(utils.square(2));
utils.cow();
```

```
// export2.ts

export default function
square(x) {
   return Math.pow(x,2)
}
```

```
// import2.ts

import square from './export';
```

TypeScript

Sintaxis TypeScript

- Una de las características más importantes que añade TypeScript a ES6 es el **tipado estático**
- El tipado estático nos permite que nuestros programas sean más robustos, ya que problemas con los tipos se detectan en tiempo de transpilación
- De otro modo, estos problemas se conocerían en tiempo de ejecución
  - Los defensores del tipado dinámico argumenta que este problema se soluciona simplemente haciendo más pruebas
- El inconveniente de esta técnica es que se pierde parte de la flexibilidad de JavaScript nativo

TypeScript

Sintaxis TypeScript

Typescript permite los tipos básicos `number`, `boolean` y `string`

Los tipos de los arrays se pueden especificar de dos formas

Mediante la palabra reservada `Function` se define el tipo función

Podemos especificar el tipo de debe devolver una función

Cuando una función no devuelve nada, usamos la palabra reservada `void`

```
// Basic types
let decimal: number = 6;
let done: boolean = false;
let color: string = "blue";

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

// Functions
let fun: Function = () => console.log("Hello");

// Expected return types
function returnNumber(): number {
   return 1;
}

// Void
function returnNothing(): void {
   console.log("Moo");
}
```

TypeScript

Sintaxis TypeScript

Los tipos enumerados se soportan de forma nativa en ES6. Podemos usar estos tipos enumerados en la defunción de variables de ese tipo

Podemos definir tipos de un objeto que previamente hayamos creado

Si no sabemos exactamente el tipo de una determinada variable, usamos la palabra reservada any

```
// Enums
enum Direction {
   Up,
   Down,
   Left,
   Right
}

let go: Direction;
go = Direction.Up;

// Class
let person: Person;
let people: Person[];

// Any
let notsure: any = 1;
```

Índice

1. Introducción
2. Node.js
3. TypeScript
4. Angular
   - Angular CLI
   - Aplicación básica Angular
   - Componentes
   - Data binding
   - Directivas
   - Inyección de dependencias
   - Servicios REST
   - Routing
   - Otras *features* de Angular
5. Librerías de componentes

Angular

Angular CLI

- Angular CLI (*command line interface*) es una herramienta que facilita el desarrollo de aplicaciones Angular
- Ha sido desarrollado en Node.js, con lo que la instalamos en nuestro sistema a través de NPM

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

> ng -v
    _                      _                 ____ _     ___
   / \   _ __   __ _ _   _| | __ _ _ __     / ___| |   |_ _|
  / △ \ | '_ \ / _` | | | | |/ _` | '__|   | |   | |    | |
 / ___ \| | | | (_| | |_| | | (_| | |      | |___| |___ | |
/_/   \_\_| |_|\__, |\__,_|_|\__,_|_|       \____|_____|___|
               |___/
Angular CLI: 1.7.4
Node: 8.11.1
OS: win32 x64
Angular:
...
```

https://cli.angular.io/

Angular

Angular CLI

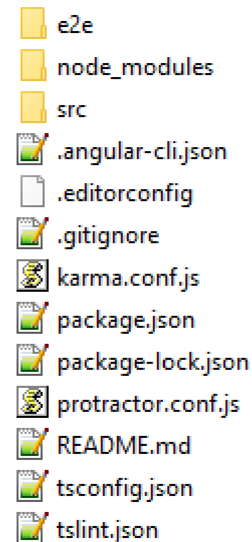
- Angular CLI nos proporciona las siguientes funcionalidades:
  - Creación inicial de la **estructura de un proyecto** (este proceso a veces e conoce como *bootstrapping* o también como *scaffolding*)
  - Servir nuestra aplicación desde un **servidor web de desarrollo** y **recarga automática** (*live reloading*)
  - **Generación automática de código** (creación de directivas, servicios, etc)
  - Automatización de **pruebas** (para ello se usa el framework de pruebas JavaScript Jasmine https://jasmine.github.io/)
  - **Empaquetado y despliegue** de la aplicación

Angular

Angular CLI

- Para la creación de la estructura inicial de un proyecto (*bootstrapping*) usamos el comando `ng new`
- La estructura de carpetas y ficheros creados con este comando es la siguiente:

```
> ng new angular-hello-world
  installing ng
  create angular-hello-world/e2e/app.e2e-spec.ts (301 bytes)
  create angular-hello-world/e2e/app.po.ts (208 bytes)
  create angular-hello-world/e2e/tsconfig.e2e.json (235 bytes)
  ...

added 1259 packages in 131.051s
Project 'angular-hello-world' successfully created.
```



Angular

Angular CLI

- Vamos a estudiar la estructura de directorios y ficheros que se han creado automáticamente con `ng new`

```
|   .angular-cli.json
|   .editorconfig
|   .gitignore
|   .project
|   karma.conf.js
|   package-lock.json
|   package.json
|   protractor.conf.js
|   README.md
|   tsconfig.json
|   tslint.json
+---e2e
|       app.e2e-spec.ts
|       app.po.ts
|       tsconfig.e2e.json
\---src
    |   favicon.ico
    |   index.html
    |   main.ts
    |   polyfills.ts
    |   styles.css
    |   test.ts
    |   tsconfig.app.json
    |   tsconfig.spec.json
    |   typings.d.ts
    +---app
    |       app.component.css
    |       app.component.html
    |       app.component.spec.ts
    |       app.component.ts
    |       app.module.ts
    +---assets
    \---environments
```

Angular

Angular CLI

En la raíz del proyecto nos encontramos con diferentes ficheros, principalmente para configuración general y documentación

Las pruebas de sistema *end-to-end* van en la carpeta `e2e`

```
|    .angular-cli.json
|    .editorconfig
|    .gitignore
|    .project
|    karma.conf.js
|    package-lock.json
|    package.json
|    protractor.conf.js
|    README.md
|    tsconfig.json
|    tslint.json
+---e2e
|        app.e2e-spec.ts
|        app.po.ts
|        tsconfig.e2e.json
\---src
    |    favicon.ico
    |    index.html
    |    main.ts
    |    polyfills.ts
    |    styles.css
    |    test.ts
    |    tsconfig.app.json
    |    tsconfig.spec.json
    |    typings.d.ts
    +---app
    |        app.component.css
    |        app.component.html
    |        app.component.spec.ts
    |        app.component.ts
    |        app.module.ts
    +---assets
    \---environments
```

Angular CLI trabaja con NPM para gestionar las dependencias. Por lo tanto, nos encontramos con el fichero `package.json`

Como ya hemos visto, las dependencias de la aplicación se almacenarán en la carpeta `node_modules`

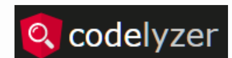
Angular

Angular CLI

- En el fichero `package.json` están definidas todas las dependencias de nuestra aplicación

```
"dependencies": {
    "@angular/animations": "^5.2.0",
    "@angular/common": "^5.2.0",
    "@angular/compiler": "^5.2.0",
    "@angular/core": "^5.2.0",
    "@angular/forms": "^5.2.0",
    "@angular/http": "^5.2.0",
    "@angular/platform-browser": "^5.2.0",
    "@angular/platform-browser-dynamic": "^5.2.0",
    "@angular/router": "^5.2.0",
    "core-js": "^2.4.1",
    "rxjs": "^5.5.6",
    "zone.js": "^0.8.19"
  },
  "devDependencies": {
    "@angular/cli": "~1.7.4",
    "@angular/compiler-cli": "^5.2.0",
    "@angular/language-service": "^5.2.0",
    "@types/jasmine": "~2.8.3",
    "@types/jasminewd2": "~2.0.2",
    "@types/node": "~6.0.60",
    "codelyzer": "^4.0.1",
    "jasmine-core": "~2.8.0",
    "jasmine-spec-reporter": "~4.2.1",
    "karma": "~2.0.0",
    "karma-chrome-launcher": "~2.2.0",
    "karma-coverage-istanbul-reporter": "^1.2.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~5.1.2",
    "ts-node": "~4.1.0",
    "tslint": "~5.9.1",
    "typescript": "~2.5.3"
  }
```

Hay que distinguir entre las as dependencias que se van a ser requeridas por la aplicación en un entorno de producción (`dependencies`) mientras que hay otras que sólo son necesarios en el entorno de desarrollo (`devDependencies`)



Angular

Angular CLI

El contenido de nuestra aplicación irá dentro de la carpeta `src`

Aquí nos encontraremos el fichero inicial de nuestra aplicación (`index.html`), el icono (`favicon.ico`), el fichero TypeScript principal (`main.ts`) o los estilos CSS globales (`styles.css`)

```
|   .angular-cli.json
|   .editorconfig
|   .gitignore
|   .project
|   karma.conf.js
|   package-lock.json
|   package.json
|   protractor.conf.js
|   README.md
|   tsconfig.json
|   tslint.json
+---e2e
|       app.e2e-spec.ts
|       app.po.ts
|       tsconfig.e2e.json
\---src
    |   favicon.ico
    |   index.html
    |   main.ts
    |   polyfills.ts
    |   styles.css
    |   test.ts
    |   tsconfig.app.json
    |   tsconfig.spec.json
    |   typings.d.ts
    +---app
    |       app.component.css
    |       app.component.html
    |       app.component.spec.ts
    |       app.component.ts
    |       app.module.ts
    +---assets
    \---environments
```

En el directorio `src/app` es donde irán los componentes TypeScript (que como veremos son las piezas fundamentales de las aplicaciones Angular):

- `*.css`: estilos CSS para el componente
- `*.html`: plantilla para el componente
- `*.spec.ts`: pruebas unitarias para el componente (ejecutadas con Karma)
- `*.ts`: lógica del componente
- `app.module.ts`: definición de los módulos que usa la aplicación

Angular

Angular CLI

```
|   .angular-cli.json
|   .editorconfig
|   .gitignore
|   .project
|   karma.conf.js
|   package-lock.json
|   package.json
|   protractor.conf.js
|   README.md
|   tsconfig.json
|   tslint.json
+---e2e
|       app.e2e-spec.ts
|       app.po.ts
|       tsconfig.e2e.json
\---src
    |   favicon.ico
    |   index.html
    |   main.ts
    |   polyfills.ts
    |   styles.css
    |   test.ts
    |   tsconfig.app.json
    |   tsconfig.spec.json
    |   typings.d.ts
    +---app
    |       app.component.css
    |       app.component.html
    |       app.component.spec.ts
    |       app.component.ts
    |       app.module.ts
    +---assets
    \---environments
```

En `environments` va la configuración para entornos de desarrollo y producción

En el directorio `assets` irán los ficheros que queremos incluir en nuestra aplicación cuando se empaquete para producción (imágenes…)

