CURSO DE PROGRAMACIÓN JAVA FULLSTACK

Angular

1. Introducción. Componentes.

Arturo Bernal Mayordomo

Edición: Noviembre 2019

Contenido

1. - Introducción

Angular es un framework para el desarrollo de aplicaciones web en la parte del cliente. Está basado en el popular AngularJS (versiones 1.x), pero ha cambiado mucho con respecto a su antecesor. Dos mejoras significativas son: un rendimiento muy superior, y una API simplificada con menos conceptos que aprender.

Angular está desarrollado en TypeScript, y es el lenguaje recomendado para construir aplicaciones con este framework. Este lenguaje es básicamente JavaScript con algunas características adicionales y el tipado de variables y funciones. Esto mejora mucho la depuración de aplicaciones en tiempo de desarrollo, el autocompletado por parte del editor (intellisense), etc. Al pasar por un proceso de compilación a JavaScript, el resultado será código compatible con todos los navegadores actuales.

Algunas de las principales características de Angular son:

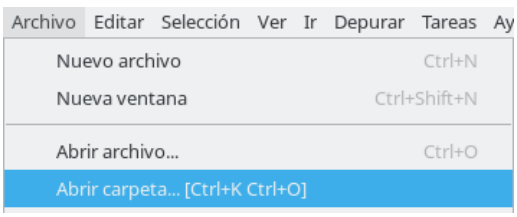
- Introduce expresividad en el código HTML a través de la interpolación de variables, data-binding, directivas, etc.
- Tiene un diseño modular. Sólo se importan las características que necesitamos en la aplicación (mejora de tamaño y rendimiento). Además permite separar nuestra aplicación en módulos de forma que el navegador vaya cargando en cada momento lo que necesita (lazy loading) → Mejora de tiempos de carga.
- Es fácil crear componentes reutilizables para la aplicación actual u otras.
- La integración con un servidor de backend basado en servicios web es muy sencilla.
- Permite ejecutar Angular en el lado del servidor para generar contenido estático que puedan indexar los motores de búsqueda (incapaces de ejecutar ellos Angular en el cliente). Esta característica se llama Angular Universal.
- Potentes herramientas de desarrollo y depuración: TypeScript, Augury (plugin de Chrome), framewors de pruebas como Karma o Jasmine, etc.
- Integración con frameworks de diseño como Bootstrap, Angular Material, Ionic…
- En Angular, creamos aplicaciones de una página (SPA), donde la página principal se carga entera sólo una vez → Mejor rendimiento.
- Debido a su naturaleza de framework completo y modular, es la solución más efectiva para desarrollar grandes aplicaciones.

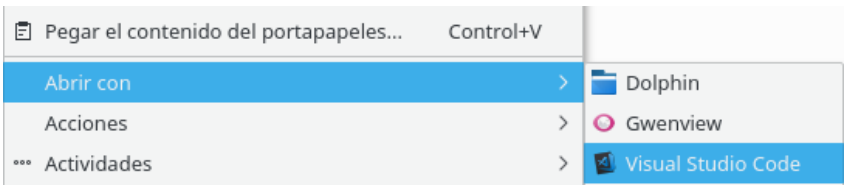
En resumen, vamos a aprender un framework completo y muy popular, desarrollado y usado por Google, y con unas mejoras significativas con respecto a la primera versión orientadas a facilitar el desarrollo de grandes aplicaciones.

2. - Creación de un proyecto Angular

Aunque varios editores (Atom, Webstorm, etc.) se integran bien con Angular por medio de extensiones, en este curso recomiendo utilizar Visual Studio Code. Es un editor/IDE de código abierto que se integra muy bien con este framework, y es el que utilizaré durante el curso para ejemplos, resolver dudas, etc. Cabe destacar que este editor es una aplicación de escritorio completamente desarrollada mediante tecnologías web (HTML, JavaScript, …) gracias a Electron.

En VS Code un proyecto es un directorio o carpeta del sistema operativo. No existe el concepto de creación de un proyecto como en otros IDE. Las carpetas se pueden abrir desde el propio editor, arrastrándola, mediante el comando **"code ."**, o mediante clic derecho en la carpeta → Abrir con Visual Studio Code.





El último icono de la barra de la izquierda del editor, representa las extensiones que se pueden instalar para ampliar las funcionalidades (integración con lenguajes, frameworks, FTP, GIT, …). Para este curso os recomiendo buscar e instalar el paquete de extensiones **Angular Essentials**.

2.1 *Creando un proyecto con Angular CLI*

Para gestionar nuestros proyectos de Angular, vamos a usar la herramienta **Angular CLI**. Esta herramienta de línea de comandos permite generar un proyecto, componentes para el mismo, compilarlo y ejecutarlo con diferentes configuraciones (desarrollo, producción, …), lanzar baterías de pruebas automatizadas, etc.

