Curso de Typescript

Diego Saavedra

Jun 14, 2024

Table of contents

1 Bienvenido

Figure 1.1: Typescript

¡Bienvenido al Curso de Typescript!

En este curso, exploraremos todo, desde los fundamentos hasta las aplicaciones prácticas.

1.1 ¿De qué trata este curso?

Este curso es una introducción a TypeScript, un lenguaje de programación de código abierto desarrollado y mantenido por Microsoft. TypeScript es un superconjunto de JavaScript que agrega tipado estático opcional y otras características avanzadas a JavaScript.

En este curso, aprenderá los conceptos básicos de TypeScript, incluidos los tipos de datos, las funciones, las clases, los módulos y mucho más. También explorará cómo TypeScript se puede utilizar para crear aplicaciones web modernas y escalables.

Este curso es ideal para principiantes y aquellos con poca o ninguna experiencia en programación. Si eres un estudiante curioso, un profesional que busca cambiar de carrera o simplemente alguien que quiere aprender TypeScript, este curso es para ti.

1.2 ¿Para quién es este curso?

Este curso es para cualquier persona interesada en aprender TypeScript, incluidos:

- Estudiantes que deseen aprender un nuevo lenguaje de programación.
- Profesionales que buscan mejorar sus habilidades de desarrollo web.
- Desarrolladores que deseen aprender TypeScript para crear aplicaciones web modernas y escalables.
- Cualquiera que quiera aprender un lenguaje de programación de código abierto y de alto rendimiento.
- Cualquiera que quiera aprender TypeScript para mejorar su carrera profesional.

- Cualquiera que quiera aprender TypeScript para crear aplicaciones web modernas y escalables.

1.3 ¿Cómo contribuir?

Valoramos su contribución a este curso. Si encuentra algún error, desea sugerir mejoras o agregar contenido adicional, me encantaría saber de usted.

Puede contribuir a través del repositorio en linea, donde puede compartir sus comentarios y sugerencias.

Juntos, podemos mejorar continuamente este recurso educativo para beneficiar a la comunidad de estudiantes y entusiastas de la programación.

Este ebook ha sido creado con el objetivo de proporcionar acceso gratuito y universal al conocimiento.

Estará disponible en línea para cualquier persona, sin importar su ubicación o circunstancias, para acceder y aprender a su propio ritmo.

Puede descargarlo en formato PDF, Epub o verlo en línea en cualquier momento y lugar.

¡Gracias por su interés en este curso y espero que disfrute aprendiendo TypeScript!

Part I

Unidad 1: Introducción a Typescript

2 Introducción a TypeScript

Figure 2.1: Typescript

2.1 Historia y Evolución de TypeScript

2.1.1 Origen de TypeScript:

TypeScript fue desarrollado por **Microsoft** y anunciado por primera vez en **octubre de 2012**. Su creador principal es **Anders Hejlsberg**, también conocido por ser el creador de **C#**.

La necesidad de TypeScript surgió debido a la **creciente complejidad de las aplicaciones web** y la necesidad de un **lenguaje que pudiera manejar mejor el desarrollo a gran escala**.

2.1.2 Evolución:

- **2012:** Versión 0.8 - Lanzamiento inicial de TypeScript.
- **2014:** Versión 1.0 - Primer lanzamiento estable, introduciendo características como módulos externos y compatibilidad con ECMAScript 6.
- **2016:** Versión 2.0 - Se introdujeron mejoras significativas como tipos no null y control de flujo basado en tipos.
- **2018:** Versión 3.0 - Añadió soporte para proyectos referenciados y mejoras en la experiencia de desarrollo.
- **2020:** Versión 4.0 - Introducción de nuevas características como variadic tuple types, label inference en destructuring y mejoras en la inferencia de tipos.
- **Actualidad:** TypeScript sigue evolucionando con nuevas versiones que mejoran la robustez del lenguaje y la experiencia del desarrollador.

2.2 Diferencias entre JavaScript y TypeScript

2.2.1 Tipado Estático vs. Tipado Dinámico:

JavaScript: Lenguaje de tipado dinámico, donde los tipos se determinan en tiempo de ejecución.

TypeScript: Añade tipado estático, permitiendo definir tipos durante la escritura del código, lo que ayuda a detectar errores antes de la ejecución.