Angular

Angular CLI

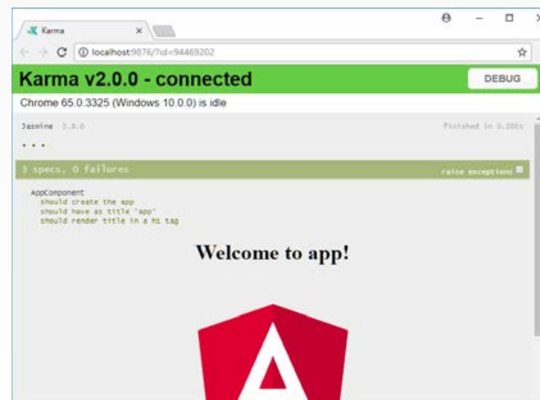
- Angular CLI nos ofrece otros comandos:
  - `ng serve` : Publica nuestra aplicación a través de un servidor web de pruebas. Por defecto, nuestra aplicación estará en http://localhost:4200/

```
> ng serve
** NG Live Development Server is running on http://localhost:4200 **
Date: 2018-04-18T23:21:15.901Z
Hash: c2dd8983cea6c5fe5a21
Time: 6789ms
chunk    {0} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 165 kB {4} [initial] [rendered]
chunk    {1} main.bundle.js, main.bundle.js.map (main) 3.69 kB {3} [initial] [rendered]
chunk    {2} styles.bundle.js, styles.bundle.js.map (styles) 9.77 kB {4} [initial] [rendered]
chunk    {3} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.39 MB [initial] [rendered]
chunk    {4} inline.bundle.js, inline.bundle.js.map (inline) 0 bytes [entry] [rendered]
webpack: Compiled successfully.
```

Angular

Angular CLI

- Angular CLI nos ofrece otros comandos:
  - `ng generate` : Nos permite generar código fuente estructural (para componentes, directivas, pipes, servicios, clases, interfaces, o enumerados)
  - `ng build` : Nos permite empaquetar (*bundle*) nuestra aplicación, típicamente para ser desplegada en un entorno de producción
  - `ng test` : Nos permite ejecutar las pruebas



Angular

Angular CLI

- Podemos usar la opción `--minimal` para generar una estructura más reducida de proyectos (sin pruebas, etc)

```
|   .angular-cli.json
|   .editorconfig
|   .gitignore
|   .project
|   karma.conf.js
|   package-lock.json
|   package.json
|   protractor.conf.js
|   README.md
|   tsconfig.json
|   tslint.json
+---e2e
|       app.e2e-spec.ts
|       app.po.ts
|       tsconfig.e2e.json
\---src
    |   favicon.ico
    |   index.html
    |   main.ts
    |   polyfills.ts
    |   styles.css
    |   test.ts
    |   tsconfig.app.json
    |   tsconfig.spec.json
    |   typings.d.ts
    +---app
    |       app.component.css
    |       app.component.html
    |       app.component.spec.ts
    |       app.component.ts
    |       app.module.ts
    +---assets
    \---environments
```

Angular

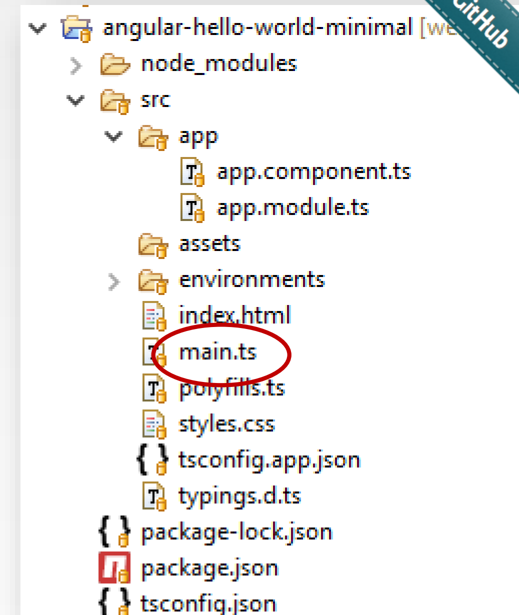
Aplicación básica Angular

- Vemos en detalles la aplicación generada con:

```
> ng new angular-hello-world --minimal
```

main.ts

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

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

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



El punto de inicio de la aplicación lo establece el fichero `main.ts` ubicado en la carpeta `src`

En este fichero encontramos la declaración del modulo de arranque (`bootstrapModule`)

Angular

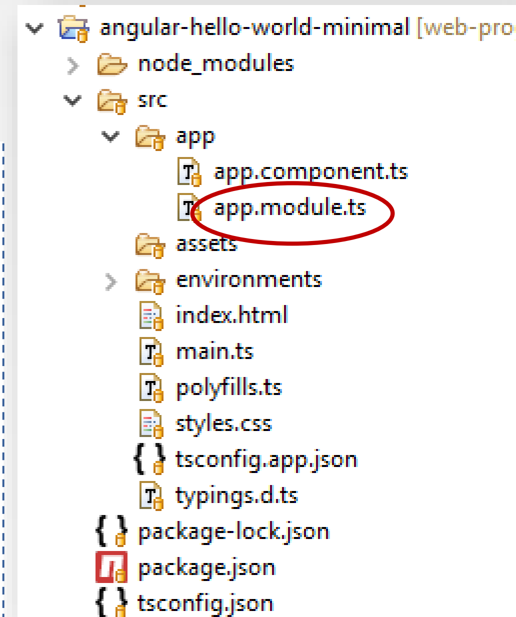
Aplicación básica Angular

En primer lugar, importamos los módulos nativos de angular que vamos a usar. Por defecto, Angular CLI añade:

- `BrowserModule` : utilidades Angular para navegadores web
- `NgModule` : decorador para declara el módulo princial

app.module.ts

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


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


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



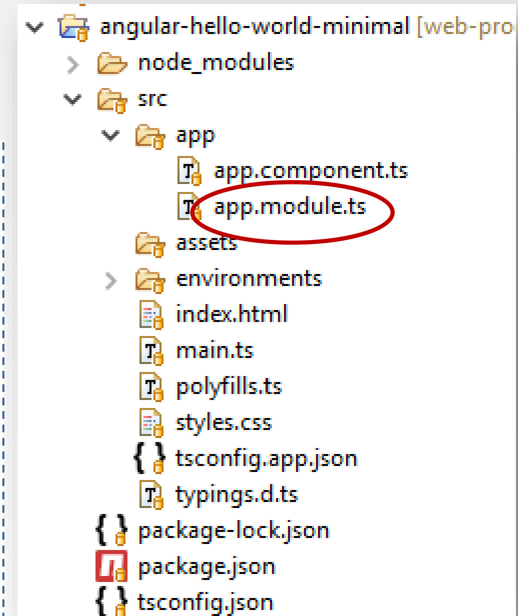
Angular

Aplicación básica Angular

En segundo lugar, importamos los módulos específicos de nuestra aplicación. En el ejemplo se importa el componente AppComponent localizado en ./app.component

app.module.ts

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


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


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



Angular

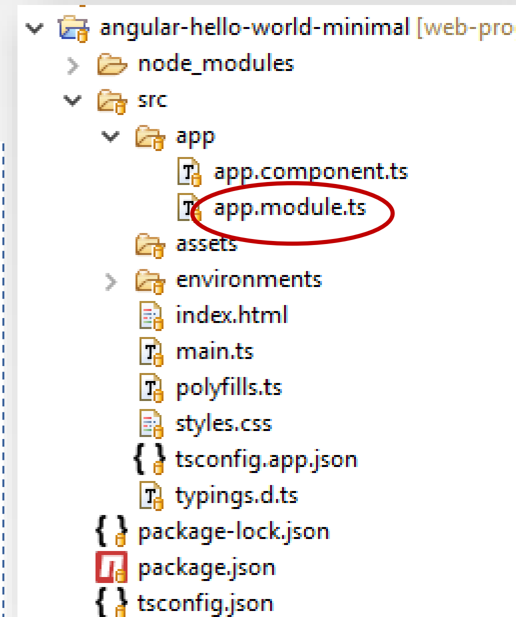
Aplicación básica Angular

Toda aplicación de Angular tiene un módulo principal que se define anotando una clase con @NgModule. En este decorador se definen los siguientes metadatos:

- `declarations`: Lista de componentes (con vista)
- `imports`: Lista de clases que van a ser usadas en el módulo
- `providers`: Lista de servicios que se definen en el módulo
- `bootstrap`: Componente raíz

app.module.ts

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


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


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



Angular

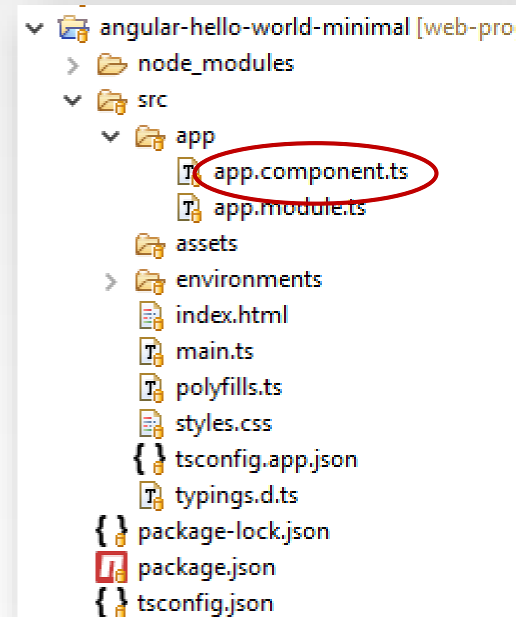
Aplicación básica Angular

Cada componente se define una clase TypeScript anotada con `@Component`. Este decorador define varias propiedades:

- `selector`: Nombre del tag HTML que se usará en la plantilla
- `template`: Contenido plantilla del componente (en su lugar se puede especificar la ruta mediante `templateUrl`)
- `styles`: Estilos CSS propios del componente (se puede poner rutas con `styleUrls`)