Más información en la guía de inicio rápido de Angular.

Para instalar (o actualizar) Angular CLI, primero debemos tener instalada la herramienta Node Package Manager (NPM), que viene incluida con NodeJS (Instalad siempre la versión LTS). Una vez cumplido esto, ejecutamos (como administrador, en distribuciones Linux con **sudo** delante):

```
npm i @angular/cli -g
```

Podemos comprobar que todo ha ido bien con el siguiente comando:

```
arturo@arturo-desktop:~$ ng --version

     _                      _                 ____ _     ___
    / \   _ __   __ _ _   _| | __ _ _ __     / ___| |   |_ _|
   / △ \ | '_ \ / _` | | | | |/ _` | '__|   | |   | |    | |
  / ___ \| | | | (_| | |_| | | (_| | |      | |___| |___ | |
 /_/   \_\_| |_|\__, |\__,_|_|\__,_|_|       \____|_____|___|
                |___/


Angular CLI: 8.3.2
Node: 10.16.3
OS: linux x64
```

Una vez instalada la herramienta, creamos un proyecto Angular:

```
ng new NOMBRE-PROYECTO
```

Esto nos creará un directorio con el nombre del proyecto indicado. Para comprobar que funciona correctamente, entramos en el directorio y ejecutamos el comando **ng serve**. Esto lanzará un servidor web con nuestra aplicación que podemos ver en http://localhost:4200.

Para más opciones del comando **ng new**, como por ejemplo, no generar archivos para test (--skip-tests), consulta la documentación de Angular CLI.

2.2 *Como funciona Angular CLI*

Angular CLI es una herramienta para crear y gestionar proyectos Angular. Permite automatizar tareas como la compilación, pruebas, generación de código, ejecución, etc. Lo que permite centrarnos en programar. Para ello se sirve de herramientas como Webpack, Jasmine, Karma, etc.

Generando componentes, directivas, pipes, servicios, etc.

A lo largo de este curso explicaremos las diferentes partes que componen una aplicación Angular. Podemos generar un esqueleto para la mayoría mediante Angular CLI. Simplemente desde el directorio (o un subdirectorio) del proyecto Angular, ejecutamos el comando **ng generate** (**ng g**). Más información.

Component	ng g component my-new-component
Directive	ng g directive my-new-directive
Pipe	ng g pipe my-new-pipe

Service	ng g service my-new-service
Class	ng g class my-new-class
Interface	ng g interface my-new-interface
Enum	ng g enum my-new-enum
Module	ng g module my-new-module
Guard	ng g guard my-new-guard

Instalando librerías y actualizando versiones

El comando **ng add** nos instalará una librería externa en nuestro proyecto similar a usar **npm install**, con la diferencia de que si la librería necesita que configuremos algo en nuestro proyecto para integrarla correctamente, ng add lo hará por nosotros (siempre que la librería tenga algún tipo de integración con Angular), mientras que con npm install habrá que hacerlo de forma manual. Ejemplo:

ng add @angular/material → Instala y añade el soporte para Angular Material a nuestro proyecto.

El comando **ng update** nos mostrará si hay versiones nuevas de Angular y nos permite actualizar el proyecto actual a una versión más reciente. Con la opción **--all** intentará actualizar todos los módulos de Angular en el proyecto.

```
We analyzed your package.json, there are some packages to update:

  Name                                Version                  Command to update
 --------------------------------------------------------------------------------
  @angular/cli                        7.1.3 -> 8.3.2           ng update @angular/cli
  @angular/core                       7.1.3 -> 8.2.4           ng update @angular/core
  @angular/core                       7.1.3 -> 7.2.15          ng update @angular/core
  rxjs                                6.3.3 -> 6.5.2           ng update rxjs


There might be additional packages that are outdated.
Run "ng update --all" to try to update all at the same time.
```

Compilando el proyecto y generando el bundle

Subir todo el proyecto tal cual lo tenemos a un servidor web es innecesario y lento (comprueba el tamaño del directorio del proyecto). Esto ocurre porque hay muchas dependencias y archivos que instala automáticamente NPM, la mayoría de las cuales son herramientas para el desarrollo, testing, y compilación que no necesitamos para una aplicación en producción, solo para desarrollar.

Cuando compilamos la aplicación, Angular CLI usará WebPack para empaquetar todos nuestros archivos CSS, JavaScript, etc. en unos pocos archivos (minificados), resolviendo las dependencias entre las diferentes partes del código y dejando sólo lo estrictamente necesario para ejecutar la aplicación. Estos archivos, listos para subir al servidor web, se encuentran el directorio **dist/**.

