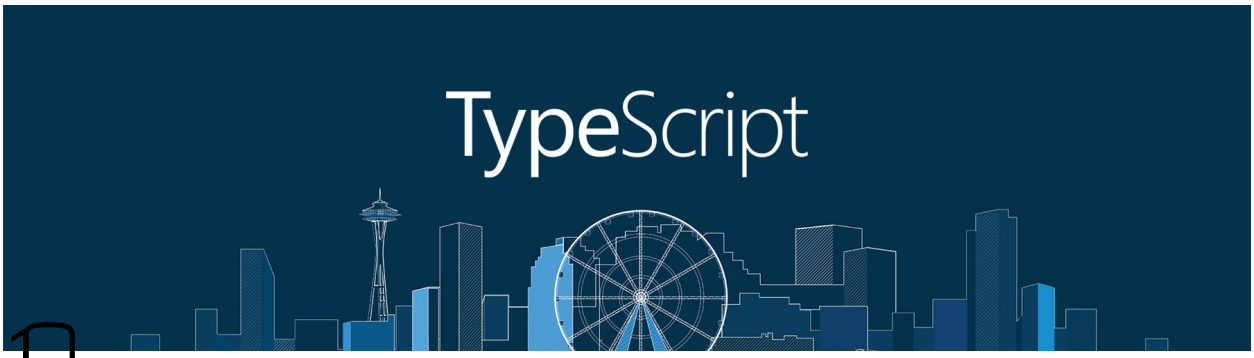


Angular S1: Typescript | Angular

Después de esta lección podrás:

1. Usar Typescript en tus proyectos.
2. Generar proyectos bien estructurados y escalables.
3. Trabajar con Clases e Interfaces.

TypeScript

Typescript es un lenguaje parecido a Javascript, pero que añade la posibilidad de tipar los valores o elementos con los que vamos a trabajar a lo largo de nuestra aplicación. De hecho Typescript se acaba transpilando a Javascript, para que lo interprete el navegador.

Lo creó Microsoft para darle más potencia a la programación web, con tipado, clases, etc. Luego llegó ES6 y demás estándares de Javascript y ahora, tanto TypeScript como JavaScript, son buenas opciones para desarrollo web.

En primer lugar, instalaremos TypeScript de forma global para que podamos compilar archivos desde nuestro terminal.

```
npm install -g typescript
```

Y para compilar un fichero (por ejemplo "nombreFichero.ts"), y crear su fichero javascript correspondiente ejecutaremos la siguiente sentencia:

```
tsc nombreFichero.ts
```

Principales Características de Typescript

La principal característica de Typescript es el tipado estático, esto significa que cuando declaramos que una variable es de un tipo, solo podremos almacenar en ella variables de ese tipo, ¿Tiene sentido no?

Recordemos que en JavaScript podemos almacenar cualquier tipo de dato en cualquier variable, sin restricción de tipos. Esto es potente, pero algo peligroso, por eso surgió Typescript.

Vamos a ver algunas características importantes de Typescript

Variables:

Las variables tienen un tipo de dato, es decir solo pueden almacenar un tipo restringido. Puede ser `boolean`, `string`, `number`...

Por ejemplo:

```
let name: string = "Pepe"; // Solo puede almacenar strings
let edad: number = 22; // Solo puede almacenar numeros
let human: boolean = true; // Solo puede almacenar booleanos
let numberList: number[] = [1];
let numberList: number[] = [1];

// Existe el tipo de dato 'any', que admite cualquier cosa
let whatever: any = "mal";
// Por lo general es una mala práctica su uso
```

Valores:

Los valores sólo se pueden asignar a variables del tipo correspondiente. Si una variable es de tipo `number` solamente le podremos asignar un `number`.

Por ejemplo:

```
let edad : number; // Asignamos el tipo number para la variable edad
edad = 20; // La variable ahora sólo puede asignar valores del tipo number
edad = "Veinte"; // Esto dará error, porque "veinte" no es un número
```

Funciones (parámetros y valor de retorno):

Las funciones con parámetros y que devuelven un valor, ahora también han de ser tipadas.