app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
      ...
      <h1>
        Welcome to {{title}}!
      </h1>
      ...
  `,
  styles: []
})
export class AppComponent {
  title = 'app';
}
```



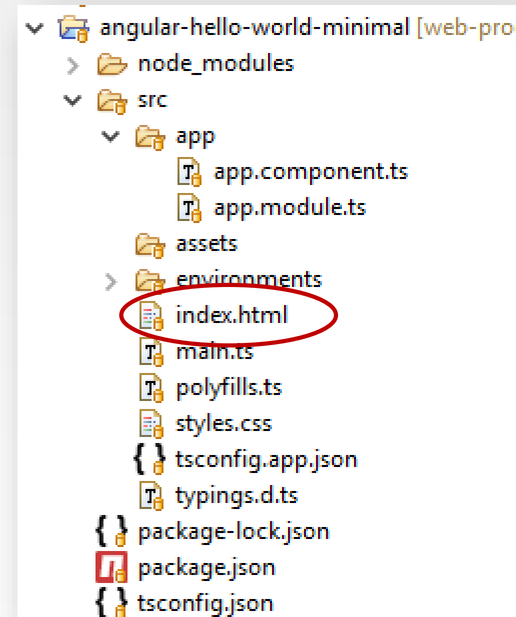
Dentro de la clase TypScript va la lógica del componente. En este ejemplo lo único que se hace es declarar una propiedad llamada `title` con el contenido `'app'`, que después se lee en la plantilla usando la directiva `{{ }}`

Angular

Aplicación básica Angular

index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularHelloWorldMinimal</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```



En la página inicial (`index.html`) se declara el componente raíz, simplemente incluyendo el tag `app-root`

Angular

Componentes

- Angular sigue un patrón similar a **MVC** para el desarrollo de aplicaciones
  - Se dice que Angular sigue realmente el patrón MVVC porque también hay comunicación desde la vista al controlador
- Podemos hacer una analogía con algo que ya conocemos (Spring)

	Spring	**Angular**
Controlador	Clases Java anotadas con `@Controller`	Clases TypeScript anotadas con `@Component`
Vista	Plantillas (nosotros hemos visto Thymeleaf)	Plantillas HTML (con *directivas* Angular)
Modelo	Mapa de datos devuelto por los métodos del controlador (tipo `ModelAndView`)	Estado de los componentes (valor de las propiedades y métodos definidos en la clase TypeScript)

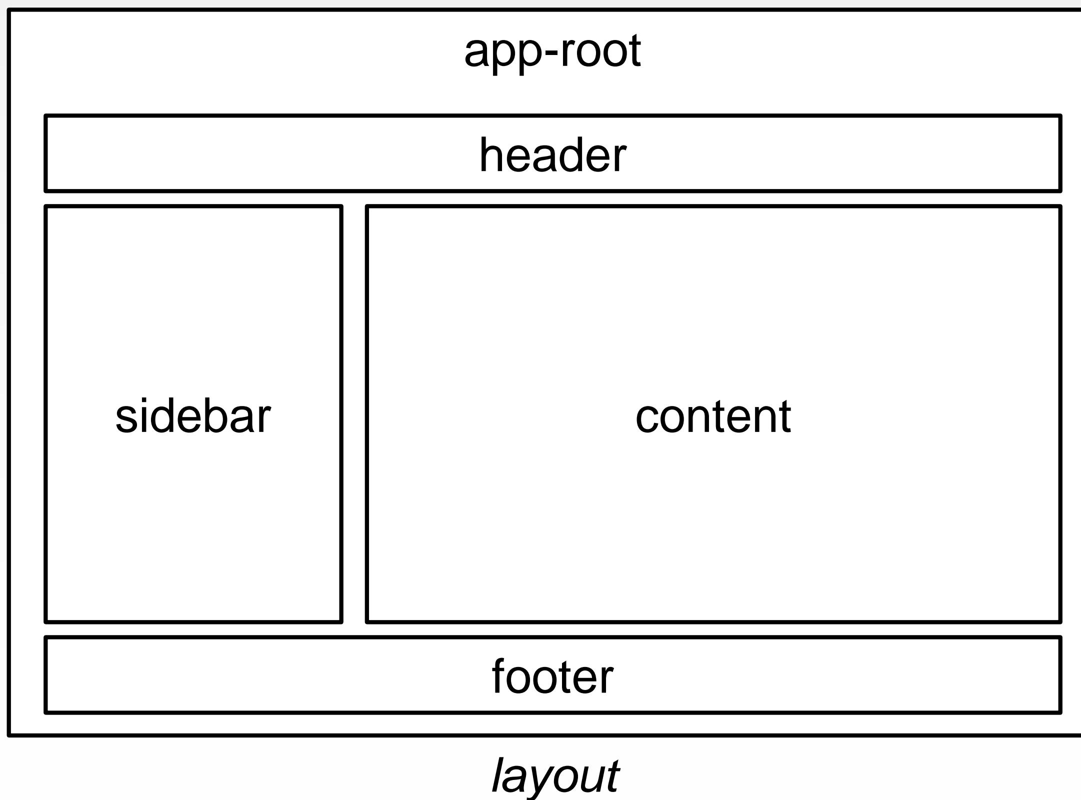
Angular

Componentes

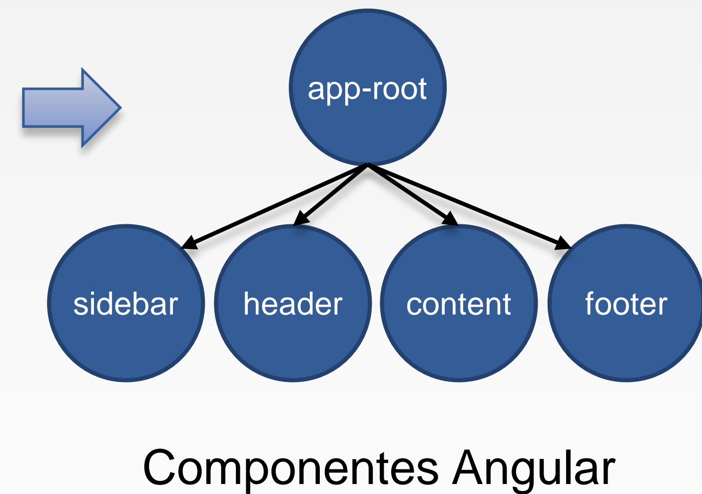
- Los componentes son las piezas fundamentales en la construcción de aplicaciones Angular
- Como vamos a ver, la **arquitectura** de una aplicación Angular se mediante la **composición de componentes**
- De momento vamos a ver una pequeña aplicación SPA con una disposición de elementos (*layout*) determinada
- Vamos a crear la estructura de esta aplicación así como los componentes de la misma con Angular CLI

Angular

Componentes



layout



Componentes Angular

Angular

Componentes

- Vamos a generar la estructura del proyecto mediante el comando:

```
> ng new angular-components --minimal
```

- Luego generamos los componentes también desde línea de comandos:

```
> ng generate component header
> ng generate component sidebar
> ng generate component content
> ng generate component footer
```

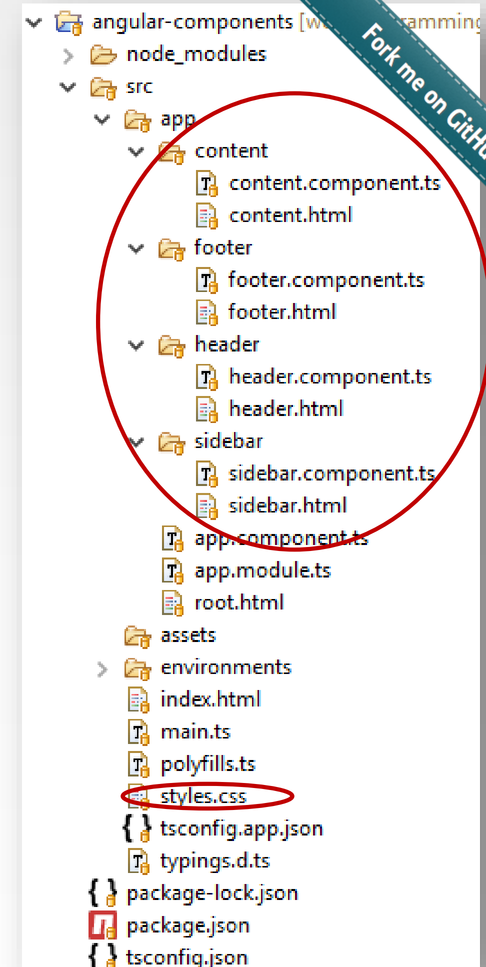
Angular

Componentes

- Partiendo del proyecto creado con Angular CLI, modificamos los componentes generados (localizados en la carpeta `src/app`)
- Por simplicidad, vamos a modificar los estilos a nivel global de la aplicación, editando el fichero `styles.css` localizado en la raíz del proyecto

`styles.css`

```
.container {
    width: 900px;
    margin: 0 auto;
}

.content {
    float: left;
    width: 100%;
    position: relative;
}

...
```



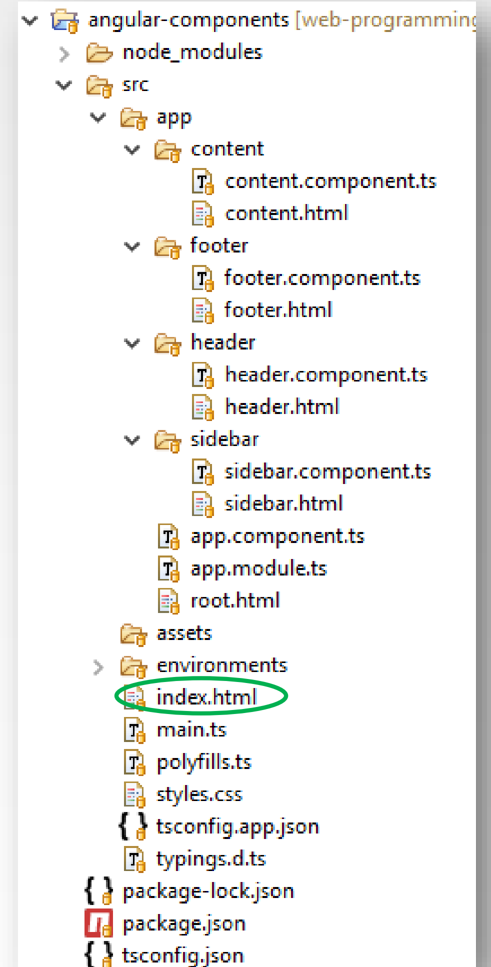
Angular

Componentes

index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularComponents</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Para este ejemplo, el contenido de `index.html` nos vale tal cual lo genera ng new



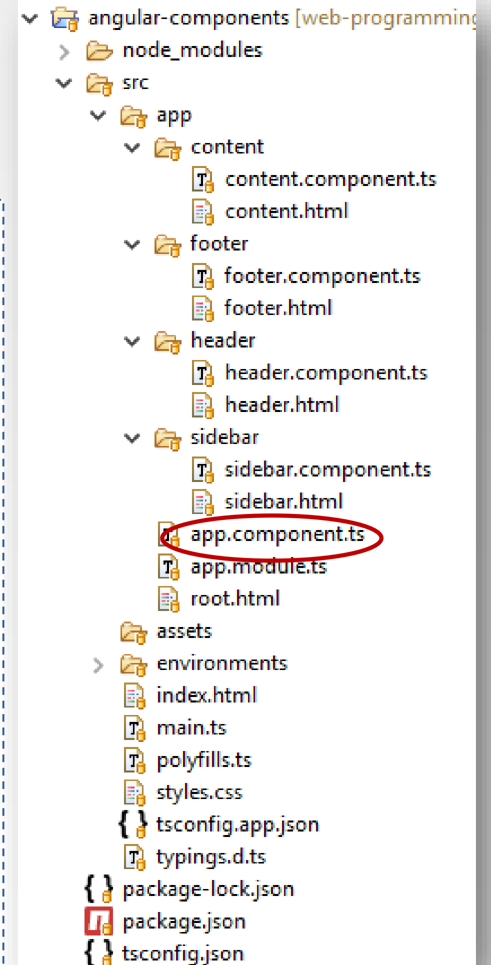
Angular

Componentes

app.module.ts

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


import { AppComponent } from './app.component';
import { HeaderComponent } from './header/header.component';
import { SidebarComponent } from './sidebar/sidebar.component';
import { ContentComponent } from './content/content.component';
import { FooterComponent } from './footer/footer.component';


@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent,
    SidebarComponent,
    ContentComponent,
    FooterComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Comprobamos que todos los componentes de la aplicación están declarados en el fichero `app.module.ts`



Angular

Componentes

root.component.ts

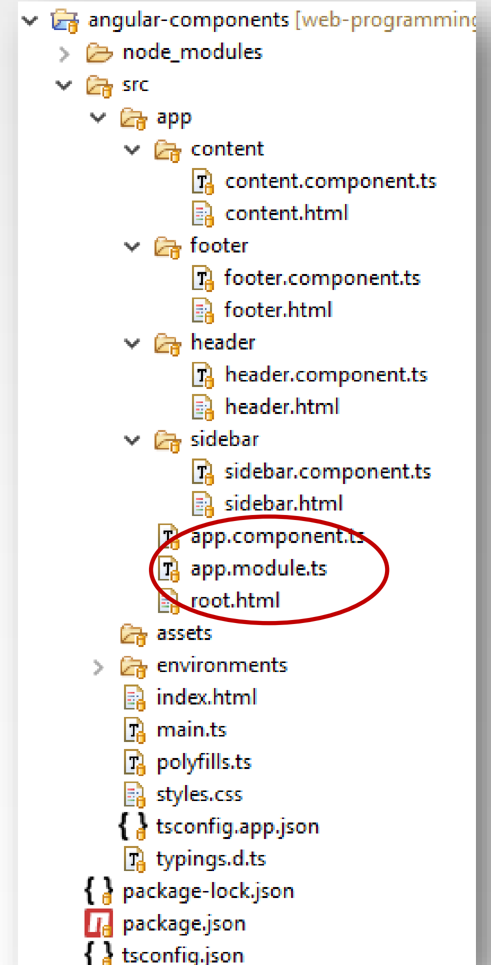
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: 'root.html',
  styles: []
})
export class AppComponent {
  title = 'app';
}
```