El comando para compilar y empaquetar nuestra aplicación es **ng build**. Por defecto, construye a aplicación en modo desarrollo (pesa más y contiene información de depuración). Para compilar en modo producción se debe ejecutar **ng build --prod**. Esta opción generará un código más pequeño y rápido, sin información de depuración.

Más información: https://github.com/angular/angular-cli

2.3 *Creando un nuevo proyecto*

Para nuestro ejemplo vamos a generar un proyecto llamado **angular-products**.

```
arturo@arturo-desktop:~/Documentos/angular$ ng new angular-products
? Would you like to add Angular routing? No
```

La primera pregunta nos pide si queremos utilizar el router de Angular. En el caso de tener una aplicación con más de 1 página la respuesta sería **sí**. Sin embargo, por ahora vamos a decir que no y añadiremos esa funcionalidad manualmente más adelante para entender mejor como funciona.

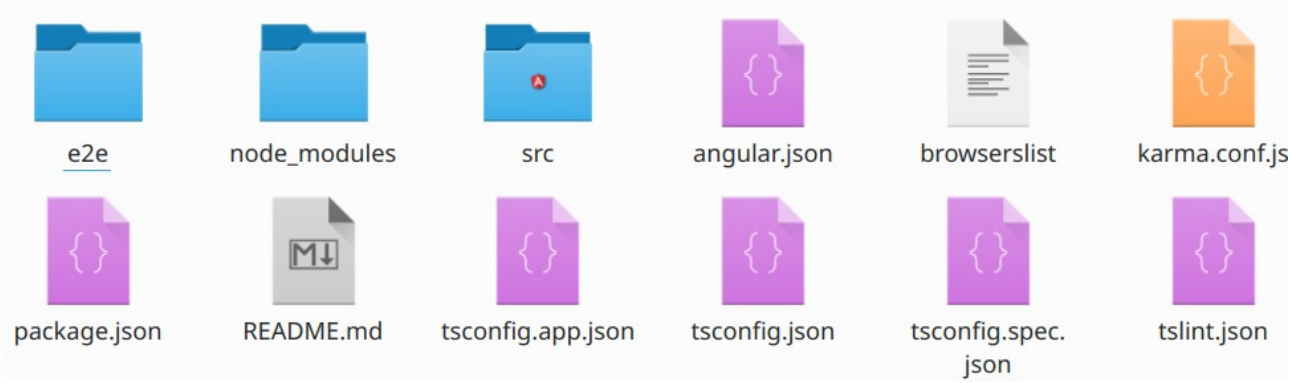
```
? Which stylesheet format would you like to use? (Use arrow keys)
› CSS
  SCSS   [ https://sass-lang.com/documentation/syntax#scss                ]
  Sass   [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less   [ http://lesscss.org                                             ]
  Stylus [ http://stylus-lang.com                                         ]
```

Las siguiente opción tiene que ver con las hojas de estilo que usaremos en la aplicación. Si queremos usar simplemente CSS, o si preferimos una herramienta más avanzada como Sass. En cualquier caso no hay que preocuparse ya que Angular CLI (junto con Webpack) se encargan de compilar automáticamente el código a CSS.

Estructura del proyecto

Vamos a analizar un poco qué directorios y archivos se crean cuando generamos un nuevo proyecto Angular.

En el directorio principal del proyecto veremos archivos de configuración para:



- **NPM** (package.json). Entre otras cosas encontramos las dependencias del proyecto
- **Angular CLI** (angular.json). Configuración global del proyecto Angular.
- El compilador de **TypeScript** (tsconfig.json)
- **TSLint** (tslint.json): una herramienta para gestionar las reglas de estilo de código y sobre qué tipo de errores (y warnings) debe avisar el editor en tiempo de desarrollo. Estos archivos son todos configurables a gusto del desarrollador o equipo de desarrollo.
- **Karma** (karma.conf.js): Una herramienta para ejecutar tests del proyecto en el navegador y que usa frameworks como **Jasmine** (pruebas unitarias), o **Protractor** (e2e/protractor.conf.js): para ejecutar pruebas funcionales.

También encontramos estos subdirectorios:

- **e2e**: Aquí se definen los tests de integración (**Protractor**).
- **node_modules**: Dependencias de la aplicación instaladas con NPM.
- **src**: Donde está nuestra aplicación (también tests unitarios: **Jasmine)**

2.4 *Carga inicial de la aplicación (Bootstrap)*

Una aplicación Angular debe tener un módulo principal al menos (**@NgModule**), y se pueden crear tantos submódulos como se quiera (lo veremos en un futuro). Dividir la aplicación es muy útil de cara tanto a separar código y funcionalidad para el desarrollo, como para que nuestra aplicación sólo cargue en memoria los módulos que necesita en cada momento (lazy loading).