Por ejemplo:

```
// Los parámetros están ahora tipados con 'number'
// Y al final además también tenemos ':number' que es el valor de retorno
function addTwoNumbers(a: number, b: number): number {
  return a + b;
}

let addedNumber: number = addTwoNumbers(1, 2);

console.log(addedNumber);
```

En caso de que una función no devuelva nada, debemos escribirla con `void` como en el siguiente ejemplo:

```
function sayMyName(name: string): void {
  console.log('Your name is ' + name);
}
```

Error de tipado:

En caso que usemos un argumento con tipado inválido, la compilación nos notifica el error:

```
function getMyName(name: string): string {
  return 'Your name is ' + name;
}
```

```
let miNombre : number = 5;
let texto = getMyName(miNombre); // ERROR en el tipo del parámetro
console.log(texto);
```

```
error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'.
```

Interfaces:

Una interfaz nos sirve para declarar un **modelo** de datos, solo con los valores internos que lo componen, sin más lógica ni código de programación. Es decir, nos ayuda a crear objetos definiendo sus propiedades. Por ejemplo, la interfaz persona, tendría dentro dos propiedades: nombre y edad.

```
interface Persona {
  nombre: string;
  edad: number;
}

let juan : Persona = { nombre: 'Juan Perez', age: 20 };
console.log(juan);
```

Clases:

Una clase es similar a una interfaz, en el sentido de que también la componen los **atributos** del **objeto**. Pero en este caso, además podremos declarar **funciones** específicas de ese objeto. A continuación, por ejemplo, vamos a declarar la clase persona y a definir una función para obtener su nombre:

```
class Persona {
  nombre: string;
  edad: number;

  // Además de definir los atributos, tenemos funciones, como el constructor
  constructor(nombre: string, edad: number) {
    this.nombre = nombre;
    this.edad = edad;
  }

  // También podemos declarar otras funciones
  getName(): string {
    return this.nombre;
  }
}

// Usaremos 'new' para generar nuevas instancias de la clase (haciendo uso del constructor interno)
let persona = new Persona('Juan Perez', 20);
let persona2 = new Persona('Julio Iglesias', 68);

console.log(persona.nombre)
console.log(persona2.nombre)


let nombre = persona.getName();
console.log(nombre)
```

Interfaces + clases:

También podemos combinar los dos elementos anteriores. Se dice que una clase **implementa** una interfaz, cuando posee todas las propiedades de la interfaz y además la extiende con sus propias funciones:

```
// mutant.ts
interface MutantInterface {
  name: string;
```

```
  power: string;
  human: boolean
}


class Mutant implements MutantInterface {
  name: string;
  power: string;
  human: boolean;

  constructor(name: string, power: string, human: boolean) {
    this.name = name;
    this.power = power;
    this.human = human;
  }
}

let ciclops = new Mutant('Scott', 'can emit energy from his eyes', true);

console.log(ciclops);
```

Esto dará como resultado una clase que usa MutantInterface para definir su estructura. Un poco redundante pero mucho más sólido.

Extender Clases:

La declaración de clases permite que podamos extenderlo a clases hijas. Las clases hijas tienen acceso a las propiedades de la clase padre. Aquí os dejamos un ejemplo:

```
// mutant.ts
class Human {
  name: string;
  human: boolean;

  constructor(name: string, human: boolean) {
    this.name = name;
    this.human = human;
  }
}

class Mutant extends Human {
  mutant: boolean;

  // Por defecto, vamos a fijar que nuestros mutantes son humanos, por si no nos pasan el parámetro
  constructor(mutant: boolean, name: string, human: boolean = true) {
    super(name, human); // La función super ejecuta el constructor de la clase padre

    this.mutant = mutant;
  }
}

let myMutant = new Mutant(true, 'Logan', true);
let myMutant2 = new Mutant(true, 'Storm');

console.log(myMutant);
```