Para una organización más limpia del código, implementamos las plantillas en un fichero independiente (usando `templateUrl` en el component)

root.html

```
<div class="container">
   <header>Loading...</header>

   <div class="content">
      <sidebar>Loading...</sidebar>
       <content>Loading...</content>
   </div>

   <footer>Loading...</footer>
</div>
```

La plantilla raíz defimos el layout de nuestra aplicación, usando el selector de cada componente Angular



Angular

Componentes

content.component.ts

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

@Component({
  selector: 'app-content',
  templateUrl: 'content.html',
  styles: []
})
export class ContentComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

content.html

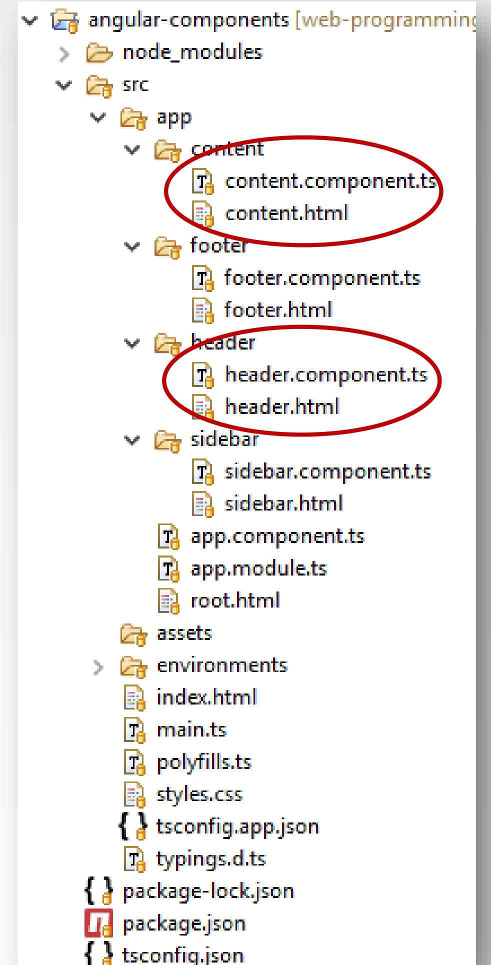
```
<div class="main">
   <h2>Title</h2>
   <p>Lorem ipsum...</p>
</div>
```

header.component.ts

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

@Component({
  selector: 'app-header',
  templateUrl: 'header.html',
  styles: []
})
export class HeaderComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

header.html

```
<div class="header">
   <h1>Header</h1>
</div>
```



Angular

Componentes

sidebar.component.ts

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

@Component({
  selector: 'app-sidebar',
  templateUrl: 'sidebar.html',
  styles: []
})
export class SidebarComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

sidebar.html

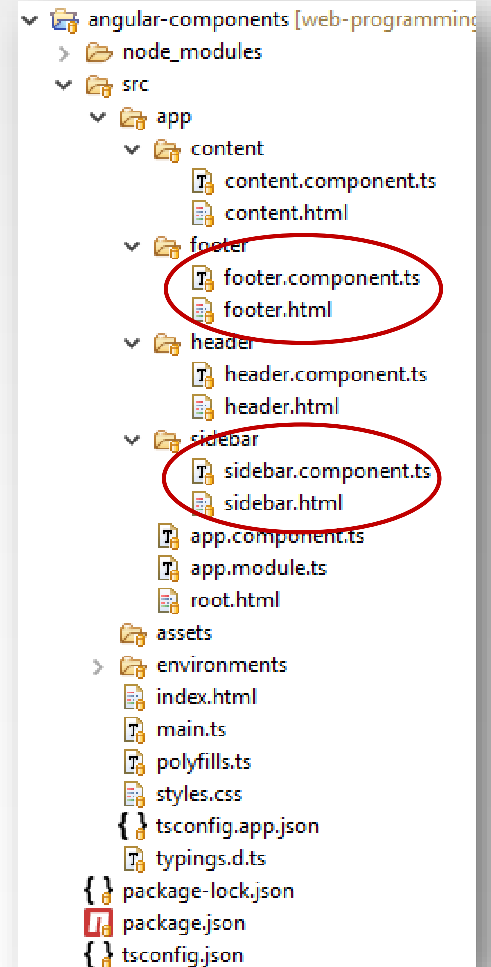
```
<div class="nav">
Section 1<br> Section 2<br> Section 3
</div>
```

footer.component.ts

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

@Component({
  selector: 'app-footer',
  templateUrl: 'footer.html',
  styles: []
})
export class FooterComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

footer.html

```
<div class="footer">Copyright &copy;
Company.com</div>
```

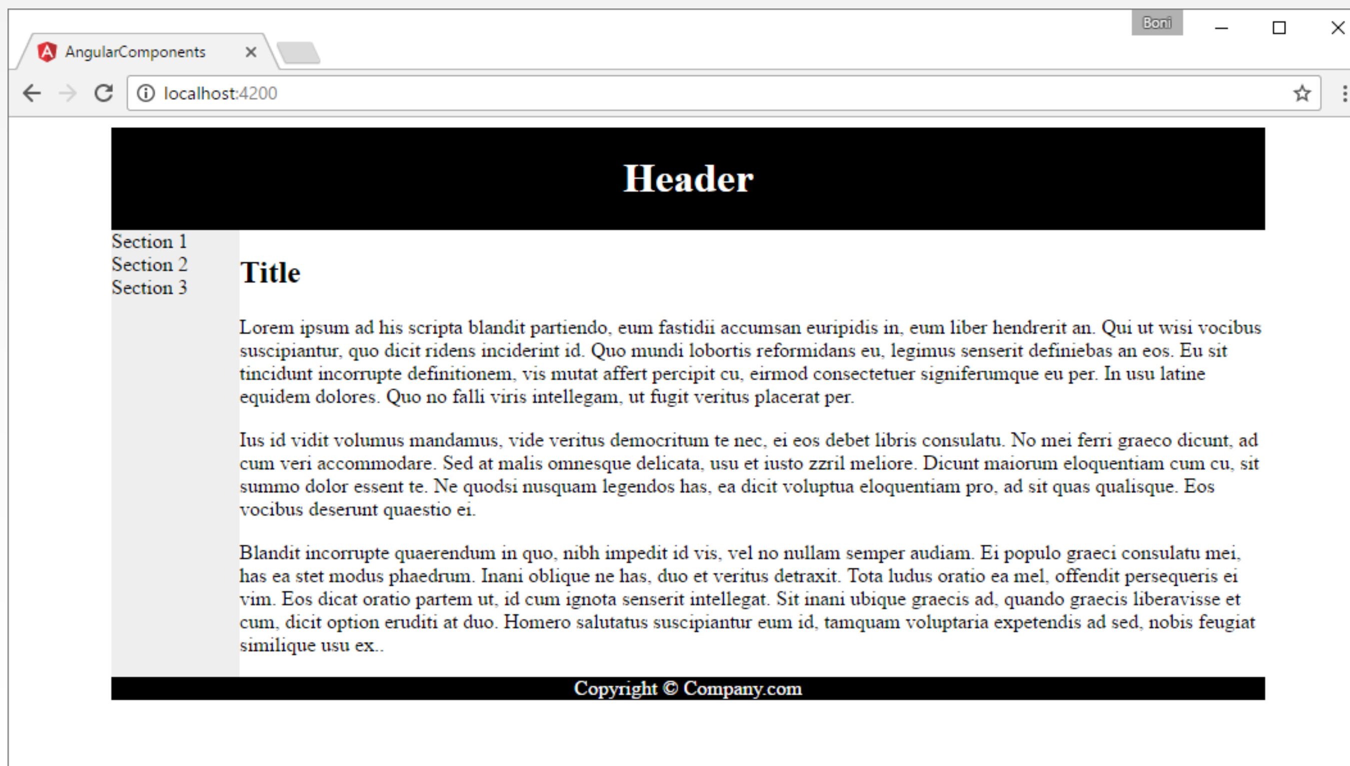


Angular

Componentes

```
> ng serve
```

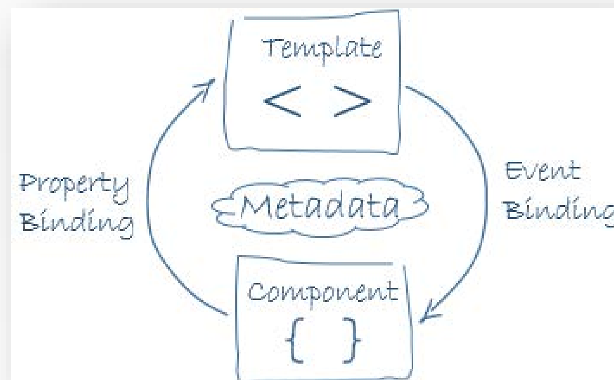
Ejecutamos la aplicación con Angular CLI



Angular

Data binding

- El enlace de datos (*data binding*) es el nombre que se le da a la acción de usar el estado de los componentes (esto es, el *modelo*) para modificar la presentación (esto es, las *plantillas*) y viceversa
- Este mecanismo es uno de los principales ventajas de Angular, ya que nos abstrae de la lógica asociada a la modificación manual del DOM y convertir los eventos de usuario en acciones concretas



Angular

Fork me on GitHub

Data binding

- Angular dispone de 4 formas de hacer data binding:

1. Interpolación
   - Dirección: hacía el DOM
   - Se usa la sintaxis {{ }} para enlazar datos del modelo en la plantilla

app.component.ts

```
import {Component} from '@angular/core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html'
})
export class AppComponent {
    name = 'Anybody';
}
```

app.component.html

```
<h1>Hello {{name}}!</h1>
```

Hello Anybody!

Angular

Fork me on GitHub

Data binding

- Angular dispone de 4 formas de hacer data binding:

2. *Property binding*
    - Dirección: hacía el DOM
    - Se usa la sintaxis [ ] para modificar una propiedad de un elemento de la plantilla

app.component.ts

```
import {Component} from '@angular/core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html'
})
export class AppComponent {
    name = 'Anybody';
    imgUrl = "assets/utad.png";
}
```

app.component.html

```
<img [src]="imgUrl">
```

Angular

Fork me on GitHub

Data binding

- Angular dispone de 4 formas de hacer data binding:

3. *Event binding*
   - Dirección: desde el DOM
   - Se usa la sintaxis `()` junto con el evento que queremos capturar (por ejemplo, `click`, `keydown`, `blur`, etc)
   - Lista completa de eventos: https://www.w3.org/TR/DOM-Level-2-Events/events.html

app.component.ts

```
import {Component} from '@angular/core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html'
})
export class AppComponent {
    name = 'Anybody';

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

app.component.html

```
<h1>Hello {{name}}!</h1>
<button (click)="setName('John')">Hello John</button>
```

Hello John!

Hello John

Angular

Fork me on GitHub

Data binding

- Angular dispone de 4 formas de hacer data binding:

4. *Two-way binding*
   - Dirección: desde/hacía el DOM
   - Se usa la directiva `[(ngModel)]` para enlazar datos del modelo en la plantilla y viceversa

app.component.ts

```
import {Component} from '@angular/core';
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html'
})
export class AppComponent {
    name = 'Anybody';
    imgUrl = "assets/utad.png";

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

app.component.html

```
<h1>Hello {{name}}!</h1>
<button (click)="setName('John')">Hello John</button>
<input type="text" [(ngModel)]="name">
```

Hello 1234567!