El módulo principal de la aplicación se crea en **src/app/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 { }
```

Analicemos primero qué significan los imports iniciales:

- **BrowserModule**: Este módulo del framework contiene lo necesario para trabajar con la funcionalidad del navegador (manipulación del DOM, eventos, animaciones, etc.). Desde la versión 2, Angular no está fuertemente integrado con el navegador (es un módulo más), por ello puede funcionar en otros entornos, como por ejemplo junto a NativeScript para el desarrollo de aplicaciones móviles con componentes nativos, etc.
- **NgModule**: Esto importa el decorador @NgModule. Los decoradores son funciones (equivalentes a las anotaciones de Java, por ejemplo), que especifican metadatos que describen una clase o un método. Estos metadatos son interpretados por Angular en este caso, para saber qué tipo de componente del framework (componente, servicio, módulo, filtro, etc.) representa la clase implementada. En este caso hablamos de un **módulo** (además, es el módulo principal de la aplicación). Existen otros decoradores como **@Component**, **@Pipe**, **@Directive**, **@Injectable**, ...
- **AppComponent**: Es el componente principal de la aplicación. Toda la aplicación se ejecuta dentro de este componente como pronto veremos.

Vamos a ver qué tipo de metadatos se definen en un módulo:

- **declarations** → **Componentes**, **directivas** y **filtros** (pipes) que se utilizarán en este módulo. Como veremos, cuando creamos un componente (o directiva, o filtro) debemos añadirlo a este array para poder usarlo.
- **imports** → En este array añadimos módulos externos (de Angular o creados por nosotros). Los elementos exportados por esos módulos (componentes, etc.) serán accesibles por el módulo actual.
- **exports** → No está presente en el ejemplo actual. En este array añadiremos los componentes, directivas, etc. del módulo actual (declarations) que queramos exportar. Es decir, cuando otro módulo importe el actual, qué cosas podrá usar.
- **providers** → En este array añadimos servicios de Angular. Esta parte la veremos más adelante.
- **bootstrap** → Este array sólo suele encontrarse en el módulo de la aplicación (AppModule). Se define el componente principal o inicial de la aplicación. Lo primero que se carga y se muestra. El resto de componentes se situarán dentro del componente principal.

¿Cómo sabe Angular, en el caso de haber varios módulos, que el módulo principal es AppModule, y es el que debe cargar al principio?. Esto se puede ver en el archivo **src/main.ts**:

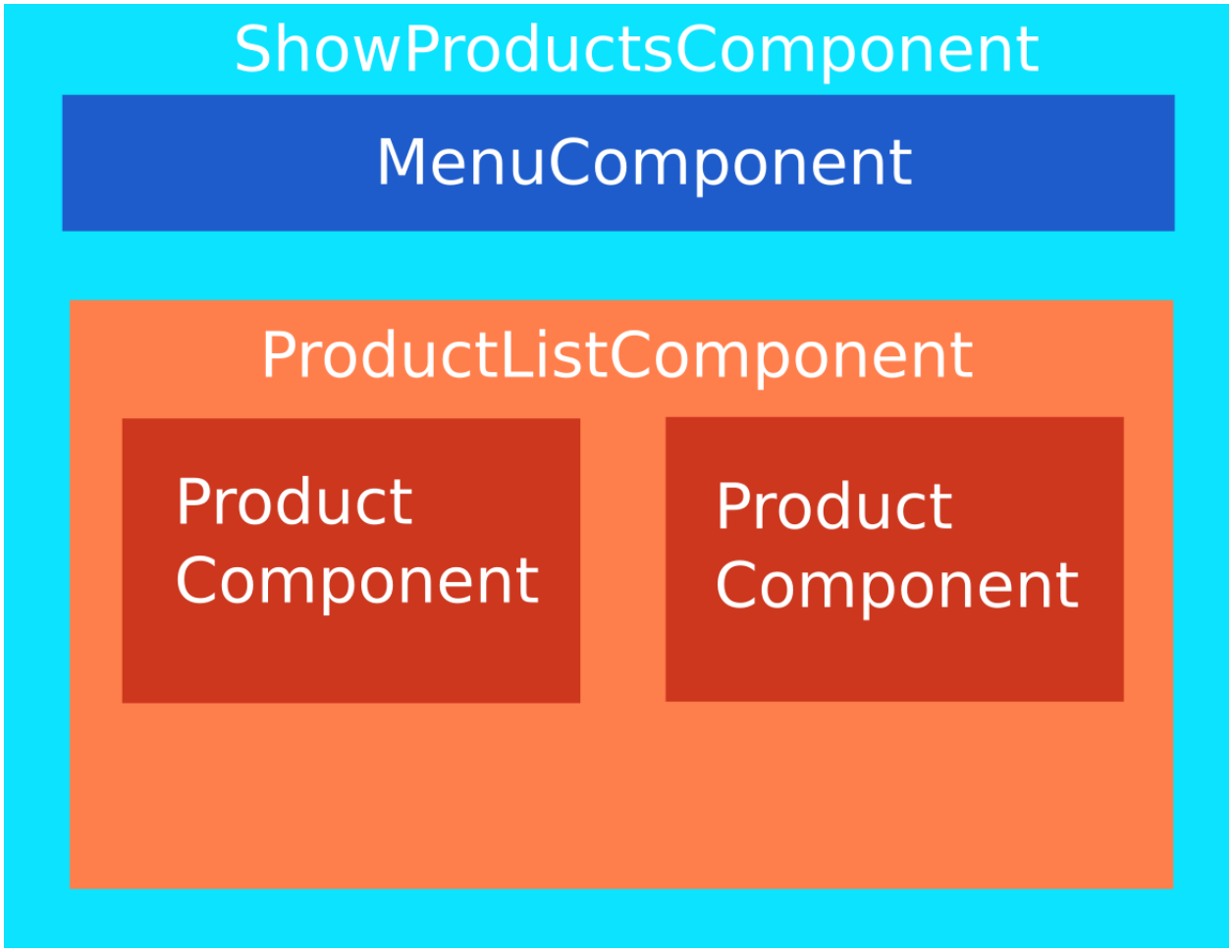
```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