Extender Interfaces:

Las interfaces también se pueden extender. Por ejemplo:

```
// mutant.ts
interface Human {
  name: string;
  human: boolean;
}

interface Mutant extends Human {
  mutant: boolean;
}
```

```
let myMutant : Mutant = {
  name: 'Hank McCoy',
  human: true,
  mutant: true,
};

console.log(myMutant);
```

Atributos opcionales:

Qué pasaría si no queremos que nuestro `Mutant` tenga que definir como obligatorio si es mutante o no, ahora mismo nos devolvería un error de compilación.

¿Qué podemos hacer para evitar esto? Es sencillo, agregamos un signo de **interrogación** a la definición del atributo y listo. Esto hace que el campo sea opcional y pueda tener un valor o no, puede usarse o no.

Por ejemplo:

```
// mutant.ts
interface Human {
  name: string;
  human: boolean;
}

interface Mutant extends Human {
  mutant?: boolean;
}

let myMutant : Mutant = {
  name: 'Hank McCoy',
  huamn: true,
};

console.log(myMutant)
```

Atributos públicos y privados:

Cada vez que declaremos atributos (o funciones), debemos decidir el ámbito de acceso. ¿Esto qué quiere decir? Pues que podemos restringir el acceso los elementos en función de:

- `public` por defecto, accesible desde cualquier punto y sin restricciones
- `private` no accesible desde fuera de la clase o ámbito
- `protected` no accesible desde fuera de la clase, excepto por instancias que extiendan de la misma

```
class Car {
  private numBastidor: string;
  protected motor: string;
  public color: string;
  // color: string;

  saveData(car: Car): void {
    this.numBastidor = "000"; // ok
    this.motor = "110CV"; // ok
    this.color = "Rojo"; // ok

    car.numBastidor = "1111"; // ok
    car.motor = "90CV"; // ok
    car.color = "Verde"; // ok
  }
}

class Fiat extends Car {
  getData(car: Car, fiat: Fiat) {
```

```
    this.motor = "82CV"; // ok
    this.color = "Negro"; // ok

    fiat.motor = "130CV"; // ok
    car.color = "Amarillo"; // ok
    fiat.color = "Azul"; // ok

    car.numBastidor = "2222"; // err
    car.motor = "150CV"; // err

    fiat.numBastidor = "3333"; // err
  }
}
```

Angular Intro

Después de esta lección podrás:

1. Entender qué es Angular y la potencia que nos da para el desarrollo de una aplicación web.
2. Generar tu primer proyecto bien estructurado con el CLI.
3. Entender los primeros artefactos de Angular.

¿Qué es y por qué usar Angular?

Angular es un *framework* (caja de herramientas) creado por Google en TypeScript de código abierto, pensado para usarse en el desarrollo de SPA (*Single Page Application*).

Nos va a permitir desarrollar una **aplicación web** gracias a los artefactos que lo componen, abstrayéndonos de las complicaciones. Las razones que respaldan su uso son muchas, pero resumiendo podemos decir que es un framework: **estructurado**, **escalable**, con una gran **comunidad** (proyectos) y una **documentación** excelente.

Antes de comenzar debemos instalar el CLI de Angular. El **CLI** (*Command Line Interface*) es un paquete de Angular que nos va a permitir crear proyectos, componentes y enlazar todas las piezas de nuestra aplicación:

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

Ahora, ya podemos usar el CLI, mediante el comando `ng` en nuestra terminal. Crearemos nuestra primera aplicación de Angular mediante:

```
ng new upgrade-app
```

La consola del terminal nos irá haciendo preguntas sobre cómo queremos generar el nuevo proyecto, debemos ir contestando sí o no a algunas configuraciones. Por ahora contestar `y` a todo y fijar `SCSS` para los estilos, y a futuro ya iremos profundizando para comprenderlo mejor.