Gracias al tipado estático, TypeScript se ha convertido en una herramienta popular para el desarrollo de aplicaciones web ya que es más seguro y fácil de mantener en compraración con JavaScript.

2.2.2 Compatibilidad:

JavaScript: Todo código JavaScript válido es código TypeScript válido, lo que significa que TypeScript es un superconjunto de JavaScript.

TypeScript: Permite usar las últimas características de ECMAScript, y transpila el código a versiones anteriores de JavaScript para compatibilidad con navegadores más antiguos.

> Tip
>
> **¿Qué es transpilación de código?**
> La transpilación es el proceso de convertir código de un lenguaje a otro. En el caso de TypeScript, el código, la herramienta más utilizada es el compilador de TypeScript (tsc), que convierte el código TypeScript a JavaScript. Sin embargo tambien se puede usar Babel para transpilar código TypeScript a JavaScript.

2.2.3 Características Adicionales:

Interfaces: TypeScript permite definir interfaces que ayudan a describir la forma de los objetos.

Enumeraciones (enums): TypeScript introduce enums, permitiendo definir conjuntos de constantes con nombres.

Decoradores: TypeScript soporta decoradores, que son una forma de modificar clases y métodos.

2.3 Instalación y Configuración del Entorno

En node.js se puede utilizar distintos **gestores de paquetes**, dentro de ellos tenemos **npm**, **yarn** y **pnpm**. En este caso se utilizará npm.

> **Tip**
>
> Un gestor de paquetes es una herramienta que permite instalar y gestionar dependencias de proyectos de forma sencilla.

En ocasiones es necesario tener más de una versión de node.js en el sistema, para ello se pueden utilizar herramientas como **nvm**, **fnm**, **brew** o **chocolatey**, inclusive **Docker** es una alternativa que se ha popularizado para tener distintas versiones de node.js, en este caso se utilizará **fnm**.

2.3.1 Ecosistema de JavaScript y TypeScript:

Es neceario conocer algunas herramientas y tecnologías que se utilizan en el ecosistema de JavaScript y TypeScript, entre ellas se encuentran:

Node.js: es un entorno de ejecución de JavaScript que permite ejecutar código JavaScript en el lado del servidor, existen otras alternativas que se han popularizado ultimamente como Deno, Bun, etc.

Deno: Es un entorno de ejecución de JavaScript y TypeScript que se ha vuelto popular por su seguridad, rendimiento y facilidad de uso.

Bun: Es un entorno de ejecución de JavaScript y TypeScript que se ha vuelto popular por su velocidad y facilidad de uso.

En este curso utilizaremos Node.js para el desarrollo de aplicaciones web con TypeScript

Npm: Es el gestor de paquetes de Node.js que se utiliza para instalar y gestionar dependencias de proyectos, existen otras alternativas como Yarn y pnpm.

Yarn: Es un gestor de paquetes de Node.js que se ha vuelto popular por su velocidad y facilidad de uso.

Pnpm: Es un gestor de paquetes de Node.js que se ha vuelto popular por su velocidad y facilidad de uso.

En este curso utilizaremos npm para instalar y gestionar dependencias de proyectos.

2.3.2 Instalación de Node.js y npm:

Para instalar **Node.js** y **npm** se puede utilizar el instalador oficial de Node.js, para ello se recomienda seguir los siguientes pasos:

1. Descargar el instalador de Node.js desde la página oficial: Node.js
2. Ejecutar el instalador y seguir las instrucciones.
3. Verificar la instalación ejecutando los siguientes comandos en la terminal:

```
node -v ①
npm -v ②
```

① Muestra la versión de Node.js instalada.
② Muestra la versión de npm instalada.

3 Extra: nvm, fnm, brew, chocolate

En ocasiones será necesario probar proyectos con distintas versiones de Node.js, para ello se pueden utilizar herramientas como nvm, fnm, brew o chocolatey.

> Tip
>
> **¿Qué es son nvm, fnm, brew o chocolatey?**
> Son herramientas que permiten instalar y gestionar distintas versiones de Node.js en el sistema.

- **nvm:** Node Version Manager, es una herramienta que permite instalar y gestionar distintas versiones de Node.js en el sistema.
- **fnm:** Fast Node Manager, es una herramienta que permite instalar y gestionar distintas versiones de Node.js de forma rápida.
- **brew:** Homebrew, es un gestor de paquetes para macOS que permite instalar y gestionar aplicaciones y utilidades.
- **chocolatey:** Es un gestor de paquetes para Windows que permite instalar y gestionar aplicaciones y utilidades.