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

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

platformBrowserDynamic().bootstrapModule(AppModule)
```

```
  .catch(err => console.log(err));
```

En breve veremos que **index.html** es el único documento HTML que se carga en la aplicación. A partir de ahí, Angular toma el control y mediante su router, podremos navegar por las diferentes secciones de nuestra aplicación sin movernos nunca del documento principal (index.html). Esto es lo que se llama una Aplicación de Página Única (Single-page Application o SPA).

3. - **Componentes**

En Angular, si comparamos con otros frameworks Modelo-Vista-Controlador, un componente sería un controlador de una vista asociada (HTML). Los componentes son entidades independientes y reutilizables en diferentes partes de una aplicación (u otras aplicaciones). También se pueden anidar (relación padre-hijo). Los



componentes representan las diferentes partes en las que dividimos la aplicación.

Cuando creamos un proyecto, se añade un componente principal de la aplicación (que contendrá al resto) en **src/app/app.component.ts**. Vamos a analizarlo:

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

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

Como podemos observar, un componente es una clase con el decorador **@Component** y su metadatos que incluyen lo siguiente:

- **selector** → Nombre de la etiqueta que se utilizará en la vista (HTML) para cargar el componente dentro. Como HTML no es sensible a mayúsculas se utiliza una sintaxis del tipo kebab-case. Mira en **src/index.html** donde verás la etiqueta **<app-root>** ya creada (ahí es donde se carga la aplicación).
  - Por defecto, todos los componentes se crean con un selector que tiene el prefijo **app**. Este prefijo sería mejor que tuviera las iniciales de nuestro proyecto (o fuese más descriptivo). Esto se puede configurar al crear el proyecto → **ng new mi-proyecto --prefix mp**. Posteriormente también se puede en el archivo **angular.json** (busca el atributo "**prefix**").
- **templateUrl** → Este archivo representa la vista asociada al componente. El HTML que se cargará dentro del selector cuando esté todo cargado.
- **styleUrls** → Angular permite asignar una o varias hojas de estilo CSS (o SASS o LESS si se ha creado así el proyecto) a un componente. Estos estilos se aplicarán **solamente** al componente asociado.
  - Los estilos generales para toda la aplicación se definen en **src/styles.css**.

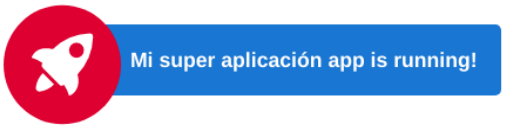
La clase AppComponent tiene una propiedad declarada llamada **title**. Esta propiedad es pública (por defecto) y de tipo string (en una asignación TypeScript asigna el tipo del valor asignado a la variable, no hace falta especificarlo).

Si observas la plantilla (**src/app/app.component.html**), verás un documento HTML bastante largo de ejemplo. Si buscas encontrarás algo como esto

```
<span>{{ title }} app is running!</span>
```

Esto se llama **interpolación** (lo veremos en la siguiente sección). Esto significa que en la plantilla {{title}} está vinculada a la propiedad title de la clase **AppComponent**. Cuando el componente se cargue, el valor se sustituirá y aparecerá el texto "**Welcome to angular-productos**". Cambia el texto de la propiedad "title" y verás el efecto (no hace falta que pares la aplicación y vuelvas a lanzarla).