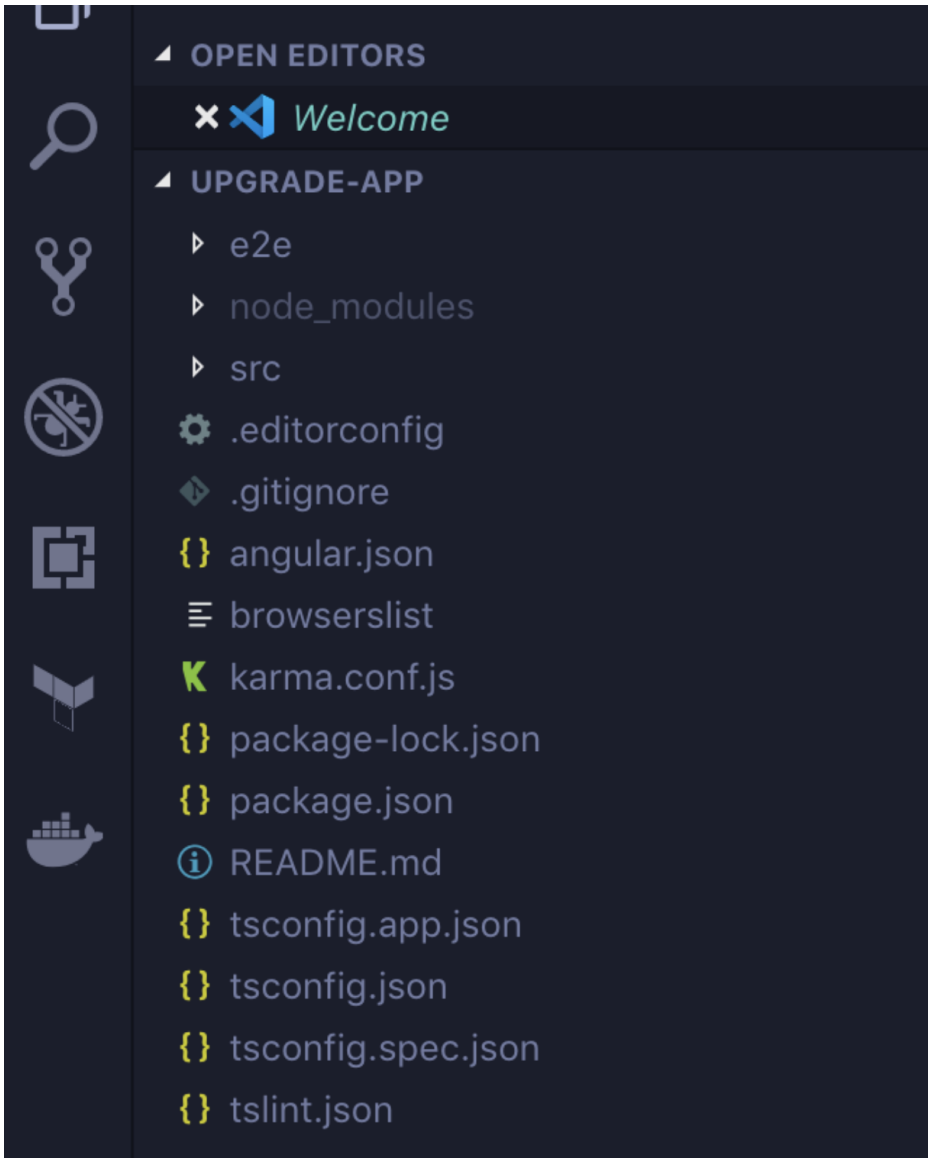
- *Would you like to add Angular routing?* `Yes`
- *Which stylesheet format would you like to use?* `SCSS`

Tu proyecto debería verse en Chrome, ejecutando el comando `serve`:

```
cd upgrade-app
ng serve --open
```

Pista: Para apagar el servidor de Angular y volver a la terminal, basta con pulsar CTRL + C.