3.1 Instalación de fnm:

Para instalar fnm se puede utilizar el instalador oficial de fnm, para ello se recomienda seguir los siguientes pasos:

1. Descargar el instalador de fnm desde la página oficial: fnm
2. Ejecutar el instalador y seguir las instrucciones.
3. Verificar la instalación ejecutando los siguientes comandos en la terminal:

```
fnm -v                                                    ①
```

① Muestra la versión de fnm instalada.

Para poder cambiar la versión de Node.js se puede utilizar el siguiente comando en la terminal:

```
fnm use <version>                                         ①
```

① Cambia la versión de Node.js a la versión especificada.

Ejemplo:

```
fnm use 14.17.0
```

El comando anterior cambiará la versión de Node.js a la versión 14.17.0.

Si queremos utilizar una versión más reciente de Node.js se puede utilizar el siguiente comando en la terminal:

```
fnm install latest
```

El comando anterior instalará la última versión de Node.js.

Si queremos utilizar una versión en particular como la 20.0.0 se puede utilizar el siguiente comando en la terminal:

```
fnm install 20.0.0
```

El comando anterior instalará la versión 20.0.0 de Node.js.

Si queremos cambiar de versión de Node.js en un proyecto en particular se puede utilizar el siguiente comando en la terminal:

```
fnm use <version> --global false                                    ①
```

① Cambia la versión de Node.js a la versión especificada en el proyecto actual.

Ejemplo:

```
fnm use 20.0.0 --global false
```

El comando anterior cambiará la versión de Node.js a la versión 20.0.0 en el proyecto actual.

De esa forma se puede cambiar de versión de Node.js en un proyecto en particular.

3.2 Instalación de TypeScript:

Para instalar TypeScript se puede utilizar el gestor de paquetes npm, para ello se recomienda seguir los siguientes pasos:

1. Instalar TypeScript de forma global ejecutando el siguiente comando en la terminal:

```
npm install -g typescript
```

El comando anterior instalará TypeScript de forma global en el sistema, lo que permitirá utilizar el compilador de TypeScript (tsc) desde cualquier directorio.

2. Verificar la instalación ejecutando el siguiente comando en la terminal:

```
tsc -v                                                                    ①
```

① Muestra la versión de TypeScript instalada.

3.3 Configuración de TypeScript:

Para configurar TypeScript en un proyecto se debe crear un archivo de configuración llamado **tsconfig.json**, que contiene las opciones de configuración del compilador de TypeScript.

Para crear un archivo de configuración de TypeScript se puede utilizar el siguiente comando en la terminal:

```
tsc --init
```

El comando anterior creará un archivo **tsconfig.json** con la configuración por defecto del compilador de TypeScript.

3.4 Compilación de TypeScript:

Para compilar un archivo TypeScript se puede utilizar el compilador de TypeScript (tsc), que convierte el código TypeScript a JavaScript.

Para compilar un archivo TypeScript se puede utilizar el siguiente comando en la terminal:

```
tsc archivo.ts
```

El comando anterior compilará el archivo **archivo.ts** y generará un archivo **archivo.js** con el código JavaScript resultante.

3.5 Ejemplo Práctico

En este ejemplo se creará un archivo TypeScript llamado **saludo.ts** que contiene una función que imprime un saludo en la consola.

3.5.1 Creación del Archivo TypeScript:

Para crear el archivo TypeScript se puede utilizar un editor de texto como Visual Studio Code, Sublime Text, Atom, etc. En este caso se utilizará Visual Studio Code.

1. Crear un archivo **saludo.ts** con el siguiente contenido:

```
function saludar(nombre: string) {
    console.log(`¡Hola, ${nombre}!`);
}

saludar("Mundo");
```

El archivo **saludo.ts** contiene una función **saludar** que recibe un parámetro **nombre** de tipo **string** y lo imprime en la consola.

3.5.2 Compilación del Archivo TypeScript:

Para compilar el archivo TypeScript se puede utilizar el compilador de TypeScript (tsc), que convierte el código TypeScript a JavaScript.