Hello John | 1234567

Angular

Fork me on GitHub

Data binding

- Angular dispone de 4 formas de hacer data binding:

4. *Two-way binding*
   - Dirección: desde/hacía el DOM
   - Se usa la directiva `[(ngModel)]` para enlazar datos del modelo en la plantilla y viceversa

app.module.ts

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {FormsModule} from '@angular/forms';
import {AppComponent} from './app.component';

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

Para hacer uso de `ngModel` hay que importar el módulo `FormsModule` de Angular en @NgModule

Angular

Directivas

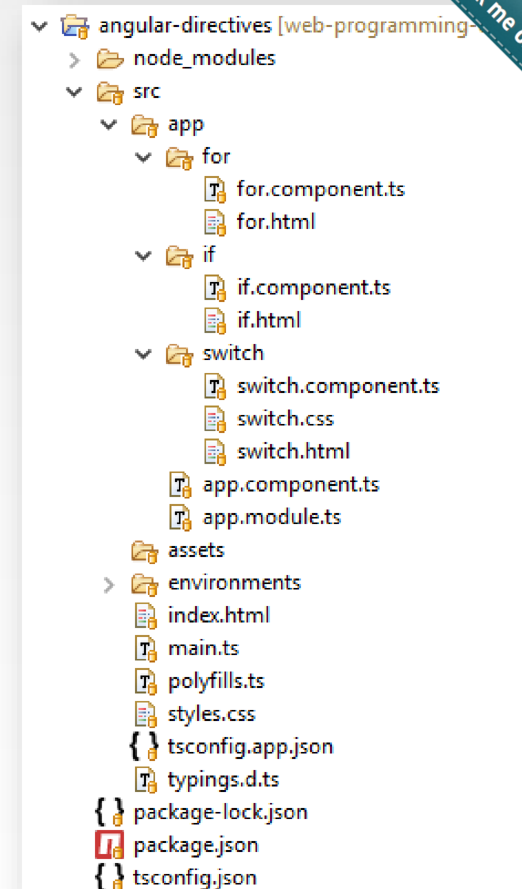
- Las **plantillas** son ficheros HTML que permiten definir la **vista** del componente en función del estado del mismo
- Dentro de las plantillas nos encontraremos con **directivas**, que son mecanismos que permiten realizar diferentes acciones en la plantilla enlazando con el información proveniente de los componentes (lo que llamado *modelo* en MVC)
- Hay dos tipos de directivas:
  - Directivas estructurales. Empiezan con el símbolo `*` y sirven para modificar el DOM
  - Directivas atributos: Van definidas entre los símbolos `[]` y sirven para alterar la apariencia o comportamiento de un elemento del DOM

Angular

Directivas

- Vamos a ver las directivas angular a través de un ejemplo
- El proyecto donde está esta aplicación se ha creado nuevamente con Angular CLI:

```
> ng new angular-directives --minimal
> ng generate component for
> ng generate component if
> ng generate component switch
```



Angular

Directivas

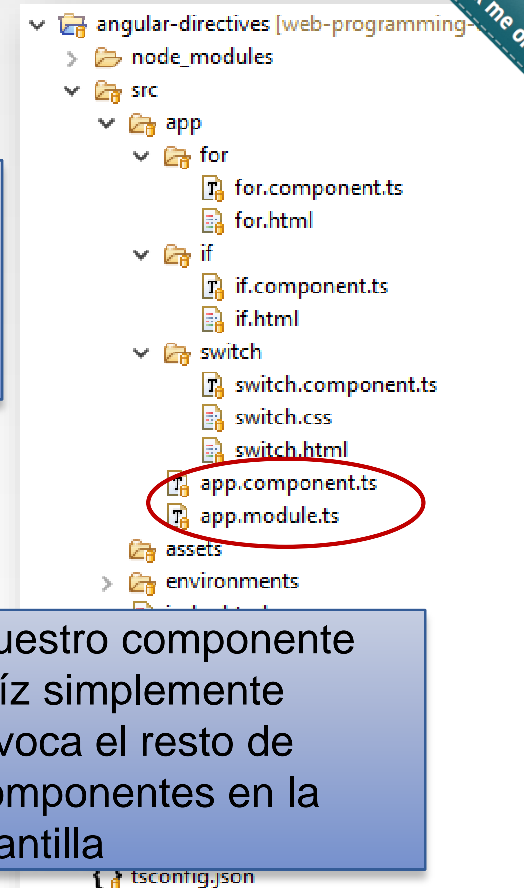
app.module.ts

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


import { AppComponent } from './app.component';
import { ForComponent } from './for/for.component';
import { IfComponent } from './if/if.component';
import { SwitchComponent } from
'./switch/switch.component';


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

Podemos comprobar que los componentes han sido declarados en nuestro @NgModule

app.module.ts

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

@Component({
  selector: 'app-root',
  template: `
    <app-for>Loading...</app-for>
    <app-if>Loading...</app-if>
    <app-switch>Loading...</app-switch>
  `,
  styles: []
})
export class AppComponent {
  title = 'app';
}
```



Nuestro componente raíz simplemente invoca el resto de componentes en la plantilla

Angular

Directivas

- La directiva `ngFor` sirve para recorrer una colección de datos

ngfor.ts

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

@Component({
  selector: 'app-for',
  templateUrl: 'for.html',
  styles: []
})
export class NgFor {
    people: any[] = [
        {"name": "Douglas Pace"},
        {"name": "Mcleod Mueller"},
        {"name": "Day Meyers"},
        {"name": "Aguirre Ellis"},
        {"name": "Cook Tyson"}
    ];
}
```

for.html

```
<h3>NgFor example</h3>
<ul>
   <li *ngFor="let person of people">
      {{ person.name }}
   </li>
</ul>
```

NgFor example

- Douglas Pace
- Mcleod Mueller
- Day Meyers
- Aguirre Ellis
- Cook Tyson

```
<h3>NgFor example (with index)</h3>
<ul>
   <li *ngFor="let person of people;
let i = index">
      {{ i + 1 }} - {{ person.name }}
   </li>
</ul>
```

NgFor example (with index)

- 1 - Douglas Pace
- 2 - Mcleod Mueller
- 3 - Day Meyers
- 4 - Aguirre Ellis
- 5 - Cook Tyson

Angular

Directivas

- La directiva `ngIf` sirve para la visualización condicional

ngif.ts

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

@Component({
  selector: 'app-if',
  templateUrl: 'if.html',
  styles: []
})
export class NgIf {
    people: any[] = [
        {"name": "Douglas Pace",
         "age": 35},
        {"name": "Mcleod Mueller",
         "age": 29},
        {"name": "Day Meyers",
         "age": 21},
        {"name": "Aguirre Ellis",
         "age": 34},
        {"name": "Cook Tyson",
         "age": 32}
    ];
}
```

ngif.html

```
<h3>NgIf example</h3>

<ul>
   <ng-container *ngFor="let person of people">
      <li *ngIf="person.age < 30">{{ person.name }} ({{ person.age }})</li>
   </ng-container>
</ul>
```

NgIf example

- Mcleod Mueller (29)
- Day Meyers (21)

En las plantillas se puede usar la etiqueta `ng-container` cuando necesitemos un grupo que no sea renderizado en el DOM. En este caso lo usamos debido a que Angular no permite el uso de dos directivas `*ng` en el mismo elemento

Angular

Directivas

- La directiva `ngSwitch` sirve para seleccionar una opción

ngswitch.ts

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

@Component({
  selector: 'app-switch',
  templateUrl: 'switch.html',
  styleUrls: ['switch.css']
})
export class NgSwitch {
    people: any[] = [
        {"name": "Douglas Pace",
         "age": 35,
         "country": 'FR'},
        {"name": "Mcleod Mueller",
         "age": 32,
         "country": 'USA'},
        {"name": "Day Meyers",
         "age": 21,
         "country": 'ES'},
        {"name": "Aguirre Ellis",
         "age": 34,
         "country": 'UK'},
        {"name": "Cook Tyson",
         "age": 32,
         "country": 'USA'}
    ];
}
```

ngswitch.html

```
<h3>NgSwitch example</h3>
<ul>
   <ng-container *ngFor="let person of people" [ngSwitch]="person.country">
      <li *ngSwitchCase="'UK'" class="primary">{{ person.name }} ({{
person.country }})</li>
      <li *ngSwitchCase="'USA'" class="warning">{{ person.name }} ({{
person.country }})</li>
      <li *ngSwitchCase="'ES'" class="success">{{ person.name }} ({{
person.country }})</li>
      <li *ngSwitchDefault class="danger">{{ person.name }} ({{
person.country }})</li>
   </ng-container>
</ul>
```

NgSwitch example

- Douglas Pace (FR)
- Mcleod Mueller (USA)
- Day Meyers (ES)
- Aguirre Ellis (UK)
- Cook Tyson (USA)

Angular

Directivas

- La directiva `ngSwitch` sirve para recorrer una colección de datos

ngswitch.ts

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

@Component({
  selector: 'app-switch',
  templateUrl: 'switch.html',
  styleUrls: ['switch.css']
})
export class NgSwitch {
    people: any[] = [
        {"name": "Douglas Pace",
         "age": 35,
         "country": 'FR'},
        {"name": "Mcleod Mueller",
         "age": 32,
         "country": 'USA'},
        {"name": "Day Meyers",
         "age": 21,
         "country": 'ES'},
        {"name": "Aguirre Ellis",
         "age": 34,
         "country": 'UK'},
        {"name": "Cook Tyson",
         "age": 32,
         "country": 'USA'}
    ];
}
```

ngswitch.html

```
<h3>NgStyle example</h3>
<ul>
   <ng-container *ngFor="let person of people" [ngSwitch]="person.country">
      <li [ngStyle]="{'color':person.country === 'ES' ? 'green' : 'red' }">{{
person.name }} ({{ person.country }})</li>
   </ng-container>
</ul>
```

NgStyle example

- Douglas Pace (FR)
- Mcleod Mueller (USA)
- Day Meyers (ES)
- Aguirre Ellis (UK)
- Cook Tyson (USA)

Angular

Directivas

- La directiva `ngSwitch` sirve para recorrer una colección de datos

ngswitch.ts

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

@Component({
  selector: 'app-switch',
  templateUrl: 'switch.html',
  styleUrls: ['switch.css']
})
export class NgSwitch {
    people: any[] = [
        {"name": "Douglas Pace",
         "age": 35,
         "country": 'FR'},
        {"name": "Mcleod Mueller",
         "age": 32,
         "country": 'USA'},
        {"name": "Day Meyers",
         "age": 21,
         "country": 'ES'},
        {"name": "Aguirre Ellis",
         "age": 34,
         "country": 'UK'},
        {"name": "Cook Tyson",
         "age": 32,
         "country": 'USA'}
    ];
}
```

ngswitch.html

```
<h3>NgClass example</h3>
<ul>
   <ng-container *ngFor="let person of people" [ngSwitch]="person.country">
      <li [ngClass]="{'success':person.country === 'USA',
'danger':person.country === 'UK'}">{{ person.name }} ({{ person.country
}})</li>
   </ng-container>
</ul>
```