```
export class AppComponent {
  title = 'Mi super aplicación';
}
```



En lugar de usar un archivo externo para la plantilla, se podría definir dentro de una cadena utilizando **template** en lugar de **templateUrl** en el decorador.

```
@Component({
  selector: 'app-root',
  template: `
    <h1>
      {{title}}
    </h1>`,
  styleUrls: ['./app.component.css']
})
```

Para quien no esté muy familiarizado con las **template strings** de la versión ES2015 de JavaScript (cadenas entre comillas invertidas ``), aquí tiene más información.

4. - Plantillas

Como hemos dicho antes, una plantilla es la vista asociada a un componente. La clase del componente se encarga de controlar la plantilla asociada.

Antes de nada, para dotar fácilmente de un estilo básico a los componentes que vayamos creando, vamos a instalar Bootstrap. En nuestro caso, sólo usaremos la parte CSS de dicha herramienta, aunque como se verá en un futuro, se integra bastante bien con Angular (https://ng-bootstrap.github.io/#/home). Vamos a instalarlo con NPM (desde el directorio principal del proyecto):

```
npm i bootstrap
```

Ahora simplemente tenemos que incluir el archivo CSS de Bootstrap en nuestro proyecto. En lugar de incluir directamente en index.html, lo incluiremos en el archivo **src/styles.css**. Cuando ejecutemos **ng serve** o **ng build**, Webpack se encargará de empaquetar todo en un único CSS automáticamente.

```
/* You can add global styles to this file, and also import other style files */
/* Incluimos Bootstrap */
@import "../node_modules/bootstrap/dist/css/bootstrap.css"
```

Otra forma de importar un archivo CSS, es en el archivo de configuración **angular.json**, dentro del array "**styles**":

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.css",
  "src/styles.css"
],
```

4.1 *Selector de un componente*

Como vimos anteriormente, los componentes tienen un atributo llamado **selector**. Cada vez que situemos esa etiqueta en una plantilla HTML, se instanciará un objeto del componente y se cargará su plantilla asociada dentro de dicho selector.

El componente AppComponent se incluye dentro de index.html mediante su selector (la etiqueta **<app-root>**). Vamos a crear un componente llamado **<product-list>** que gestionará una lista de productos y lo incluiremos dentro de AppComponent.

4.2 *Creando un nuevo componente*

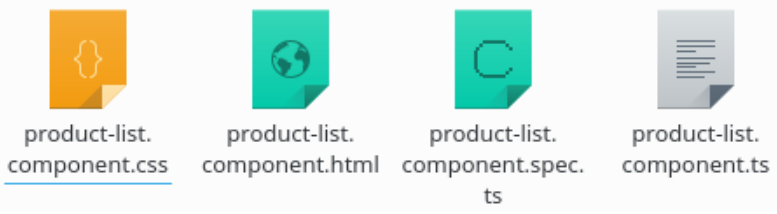
Antes de nada, en el archivo **angular.json**, vamos a establecer un prefijo vacío para los selectores (en lugar de app). Si vamos a crear una librería de componentes, se recomienda usar un prefijo que los identifique. En el caso de una aplicación con componentes que solo se van a usar en la aplicación actual, podemos quitarlo.

```
"prefix": "",
```

También debemos quitarlo del archivo tslint.json (carpeta raíz del proyecto), ya que si no se quejará de que los selectores de los componentes deben tener el prefijo prestablecido. Lo más fácil es definir el prefijo al crear el proyecto y ahorrarnos esto.

```
"directive-selector": [
  true,
  "attribute",
  "",
  "camelCase"
],
"component-selector": [
  true,
  "element",
  "",
  "kebab-case"
]
```

Podemos crear componentes de forma manual o usando Angular CLI. Si ejecutamos **ng g component product-list**, generará un directorio llamado como el componente (dentro de **src/app**) con los siguientes archivos:



También actualizará automáticamente el archivo **app.module.ts** para incluir el componente en la aplicación:

```
...
import { AppComponent } from './app.component';
import { ProductListComponent } from './product-list/product-list.component';

@NgModule({
```

```
  declarations: [
    AppComponent,
    ProductListComponent
  ],
  ...
})
export class AppModule { }
```

Los archivos con extensión **spec.ts** (product-list.component.spec.ts) es donde se definen los tests unitarios de los componentes (karma, Jasmine).

Este es el contenido del archivo **product-list.component.ts**:

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

@Component({
  selector: 'product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

Veremos qué hace el método **ngOnInit** cuando lleguemos al ciclo de vida de los componentes. Por ahora, vamos a centrarnos en aspectos más básicos.