1. Compilar el archivo **saludo.ts** ejecutando el siguiente comando en la terminal:

```
tsc saludo.ts
```

El comando anterior compilará el archivo **saludo.ts** y generará un archivo **saludo.js** con el código JavaScript resultante.

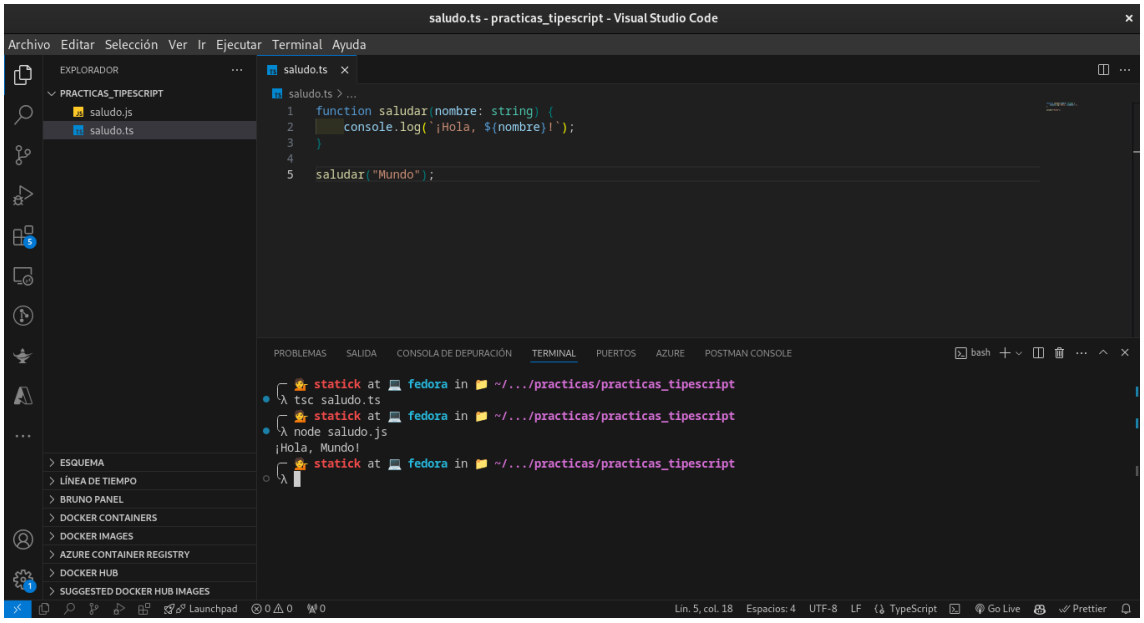
3.5.3 Ejecución del Archivo JavaScript:

Para ejecutar el archivo JavaScript generado se puede utilizar un intérprete de JavaScript como Node.js.

1. Ejecutar el archivo **saludo.js** ejecutando el siguiente comando en la terminal:

```
node saludo.js
```

El comando anterior ejecutará el archivo **saludo.js** y mostrará el saludo en la consola.



3.6 Conclusiones

- TypeScript es un lenguaje de programación desarrollado por Microsoft que añade tipado estático a JavaScript.
- TypeScript es un superconjunto de JavaScript, lo que significa que todo código JavaScript válido es código TypeScript válido.
- TypeScript permite definir interfaces, enums y decoradores, entre otras características.
- Para instalar TypeScript se puede utilizar el gestor de paquetes npm.
- Para configurar TypeScript en un proyecto se debe crear un archivo **tsconfig.json** con las opciones de configuración del compilador de TypeScript.
- Para compilar un archivo TypeScript se puede utilizar el compilador de TypeScript (tsc).
- TypeScript es una herramienta poderosa para el desarrollo de aplicaciones web, ya que permite detectar errores en tiempo de compilación y mejorar la calidad del código.

3.7 Referencias

- TypeScript
- Node.js
- npm
- Deno
- Bun
- fnm
- Visual Studio Code

4 Fundamentos de TypeScript

En este capítulo se abordarán los fundamentos de TypeScript, como los tipos primitivos, tipos personalizados e Inferencia de tipos en TypeScript.

4.1 Tipos primitivos (string, number, boolean, any, void, null, undefined).

4.1.1 string

```
let nombre: string = 'Juan';
```

En el caso de las cadenas de texto, TypeScript permite el uso de comillas simples o dobles.