NgClass example

- Douglas Pace (FR)
- Mcleod Mueller (USA)
- Day Meyers (ES)
- Aguirre Ellis (UK)
- Cook Tyson (USA)

Angular

Inyección de dependencias

- Como ya hemos visto en la parte anterior de la asignatura, la inyección de dependencias es un mecanismo muy usado en el lado servidor (Spring) por sus ventajas:
  - Promueve la reusabilidad
  - Facilita las pruebas
  - Facilita el mantenimiento
  - Facilita la integración (permite diferenciar roles dentro de un equipo de trabajo)

Angular

Inyección de dependencias

- A los elementos de la aplicación que no se encargan del interfaz de usuario se les conoce como **servicios**
- El proceso para crear servicios Angular es el siguiente:
  1. Se crea una nueva clase para el servicio
  2. Se anota esa clase con el decorador `@Inyectable`
  3. Se indica esa clase en la lista de `providers` de la clase principal (anotada con `@NgModule`)
- Después de esto, simplemente habrá que usar este servicio en otra clase. Para ello se inyecta como parámetro en el constructor de la clase que lo consume

Angular

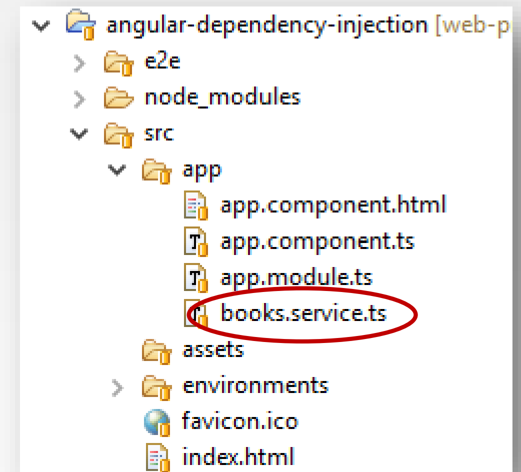
Inyección de dependencias

- Vamos a verlo a través de un ejemplo
- Creamos la estructura del proyecto mediante la línea de comandos:

```
> ng new angular-dependency-injection --minimal
```

- Creamos la estructura del servicio también usando ng:

```
> ng generate service books
```



Angular

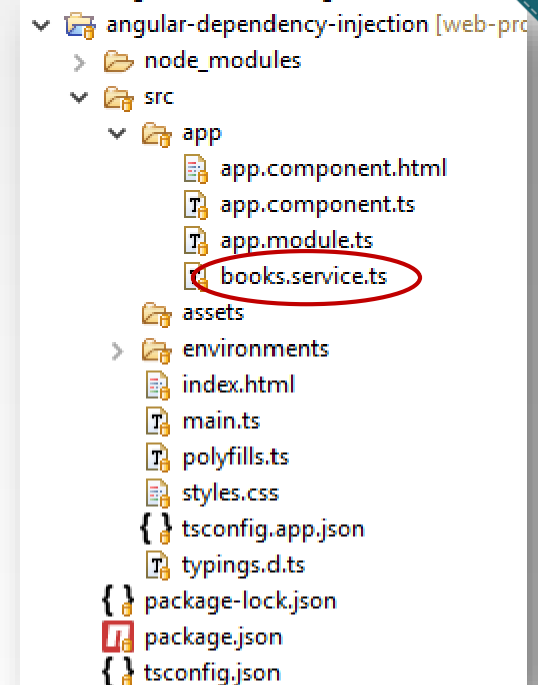
Inyección de dependencias

- Vamos a verlo a través de un ejemplo

La clase que implementa el servicio estará anotada con `@Injectable()`. Esta clase no tiene asociada ninguna plantilla (ya que no tiene vista como ocurría en los componentes)

books.service.ts

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

@Injectable()
export class BooksService {

    private myBooks: string[] = ['Spring in
Action', 'Java for Web Applications', 'Spring
Boot Cookbook'];

    getBooks(key: string) {
        let out: string[] = [];
        for (let book of this.myBooks) {
            if (book.includes(key)) {
                out.push(book);
            }
        }
        return out;
    }

    getAllBooks() {
        return this.myBooks;
    }
}
```



Angular

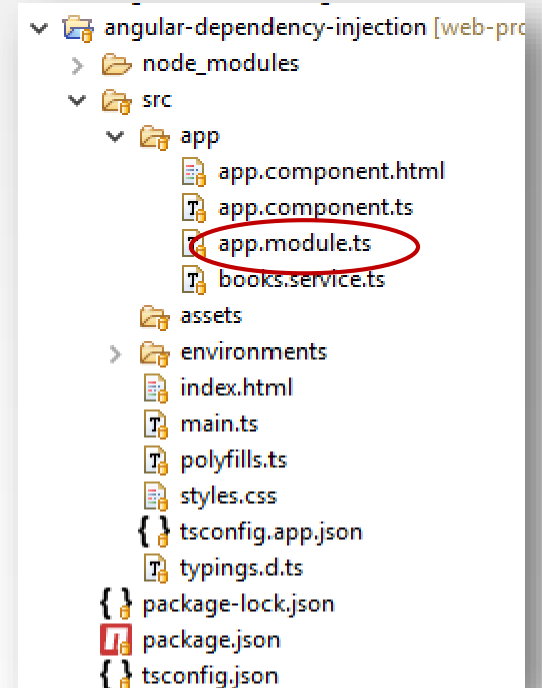
Inyección de dependencias

- Vamos a verlo a través de un ejemplo

app.module.ts

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

import {AppComponent} from './app.component';
import {BooksService} from './books.service';

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

En segundo lugar, añadimos el servicio a la lista de `providers`



Angular

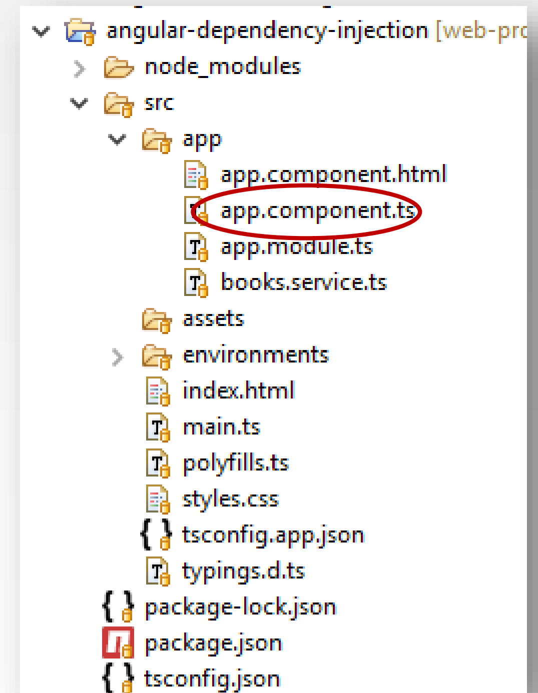
Inyección de dependencias

- Vamos a verlo a través de un ejemplo

app.component.ts

```
import {Component} from '@angular/core';
import {BooksService} from './books.service';

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styles: []
})
export class AppComponent {
  books: string[] = [];

  constructor(private booksService: BooksService) {}

  search(title: string) {
    this.books = this.booksService.getBooks(title);
  }

  list() {
    this.books = this.booksService.getAllBooks();
  }
}
```

Por último, inyectamos el servicio en un componente como argumento en el constructor de la clase



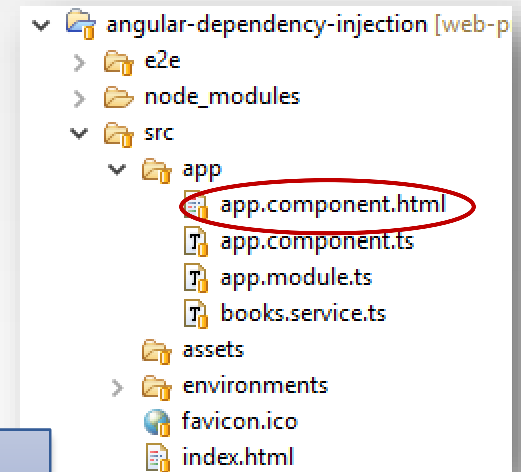
Angular

Inyección de dependencias

- Vamos a verlo a través de un ejemplo

app.component.html

```
<h1>Search Books</h1>

<input #title type="text">
<button (click)="search(title.value); title.value=''">Buscar</button>
<br><br>

<button (click)="list()">Listar todos</button>

<p *ngFor="let book of books">{{book}}</p>
```



El componente `AppComponent` tendrá una plantilla que nos permite listar todos los libros que ofrece el servicio, así como buscar por palabra contenida en el título

Los elementos de la plantilla se pueden asociar a una variable usando el símbolo #. Esa variable puede ser usada en el código embebido en la propia plantilla

Angular

Inyección de dependencias

```
> ng serve
```

Ejecutamos la aplicación con Angular CLI

Angular

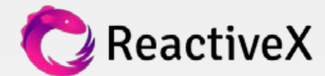
Servicios REST

- Angular dispone de su propio cliente de API REST
- Es un objeto de la clase `Http` (definido en el módulo `@angular/http`)
- Habrá que incluirlo en la lista de `imports` de `@NgModule` y después usarlo por inyección de dependencias en el componente que lo vaya a usar

app.module.ts

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

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

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

Angular



Servicios REST

- Una vez inyectado, podemos hacer uso del objeto `Http`
- Por defecto, las peticiones a servicios REST se manejan en Angular mediante lo que se conoce como **`Observables`** (forman parte de la librería RxJs usada en Angular)
- Es un método alternativo a las `Callbacks` y `Promises` para gestionar las operaciones asíncronas

GET

```
let url = "http://...";
this.http.get(url).subscribe(
   successResponse => callToSuccessFunction(successResponse);
   errorResponse => console.error(errorResponse)
);
```

DELETE

```
let url = "http://...";
this.http.delete(url).subscribe(
   successResponse => callToSuccessFunction(successResponse);
   errorResponse => console.error(errorResponse)
);
```

POST

```
let url = "http://...";
let data = { ... };
this.http.post(url, data).subscribe(
   successResponse => callToSuccessFunction(successResponse);
   errorResponse => console.error(errorResponse)
);
```

PUT

```
let url = "http://...";
let data = { ... };
this.http.put(url, data).subscribe(
   successResponse => callToSuccessFunction(successResponse);
   errorResponse => console.error(errorResponse)
);
```

Angular

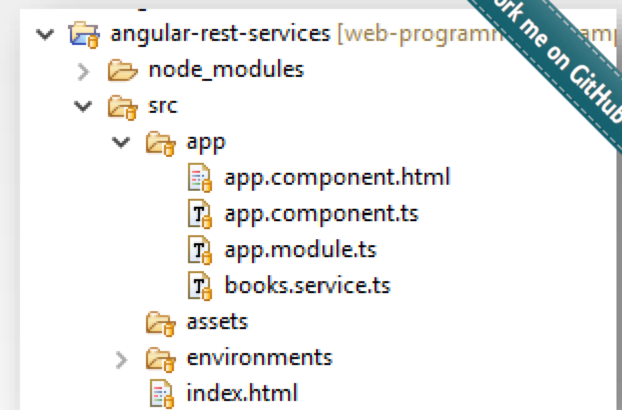
Servicios REST

- Vamos a ver un ejemplo sencillo

books.service.ts

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import 'rxjs/Rx';

@Injectable()
export class BooksService {

    constructor(private http: Http) { }

    getBooks(title: string) {
        let url = "https://www.googleapis.com/books/v1/volumes?q=intitle:" + title;
        return this.http.get(url).map(response => this.extractTitles(response))
    }

    private extractTitles(response: Response) {
        let out = response.json().items.map(book => book.volumeInfo.title);
        return out;
    }

}
```