Una vez creada la app, comenzaremos a echar un vistazo a los ficheros y estructura:



Como vemos, el CLI nos ha adelantando un montón de trabajo, creando un montón de ficheros y directorios:

- `/e2e` será un directorio para test de integración (olvidémonos por el momento)
- `/node_modules` es la carpeta donde NPM descarga las librerías que complementan nuestra aplicación web, mejor no tocar aquí :)
- `/src` el directorio **más importante**, aquí es donde se concentra la aplicación angular, son los ficheros fuente (*source*)
- El resto de ficheros de la raíz son en su mayoría ficheros de configuración, sobre Angular, sobre Karma (motor de testing), sobre NPM y sobre TypeScript. Detallemos los más importantes:

- `tslint.json` contiene las reglas básicas de lintado con TypeScript. El linter de **TSLint** es una herramienta que nos va a ayudar a crear código más limpio.
- `tsconfig.json` tiene la configuración del *build* de nuestro código TypeScript. Es decir, de cómo se construirá la aplicación final que servimos a la web.

Y ahora, centrémonos en la carpeta `/src/app` de nuestro proyecto, ya que aquí desarrollaremos todo nuestro código de la aplicación. A partir de aquí, empezaremos a hablar de **Módulos y Componentes**, pero no os asustéis, tendremos una sección específica para explicar estos artefactos al detalle.

app.module.ts

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

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

¿Qué está haciendo este fichero? Está creando el **módulo principal** de la aplicación. `@NgModule` es un **decorador** que toma un solo objeto de metadatos, cuyas propiedades describen el módulo. Las propiedades son las siguientes.

- **Declaraciones**: los componentes, directivas y pipes que pertenecen a este NgModule.
- **Exportaciones**: declaraciones que deberían ser visibles y utilizables por los componentes de otros NgModules.
- **Importaciones**: otros módulos cuyas clases exportadas son necesarias para los componentes declarados en este NgModule.
- **Proveedores**: proveedores de servicios (profundizaremos cuando veamos los **servicios**).
- **Bootstrap**: indica el componente de entrada a nuestra aplicación. Solo el **NgModule raíz** debe establecer esta propiedad.

app-routing.module.ts

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


const routes: Routes = [];

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

Este otro módulo está enfocado al **sistema de enrutamiento**, es decir, a establecer las rutas para navegar por nuestra aplicación. En el apartado de rutas aprenderemos a crear diferentes secciones en nuestra app, para navegar por varias

vistas distintas.

Fijaos cómo este módulo se importó en el anterior **app.module.ts** y se configuró dentro de la matriz de **imports**.

app.component.ts

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

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

Este sería un componente de Angular. El decorador `@Component` nos permite declarar componentes indicando su:

- `selector` etiqueta que usaremos en el HTML
- `template` asociando la vista HTML a la lógica que habrá en el fichero de TypeScript
- `style` indicando los ficheros de estilos para darle forma al componente

Aprenderemos más sobre los componentes en la siguiente sección.

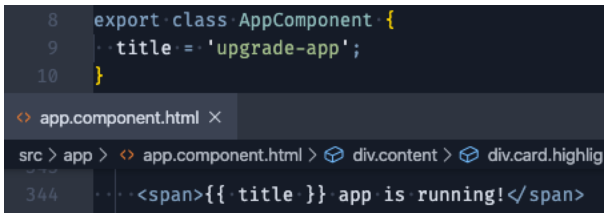
app.component.html

```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo" src="
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour of Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
  </li>
</ul>

<router-outlet></router-outlet>
```

Esta sería la vista HTML asociada al componente que acabamos de mencionar. Los elementos que se visualizan ya nos deberían sonar: `div`, `h1`, `li`, `a`, etc.