4.1.2 number

```
let edad: number = 30;
```

En el caso de los números, TypeScript permite el uso de números enteros y decimales.

4.1.3 boolean

```
let esMayorDeEdad: boolean = true;
```

En el caso de los booleanos, TypeScript permite el uso de **true** y **false**.

4.1.4 any

```
let variable: any = 'Hola';
variable = 10;
variable = true;
```

En el caso de la variable **any**, TypeScript permite asignar cualquier tipo de valor.

4.1.5 void

```
function saludar(): void {
    console.log('Hola mundo');
}
```

En el caso de las funciones, TypeScript permite el uso de **void** para indicar que la función no retorna ningún valor.

4.1.6 null

```
let variable: null = null;
```

En el caso de las variables, TypeScript permite el uso de **null**.

4.1.7 undefined

```
let variable: undefined = undefined;
```

En el caso de las variables, TypeScript permite el uso de **undefined**.

4.2 Tipos personalizados (enums, tuples)

En esta sección se abordarán los tipos personalizados en TypeScript, como los **enums** y **tuples**.

4.2.1 enums

```
enum Color {
    Rojo,
    Verde,
    Azul
}

let color: Color = Color.Verde;
```

En el caso de los **enums**, TypeScript permite definir un conjunto de valores.

4.2.2 tuples

```
let persona: [string, number] = ['Juan', 30];
```

En el caso de las **tuples**, TypeScript permite definir un conjunto de valores con tipos específicos.

4.3 Inferencia de tipos

En TypeScript, la Inferencia de tipos permite asignar un tipo de dato a una variable sin necesidad de especificarlo.

```
let nombre = 'Juan';
```

En el caso anterior, TypeScript infiere que la variable **nombre** es de tipo **string**.

4.4 Ejemplos Prácticos.

4.4.1 Ejemplo 1: Tipos primitivos

1. Crear un archivo **tipos-primitivos.ts**.
2. Agregar el siguiente código:

```
let nombre: string = 'Juan';
let edad: number = 30;
let esMayorDeEdad: boolean = true;
let variable: any = 'Hola';
variable = 10;
variable = true;
let variableNula: null = null;
let variableIndefinida: undefined = undefined;

console.log(nombre);
console.log(edad);
console.log(esMayorDeEdad);
console.log(variable);
console.log(variableNula);
console.log(variableIndefinida);
```

En el caso anterior, se declararon variables de tipo **string**, **number**, **boolean**, **any**, **null** y **undefined**.

3. Ejecutar el archivo **tipos-primitivos.ts**.

```
tsc tipos-primitivos.ts
node tipos-primitivos.js
```

En el caso anterior, se compila el archivo **tipos-primitivos.ts** y se ejecuta el archivo **tipos-primitivos.js**.

4.4.2 Ejemplo 2: Tipos personalizados

1. Crear un archivo **tipos-personalizados.ts**.
2. Agregar el siguiente código:

```
let persona1: [string, number] = ['Juan', 30];

console.log(persona1);
```

En el caso anterior, se declaró un **enum** llamado **Color** y una **tuple** llamada **persona**.

3. Ejecutar el archivo **tipos-personalizados.ts**.

```
tsc tipos-personalizados.ts
node tipos-personalizados.js
```

En el caso anterior, se compila el archivo **tipos-personalizados.ts** y se ejecuta el archivo **tipos-personalizados.js**.

4.4.3 Ejemplo 3: Inferencia de tipos

1. Crear un archivo **inferencia-tipos.ts**.
2. Agregar el siguiente código:

```
let nombre = 'Juan';

console.log(nombre);
```

En el caso anterior, se declaró una variable **nombre** sin especificar el tipo de dato.

3. Ejecutar el archivo **inferencia-tipos.ts**.

```
tsc inferencia-tipos.ts
node inferencia-tipos.js
```

En el caso anterior, se compila el archivo **inferencia-tipos.ts** y se ejecuta el archivo **inferencia-tipos.js**.