Este servicio consume la API REST de Google para buscar libros



El método map del objeto `Observable` sirve para transformar el resultado del objeto observable aplicando una función a cada elemento de la respuesta

Angular

Servicios REST

- Vamos a ver un ejemplo sencillo

app.component.ts

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

import { BooksService } from './books.service';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html'
})
export class AppComponent {
    books: string[] = [];

    constructor(private http: Http, private service: BooksService) { }

    search(title: string) {
        this.books = [];
        this.service.getBooks(title).subscribe(
            success => this.books = success,
            error => console.error(error)
        );
    }
}
```

No se considera una buena práctica consumir un servicio REST directamente desde un componente. En su lugar, se recomienda implementar un servicio inyectado en el componente

El servicio `BooksService` implementa una función asíncrona (llamada al servicio REST), con lo que tiene que devolver un objeto `Observable` que se gestiona también con `subscribe`

Fork me on GitHub

Angular

Servicios REST

- Vamos a ver un ejemplo sencillo

app.component.html

```
<h1>Search Books (from www.googleapis.com/books)</h1>

<input #title type="text">

<button (click)="seek(title.value); title.value=''">Search
(logic in a service)</button>

<p *ngFor="let book of books">{{book}}</p>
```

Search Books (from www.googleapis.com/books)

java | Search (logic in the controller)

Java a tope: Java2D. Cómo tratar con Java figuras, imágenes y texto en dos dimensiones

JAVA: OCP JAVA SE 6 PROGRAMMER, EXAM 310-065, PRACTICE EXAMS

Introduccion Al Desarrollo de Programas Con Java

Curso de Programación en Java-J2EE

Java 算法/Algorithms in Java/影印版(第 3 版.第 1 卷)/算法经典丛书

Java para estudiantes

Programming Language Processors in Java

Compiling with C# and Java

El factor de java en la industria azucarera argentina

La biblia de Java 2

Angular

Routing

- Las aplicaciones SPA que hemos visto hasta ahora cambian el estado de las vistas sin un cambio en la URL de la aplicación
- No obstante, en ciertas ocasiones vamos a querer que nuestras aplicaciones SPA se identifiquen con diferentes URLs
- Para ello, Angular proporciona un componente especial (***router***) que permite la navegación usando diferentes URLs

Angular

Fork me on GitHub

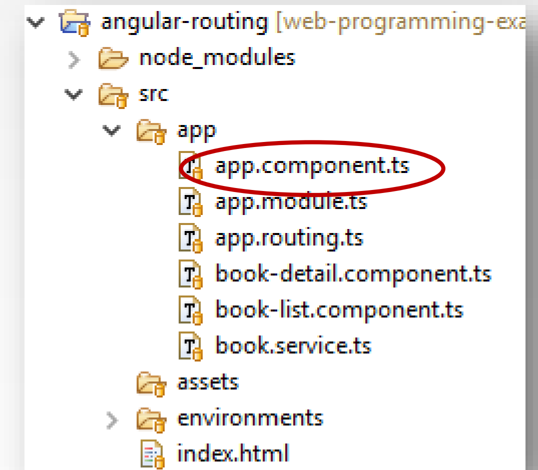
Routing

- Como siempre, vamos a verlo a través de un ejemplo:

app.component.ts

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

@Component({
    selector: 'app-root',
    template: `
<h1>Library</h1>
<router-outlet></router-outlet>
`
})
export class AppComponent { }
```

En este ejemplo, vamos embeber las plantillas dentro del propio componente usando la palabra clave `template` en lugar de `templateUrl`



Mediante el tag `router-outlet` establecemos la parte de la plantilla cuyo contenido es variable (dependerá de la URL)

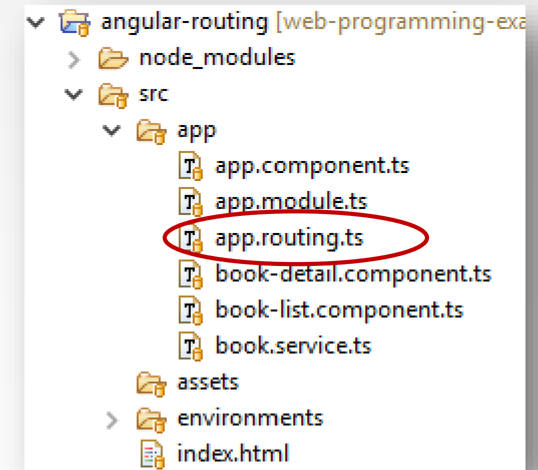
Angular

Routing

app.routing.ts

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

import { BookListComponent } from './book-list.component';
import { BookDetailComponent } from './book-detail.component';

const appRoutes = [
    { path: '', redirectTo: 'books', pathMatch: 'full' },
    { path: 'books', component: BookListComponent },
    { path: 'book/:id', component: BookDetailComponent, }
]

export const routing = RouterModule.forRoot(appRoutes);
```

En este fichero se asocian URLs con diferentes componentes. Además, se establece la ruta por defecto (`redirectTo`)



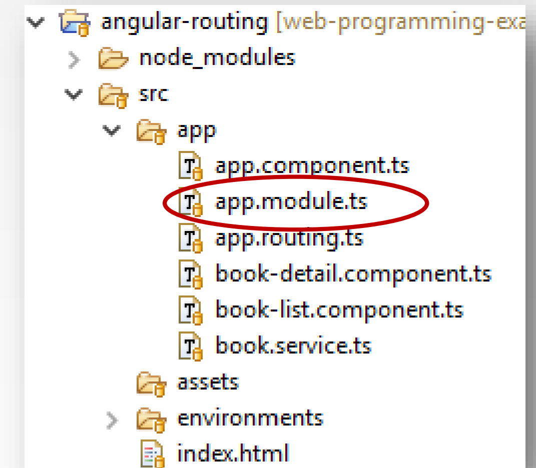
Angular

Routing

Fork me on GitHub

app.module.ts

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

import { AppComponent } from './app.component';
import { BookListComponent } from './book-list.component';
import { BookDetailComponent } from './book-detail.component';
import { BookService } from './book.service';
import { routing }  from './app.routing';

@NgModule({
    declarations: [AppComponent, BookDetailComponent, BookListComponent],
    imports: [BrowserModule, FormsModule, HttpModule, routing],
    bootstrap: [AppComponent],
    providers: [BookService]
})
export class AppModule { }
```



Las rutas se considera un módulo que hay que importar en el módulo principal

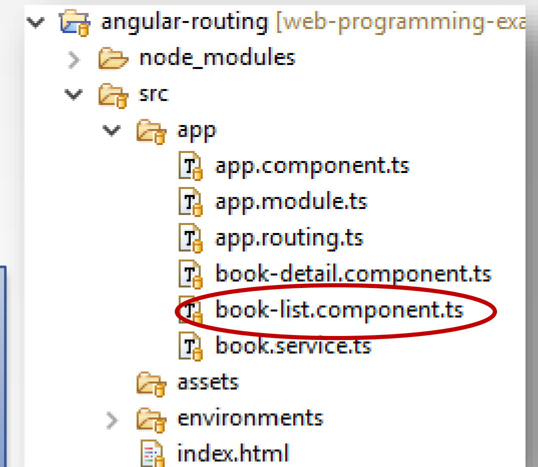
Angular

Routing

book-list.component.ts

```
import { Component } from '@angular/core';
import { Book, BookService } from './book.service';

@Component({
    template: `
<h2>Books</h2>
<ul>
   <li *ngFor="let book of books">
      <a [routerLink]="['/book', book.id]">{{book.id}} - {{book.title}}</a>
   </li>
</ul>
`
})
export class BookListComponent {

    books: Book[];

    constructor(service: BookService) {
        this.books = service.getBooks();
    }
}
```

En lugar de `href`, los links usan `[routerLink]`. La URL se puede indicar como un cadena (completa) o como un array de cadenas (si hay parámetros)

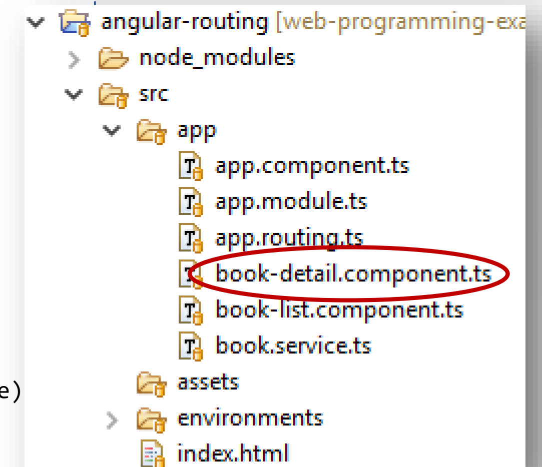


Angular

Routing

book-detail.component.ts

```
import { Component } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
import { Book, BookService } from './book.service';

@Component({
    template: `
<h2>{{book.title}}</h2>
<div><label>Id: </label>{{book.id}}</div>
<div><label>Author(s): </label>{{book.description}}</div>
<p><button (click)="back()">Back</button></p>`
})
export class BookDetailComponent {

    book: Book;

    constructor(private router: Router, activatedRoute: ActivatedRoute, service: BookService)
        let id = activatedRoute.snapshot.params['id'];
        this.book = service.getBook(id);
    }

    back() {
        this.router.navigate(['/books']);
    }
}
```



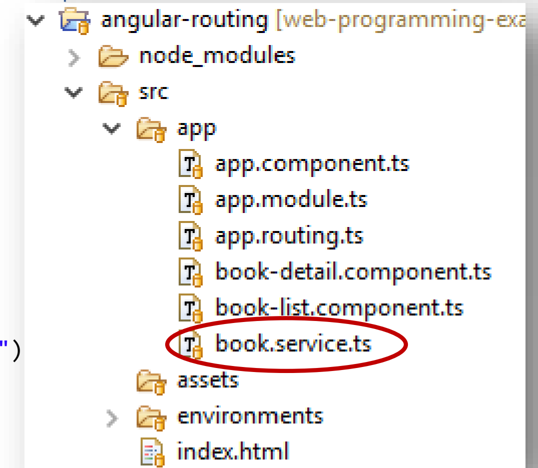
Para acceder a los parámetros desde el componente usamos el servicio `ActivatedRoute`