Vamos a definiar la plantilla (**product-list.component.html**) para el componente que acabamos de crear. Mostraremos una tabla (sólo los encabezados por ahora) donde listaremos los productos.

```
<div class="card">
  <div class="card-header bg-primary text-white">
    My product's list
  </div>
  <div class="card-block">
    <div class="table-responsive">
      <table class="table table-striped">
        <thead>
          <tr>
            <th>Producto</th>
            <th>Precio</th>
            <th>Disponible</th>
          </tr>
        </thead>
        <tbody>
                    <!-- Aquí van los productos. Por ahora se queda vacío -->
        </tbody>
      </table>
    </div>
  </div>
</div>
```

4.3 *Usando el selector de un componente*

Una vez creado el componente, para que sea visible, debemos incluir su selector (**product-list** en este caso) en la plantilla de otro/s componente/s. Vamos a situarlo en la plantilla del componente principal de la aplicación (app.component.html), borrando previamente todo el contenido de ejemplo de la misma. De paso vamos a cambiar el título de la aplicación por “Productos Angular”.

```
<div class="container">
  <h1>
```

```
    Bienvenido/a a {{title}}
  </h1>
  <product-list></product-list>
</div>
```

Si ejecutamos **ng serve** (http://localhost:4200), veremos lo siguiente:

4.4 *Interpolación*

Como vimos en la sección "Componentes", se utiliza un concepto llamado interpolación para mostrar en la vista las propiedades de un componente. Cuando ponemos la propiedad dentro de {{dobles llaves}}, esta se reemplaza por el valor.

Mediante la interpolación podemos hacer lo mismo que cuando asignamos un valor a alguna variable: concatenar valores, mostrar propiedades, llamar a métodos del componente (se muestra lo que devuelven), cálculos matemáticos, etc.

```
{{title}}
{{'Title: ' + title}}
{{'Title: ' + getTitle()}}
{{4*54+6/2}}
```

Vamos a poner el título de la lista de productos y los encabezados de la tabla en propiedades dentro del componente y mostrarlos en la plantilla mediante interpolación (esto permite modificarlas, traducirlas, etc. fácilmente y en tiempo real).

```
export class ProductListComponent implements OnInit {
  title = 'Mi lista de productos';
  headers = {description: 'Producto', price: 'Precio', available: 'Disponible'};

  constructor() { }

  ngOnInit() { }
}
```

Usando la interpolación, vinculamos esas propiedades en la plantilla:

```
<div class="card">
  <div class="card-header bg-primary text-white">
    {{title}}
  </div>
  <div class="card-block">
    <div class="table-responsive">
      <table class="table table-striped">
        <thead>
          <tr>
            <th>{{headers.description}}</th>
            <th>{{headers.price}}</th>
             <th>{{headers.available}}</th>
          </tr>
        </thead>
        <tbody>
          <!-- Aquí van los productos. Por ahora se queda vacío -->
```

```
      </tbody>
    </table>
  </div>
 </div>
</div>
```

4.5 *Directivas estructurales (*ngFor, *ngIf)*

Vamos a añadir algunos productos a nuestra tabla. Antes de nada, vamos a decirle a TypeScript qué propiedades (y tipos) debe tener un producto, lo que nos permitirá que no nos equivoquemos al definirlos y activará el autocompletado en el editor. Para ello creamos una interfaz llamada IProduct donde definiremos dichas propiedades. Desde el directorio principal del proyecto ejecutamos:

```
ng g interface interfaces/i-product
```

Ahora definiremos las propiedades de la interfaz dentro del archivo recién creado.

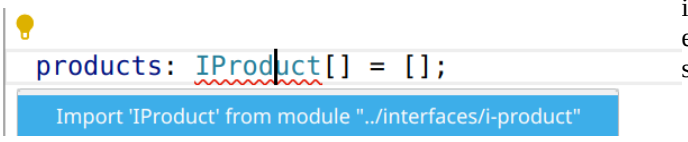
```
export interface IProduct {
  id: number;
  description: string;
  price: number;
  available: Date;
  imageUrl: string;
  rating: number;
}
```

Antes de seguir, vamos a mostrar la tabla sólo cuando haya productos para listar. Si no hay productos, la tabla directamente no aparecerá (no estará en el DOM).

Esto se consigue con la directiva de Angular ***ngIf**. Esta directiva se añade como atributo a un elemento HTML, y se establece una condición como valor. Si es cierta, el elemento se mostrará, y cuando sea falsa, el elemento pasará a estar oculto. En este caso, la condición será que exista el array **products** y no esté vacío.

```
<div class="table-responsive" *ngIf="products && products.length">
 <table class="table table-striped">
  <thead>
   <tr>
    <th>{{headers.desc}}</th>
    <th>{{headers.price}}</th>
    <th>{{headers.avail}}</th>
   </tr>
  </thead>
  <tbody>
   <!-- Leave empty now -->
  </tbody>
 </table>
</div>
```