Pero podemos destacar cosas nuevas, como por ejemplo en la **línea 4** del ejemplo `{{ title }}`, es nuestra primera interpolación. Es la manera en la que la **vista** puede mostrar **datos** del fichero de **lógica** TypeScript.



También vemos la declaración del `<router-outlet>`, componente que permite hacer uso del enrutado que mencionamos anteriormente.

index.html

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

¡Por fin, un fichero que nos suena! El `index.html` será la primera vista de entrada a nuestra aplicación, como en cualquier aplicación web.

La magia de **Angular** comienza con el uso del componente `<app-root>` dentro del `body`. De ahí en profundidad, crece nuestra aplicación Angular. En el fichero **app.componente.ts** declaramos el selector `app-root`. ¡Todo empieza a encajar!

Ahora que ya hemos repasado los primeros ficheros de nuestra aplicación, vamos a profundizar en cómo funciona un módulo Angular y cómo ir creando más componentes 🚀

Angular: Components

Después de esta lección podrás:

1. Entender cómo declarar componentes.
2. Organizar un proyecto Angular de manera escalable.

Modulos Angular

Los primeros artefactos que vamos a estudiar son los **módulos** Angular. ¿Qué es un módulo? Pues un módulo es un conjunto de utilidades, todas juntas en la misma cápsula. La principal utilidad que tiene los módulos, es que nos permiten escalar nuestra aplicación. Entre los módulos más importantes definidos en Angular, tenemos:

- `BrowserModule` módulo de infraestructura, permite hacer funcionar la app.
- `CommonModule` módulo de directivas principales de Angular (profundizaremos más adelante).

- `FormsModule` módulo para creación de formularios.
- `ReactiveFormsModule` módulo para la creación de formularios reactivos.
- `RouterModule` módulo orientado al enrutamiento (profundizaremos más adelante).
- `HttpClientModule` módulo preparado para hacer peticiones al servidor mediante servicios.

NgModule	Import it from	Why you use it
`BrowserModule`	`@angular/platform-browser`	When you want to run your app in a browser
`CommonModule`	`@angular/common`	When you want to use `NgIf`, `NgFor`
`FormsModule`	`@angular/forms`	When you want to build template driven forms (includes `NgModel`)
`ReactiveFormsModule`	`@angular/forms`	When you want to build reactive forms
`RouterModule`	`@angular/router`	When you want to use `RouterLink`, `.forRoot()`, and `.forChild()`
`HttpClientModule`	`@angular/common/http`	When you want to talk to a server

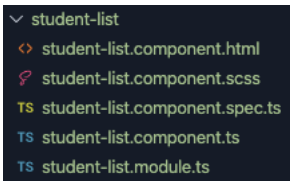
Componentes

¿Qué son los componentes? Los componentes nos sirven para ir construyendo nuestra aplicación a base de elementos. Cada componente tiene su fichero de **lógica**, su **vista** y sus **estilos**. Este enfoque nos ayuda de nuevo a organizar el código por bloques, y la posibilidad de reutilizar estos bloques en varios sitios.

Para generar nuestro primer componente, vamos a seguir unos pasos muy parecidos a los anteriores, nos vamos a apoyar en el CLI de la siguiente manera:

```
cd src/app
ng generate component student-list
```

Ahora nuestro app.module contiene **student-list-component**:



Veamos ahora nuestro nuevo **student-list.component.ts**:

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

@Component({
  selector: 'app-student-list',
  templateUrl: './student-list.component.html',
  styleUrls: ['./student-list.component.scss']
})
export class StudentListComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

Toda la lógica asociada a este componente deberá ir en el interior de esta clase, por lo que iremos codificando atributos y funciones, pero antes de empezar, vemos dos funciones. ¿Qué son y cuáles son las diferencias entre **constructor** y **ngOnInit**?

El método `constructor` permite la creación del componente y la asignación de los parámetros de entrada a los valores iniciales a los atributos del componente.
El método `ngOnInit` maneja el comportamiento que el componente seguirá cuando aparezca por primera vez.