Angular

Routing

book.service.ts

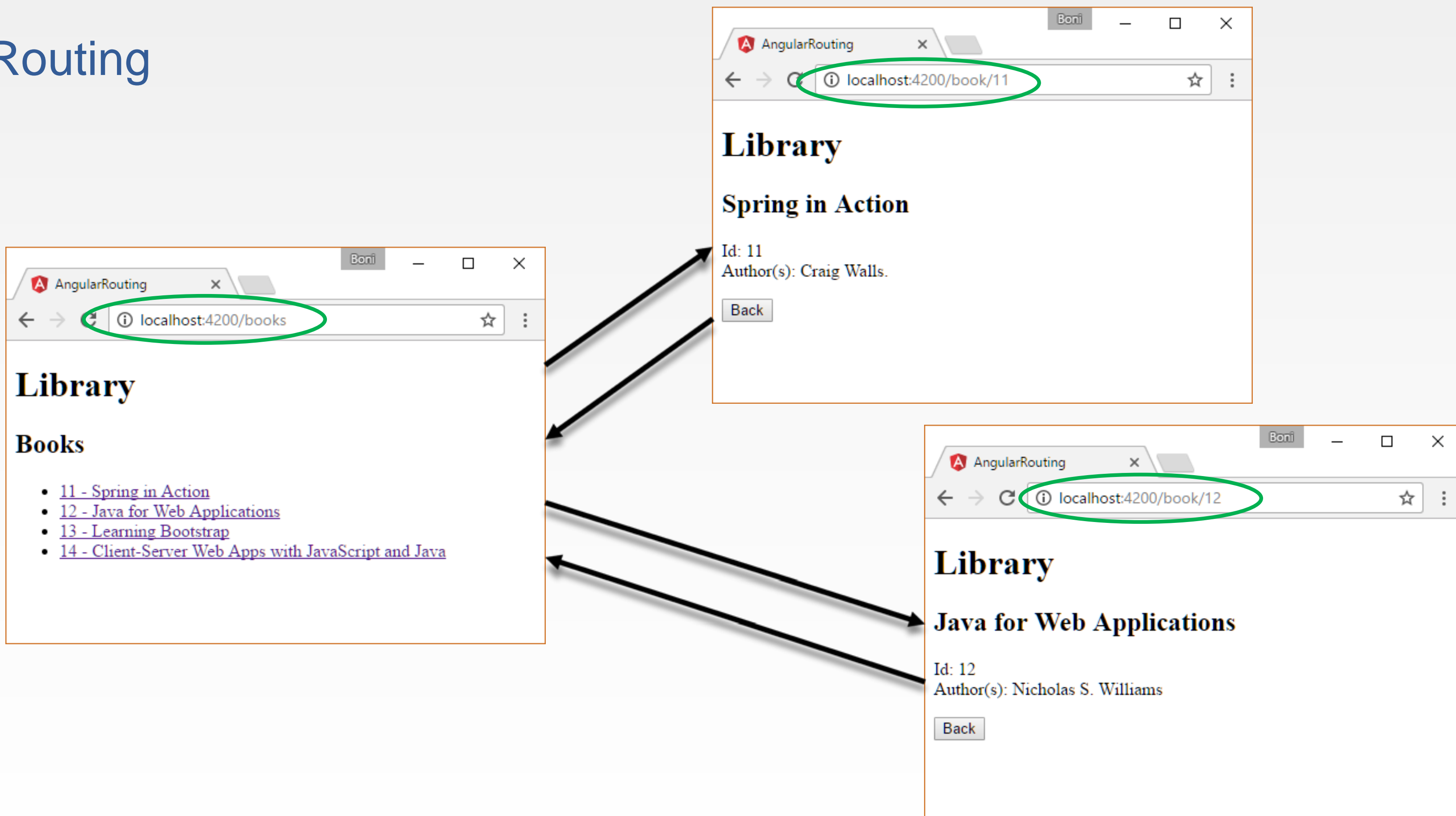
```
import { Injectable } from '@angular/core';

export class Book {
    constructor(public id: number, public title: string, public description: string) { }
}

@Injectable()
export class BookService {

    private books = [
        new Book(11, "Spring in Action", "Craig Walls."),
        new Book(12, "Java for Web Applications", "Nicholas S. Williams"),
        new Book(13, "Learning Bootstrap", "Aravind Shenoy, Ulrich Sossou"),
        new Book(14, "Client-Server Web Apps with JavaScript and Java",  "Casimir Saternos.")
    ];

    getBooks() {
        return this.books;
    }

    getBook(id: number) {
        return this.books.find(book => book.id == id);
    }
}
```



Esta clase almacena una lista de libros y exporta una función para buscar por identificador (numérico)

Angular

Routing



Angular

Otras *features* de Angular

- Composición de componentes (comunicación padre-hijo y eventos hijo-padre)
- Creación de directivas propias
- Programación reactiva con RxJS
- Transformación de datos en una plantilla (*pipes*)
- Manejo y validación de formularios
- Pruebas (unitarias, e2e)
- Despliegue de aplicaciones
- Optimización de código: *Ahead-of-Time* (AOT) *Compilation*

https://angular.io/docs/ts/latest/cookbook/

Índice

1. Introducción
2. Node.js
3. TypeScript
4. Angular
5. Librerías de componentes
   - Bootstrap
   - Material
   - PrimeNG

Librerías de componentes

Bootstrap

- Existen diferentes librerías que facilitan el diseño de páginas web
- Para usar Bootstrap en una aplicación Angular en primer lugar hay que instalar la dependencia (con NPM)

```
> npm install --save bootstrap@4.0.0-beta
```

- Después añadimos la ruta de la hoja de estilos en `styles.css`:

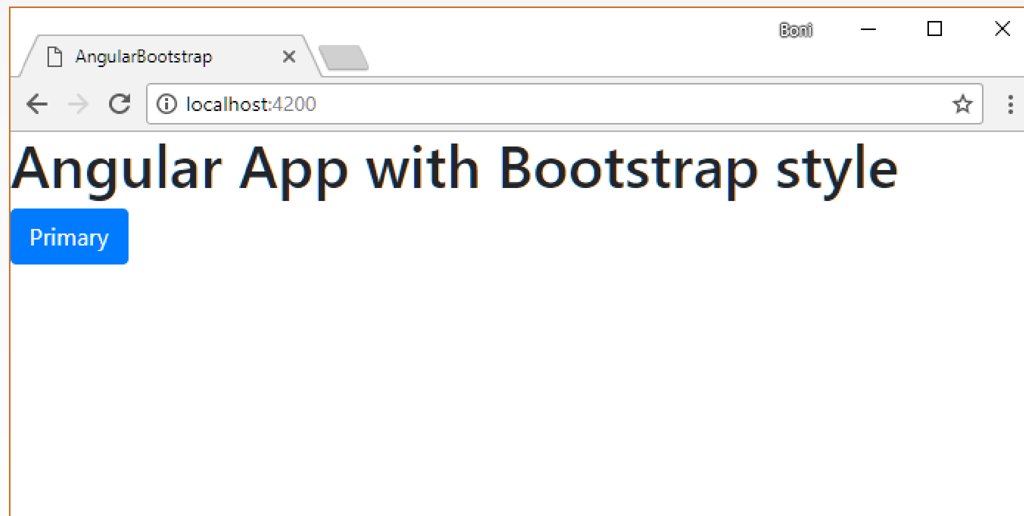
```
@import "~bootstrap/dist/css/bootstrap.min.css";
```

Librerías de componentes

Bootstrap

Fork me on GitHub

```
<h1>Angular App with Bootstrap style</h1>

<button type="button" class="btn btn-primary">Primary</button>
```



Librerías de componentes

Material

- Librería de componentes con el estilo *Material* de Google
- Para usar Angular Material hay que instalar la dependencia (con NPM)

```
> npm install --save @angular/material @angular/cdk @angular/animations
```

- Después añadimos la ruta de la hoja de estilos en `styles.css`:

```
@import "~@angular/material/prebuilt-themes/purple-green.css";
```

Esta hoja de estilos es una de las que hay disponibles (podemos cambiar entre esta y otras)

MATERIAL

Librerías de componentes

Material

- Además hay que registrar el módulo `BrowserAnimationsModule` así como los módulos que vayamos a usar en nuestra aplicación:

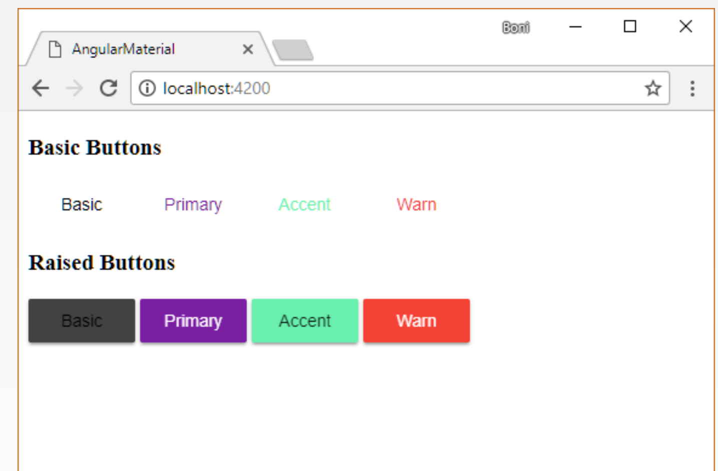
```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {BrowserAnimationsModule} from '@angular/platform-browser/animations';
import {MatButtonModule, MatCheckboxModule} from '@angular/material';

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


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

MATERIAL

Librerías de componentes

Material

```
<h3>Basic Buttons</h3>
<div class="button-row">
    <button mat-button>Basic</button>
    <button mat-button color="primary">Primary</button>
    <button mat-button color="accent">Accent</button>
    <button mat-button color="warn">Warn</button>
    <button mat-button disabled>Disabled</button>
</div>

<h3>Raised Buttons</h3>
<div class="button-row">
    <button mat-raised-button>Basic</button>
    <button mat-raised-button color="primary">Primary</button>
    <button mat-raised-button color="accent">Accent</button>
    <button mat-raised-button color="warn">Warn</button>
    <button mat-raised-button disabled>Disabled</button>
</div>
```



https://material.angular.io/

MATERIAL

Librerías de componentes

PrimeNG

- Para usar PrimeNG hay que instalar la dependencia (con NPM)

```
> npm install primeng --save
```

- Después añadimos los estilos en `styles.css`:

```
@import "~primeng/resources/themes/omega/theme.css";
@import "~primeng/resources/primeng.min.css";
```

Librerías de componentes

PrimeNG

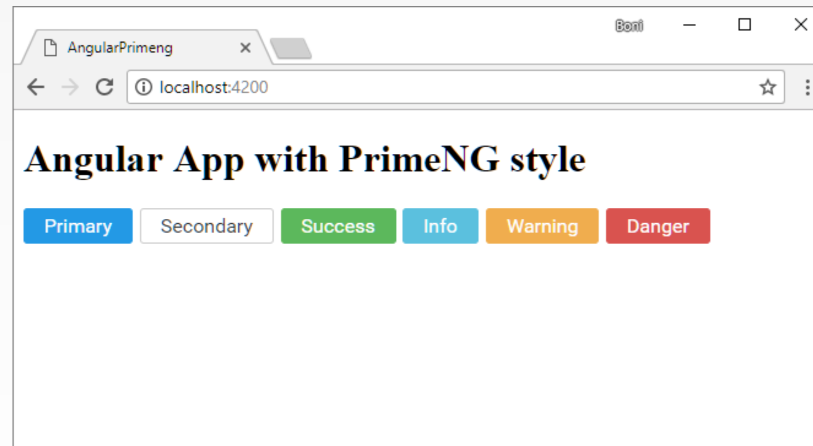
- Además hay que registrar el módulo que necesitemos en el módulo principal de la aplicación:

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

import {ButtonModule} from 'primeng/button';
import {AppComponent} from './app.component';


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

Librerías de componentes

PrimeNG

```
<h1>Angular App with PrimeNG style</h1>
<button pButton type="button" label="Primary"></button>
<button pButton type="button" label="Secondary" class="ui-button-secondary"></button>
<button pButton type="button" label="Success" class="ui-button-success"></button>
<button pButton type="button" label="Info" class="ui-button-info"></button>
<button pButton type="button" label="Warning" class="ui-button-warning"></button>
<button pButton type="button" label="Danger" class="ui-button-danger"></button>
```



https://www.primefaces.org/primeng/