Prueba a crear un array de productos vacío en el componente para ver que la tabla no aparece. Las propiedades de los productos están en la interfaz IProduct, por lo que si queremos activar el autocompletado y la detección de errores en los nombres de las propiedades, se lo tenemos que indicar a TypeScript. Además, debemos importar la interfaz (el editor se encargará de avisarnos y darnos la solución al situar el cursor encima).:



Finalmente, añadimos un par de productos al array con sus propiedades. Las imágenes tienen que situarse en el directorio **src/assets**:

```
import { Component, OnInit } from '@angular/core';
import { IProduct } from '../interfaces/iproduct'
```

```
@Component({
  selector: 'product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {
  title = "My product's list";
  headers = {desc: 'Product', price: 'Price', avail: 'Available'};

  products: IProduct[] = [{
    id: 1,
    description: 'SSD hard drive',
    available: new Date('2016-10-03'),
    price: 75,
    imageUrl: 'assets/ssd.jpg',
    rating: 5
  },{
    id: 2,
    description: 'LGA1151 Motherboard',
    available: new Date('2016-09-15'),
    price: 96.95,
    imageUrl: 'assets/motherboard.jpg',
    rating: 4
  }];

  constructor() { }

  ngOnInit() { }

}
```

La directiva ***ngFor** nos permite iterar por esta colección de productos y generar para cada uno el HTML necesario (una fila de la tabla) para mostrarlos.

```
<tbody>
  <tr *ngFor="let product of products">
    <td>{{product.description}}</td>
    <td>{{product.price}}</td>
    <td>{{product.available}}</td>
  </tr>
</tbody>
```

Esto equivale a la instrucción **foreach** en muchos lenguajes. Para cada producto de la lista, se crea una variable llamada **product** y se le asignará el objeto correspondiente a cada iteración. La estructura donde se aplica la directiva (<tr>), se repetirá tantas veces como productos haya en el array.

Bienvenido/a a Productos Angular

Mi lista de productos

Producto	Precio	Disponible
SSD hard drive	75	Mon Oct 03 2016 02:00:00 GMT+0200 (hora de verano de Europa central)
LGA1151 Motherboard	96.95	Thu Sep 15 2016 02:00:00 GMT+0200 (hora de verano de Europa central)

Si examinamos el DOM desde las herramientas de desarrollador del navegador, veremos el HTML que ha generado Angular con los datos de los productos gracias a la directiva ***ngFor** y a la interpolación.

```
▼<tbody _ngcontent-mew-2>
   <!--template bindings={
     "ng-reflect-ng-for-of": "[object Object],[object Object]"
   }-->
  ▼<tr _ngcontent-mew-2>
     <td _ngcontent-mew-2>SSD hard drive</td>
     <td _ngcontent-mew-2>75</td>
     <td _ngcontent-mew-2>Mon Oct 03 2016 02:00:00 GMT+0200 (CEST)</td>
   </tr>
  ▼<tr _ngcontent-mew-2>
     <td _ngcontent-mew-2>LGA1151 Motherboard</td>
     <td _ngcontent-mew-2>96.95</td>
     <td _ngcontent-mew-2>Thu Sep 15 2016 02:00:00 GMT+0200 (CEST)</td>
   </tr>
 </tbody>
```

La fecha y el precio todavía no tienen un formato legible (dd/mm/yyyy y 0.00€). Veremos como formatear estos datos más adelante usando filtros (pipes) predefinidos en Angular (y aprenderemos a crear nuevos filtros).

La directiva ***ngFor** tiene propiedades implícitas que podemos utilizar:

- **index: number**: El índice (en el array) del elemento actual.
- **first: boolean**: True cuando estamos en el primer elemento.
- **last: boolean:** True cuando es el último elemento.
- **even: boolean:** True cuando el índice actual es par.
- **odd: boolean**: True cuando el índice actual es impar.

Para usar alguna de estas propiedades, debes asignarlas a una variable (separando las asignaciones por punto y coma ';'), como se puede ver en el siguiente ejemplo:

```
<tr *ngFor="let product of products; let i = index; let isEven = even">
  <td [ngClass]="{'even': isEven}">{{'Index: ' + i}}</td>
  ...
</tr>
```

También se pueden asignar las variables con la sintaxis "as":

```
<tr *ngFor="let product of products; index as i; even as isEven">
```

Lo que hace este ejemplo es asignar una clase CSS (even) a la primera celda de los productos con el índice par (0, 2, 4, 6, 8, ...).

Para probar esto, crea una celda adicional en la tabla (número de producto) y crea un selector CSS para la clase even (.even) y dale algo de estilo estilo diferenciador. Cuando continuemos desarrollando este ejemplo, no tendremos en cuenta esta celda de la tabla, por lo que recomiendo borrarla una vez probado que funciona.