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

@Component({
  selector: 'app-student-list',
  templateUrl: './student-list.component.html',
  styleUrls: ['./student-list.component.scss']
})
export class StudentListComponent implements OnInit {
  title: string;

  constructor() {
    console.log(this.title);
    this.title = 'Listado de Alumnos';
    console.log('constructor finalizado');
  }

  ngOnInit() {
    console.log(this.title);
    console.log('ngOnInit finalizado');
  }
}
```

Estos dos anteriores métodos se generan automáticamente, pero existen otros métodos que tienen que ver con el **ciclo de vida** de la clase. El ciclo de vida de un componente está compuesto por diferentes etapas que van desde su construcción, pasando por su uso, hasta su destrucción.

Conviene destacar otras dos funciones importantes:

- `ngOnChanges` este evento será lanzado por cada cambio en las propiedades del componente
- `ngOnDestroy` este evento se ejecuta justo antes de destruir un componente, nos será útil para "hacer limpieza" y no dejarnos cables sueltos

Para comprobar como se construye nuestro componente, debemos usarlo en nuestra aplicación. Primero debemos exportarlo en su propio módulo, para poder usarlo en otros puntos.

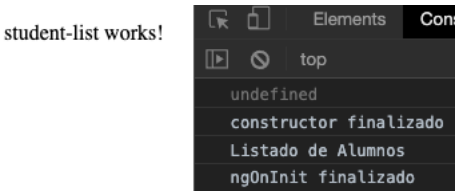
```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { StudentListComponent } from './student-list.component';

@NgModule({
  declarations: [StudentListComponent],
  imports: [
    CommonModule
  ],
  exports: [
    StudentListComponent
  ],
})
export class StudentListModule { }
```

Seguidamente, abrimos el fichero **app.component.html** y escribimos el selector de nuestro componente:

```
<app-student-list></app-student-list>
```

¡Y listo! Tenemos nuestro primer componente creado. Para comprobarlo, puedes lanzar tu aplicación Angular desde una terminal con el comando `ng serve --open`:



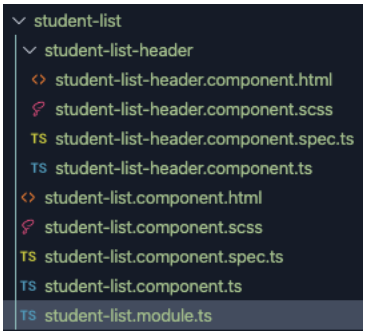
Componentes hijos

Pues ahora que ya tenemos nuestro primer componente, vamos a aprender a generar componentes hijos, para que nuestra aplicación siga creciendo.

De manera similar al apartado anterior, vamos a crear un componente hijo llamado **student-list-header**:

```
cd src/app/student-list
ng generate component student-list-header
```

Como vemos, se genera una carpeta separada, con el nuevo componente hijo.



Y para comprobar su visualización, nos dirigimos a **student-list.component.html** y escribimos la nueva etiqueta:

```
<app-student-list-header></app-student-list-header>
```

Si todo funciona correctamente deberíamos ver esto en nuestro navegador:

```
<p>student-list works!</p>
```

Un apunte importante, es que si nos fijamos en el contenido del módulo, veremos que el nuevo componente se ha incluido en el array de **declarations**, pero esta vez no ha hecho falta incluirlo en el array de **exports**. ¿Por qué?

Porque el componente padre lo hemos usado fuera del módulo **student-list**, ¿recordáis? hace falta que lo exportemos para poder usarlo al nivel del **app.component.html**. Sin embargo este componente hijo, lo estamos usando en el propio módulo interno de **student-list**, por lo que no ha hecho falta exportarlo.

De todas maneras, si se produce algún **error**, siempre podéis consultar la consola del navegador Chrome, o la propia consola del terminal. Angular es muy verboso a la hora de informarnos que algo va mal.