4.5 Ejemplos

1. Crear un archivo **tipos-primitivos.ts** que contenga las siguientes variables:

- Una variable de tipo **string** llamada **nombre** con el valor **'Juan'**.
- Una variable de tipo **number** llamada **edad** con el valor **30**.
- Una variable de tipo **boolean** llamada **esMayorDeEdad** con el valor **true**.
- Una variable de tipo **any** llamada **variable** con el valor **'Hola'**.
- Una variable de tipo **any** llamada **variable** con el valor **10**.
- Una variable de tipo **any** llamada **variable** con el valor **true**.
- Una variable de tipo **null** llamada **variableNula** con el valor **null**.
- Una variable de tipo **undefined** llamada **variableIndefinida** con el valor **undefined**.
- Crear un archivo **tipos-primitivos.ts**.

```
let nombre: string = 'Juan';
let edad: number = 30;
let esMayorDeEdad: boolean = true;
let variable: any = 'Hola';
variable = 10;
variable = true;
let variableNula: null = null;
let variableIndefinida: undefined = undefined;

console.log(nombre); // Juan
console.log(edad); // 30
console.log(esMayorDeEdad); // true
console.log(variable); // true
console.log(variableNula); // null
console.log(variableIndefinida); // undefined
```

- Ejecutar el archivo **tipos-primitivos.ts**.

```
tsc tipos-primitivos.ts
node tipos-primitivos.js
```

2. Crear un archivo **tipos-personalizados.ts** que contenga las siguientes variables:

- Un **enum** llamado **Color** con los valores **Rojo**, **Verde** y **Azul**.
- Una **tuple** llamada **persona** con los valores **'Juan'** y **30**.
- Crear un archivo **tipos-personalizados.ts**.

```
enum Color {
    Rojo,
    Verde,
    Azul
}

let color: Color = Color.Verde;

console.log(color); // 1

let persona: [string, number] = ['Juan', 30];

console.log(persona); // ['Juan', 30]
```

- Ejecutar el archivo **tipos-personalizados.ts**.

```
tsc tipos-personalizados.ts
node tipos-personalizados.js
```

En el caso anterior, se declaró un **enum** llamado **Color** y una **tuple** llamada **persona**.

3. Crear un archivo **inferencia-tipos.ts** que contenga la siguiente variable:

- Una variable llamada **nombre** con el valor **'Juan'**.
- Crear un archivo **inferencia-tipos.ts**.

```
let nombre = 'Juan';

console.log(nombre); // Juan
```

- Ejecutar el archivo **inferencia-tipos.ts**.

```
tsc inferencia-tipos.ts
node inferencia-tipos.js
```

En el caso anterior, se declaró una variable **nombre** sin especificar el tipo de dato.

4.6 Reto

1. Crear un archivo **tipos-primitivos.ts** que contenga las siguientes variables:

- Una variable de tipo **string** llamada **nombre** con el valor **'Juan'**.
- Una variable de tipo **number** llamada **edad** con el valor **30**.
- Una variable de tipo **boolean** llamada **esMayorDeEdad** con el valor **true**.
- Una variable de tipo **any** llamada **variable** con el valor **'Hola'**.
- Una variable de tipo **any** llamada **variable** con el valor **10**.

- Una variable de tipo **any** llamada **variable** con el valor **true**.
- Una variable de tipo **null** llamada **variableNula** con el valor **null**.
- Una variable de tipo **undefined** llamada **variableIndefinida** con el valor **undefined**.

Ver código

- Crear un archivo **tipos-primitivos.ts**.

```
let nombre: string = 'Juan';
let edad: number = 30;
let esMayorDeEdad: boolean = true;
let variable: any = 'Hola';
variable = 10;
variable = true;
let variableNula: null = null;
let variableIndefinida: undefined = undefined;

console.log(nombre); // Juan
console.log(edad); // 30
console.log(esMayorDeEdad); // true
console.log(variable); // true
console.log(variableNula); // null
console.log(variableIndefinida); // undefined
```

- Ejecutar el archivo **tipos-primitivos.ts**.

```
tsc tipos-primitivos.ts
node tipos-primitivos.js
```

4.7 Conclusiones

En este capítulo se abordaron los fundamentos de TypeScript, como los tipos primitivos, tipos personalizados e Inferencia de tipos en TypeScript.

4.8 Enlaces de interés

- Fundamentos de TypeScript
- Tipos primitivos en TypeScript
- Tipos personalizados en TypeScript
- Inferencia de tipos en TypeScript

5 Interfaces y Tipos Avanzados

En este capítulo analizaremos las interfaces y los tipos avanzados de TypeScript, asi tambien como los tipos literales y los tipos de unión.

5.1 Interfaces: definición y uso.

Las interfaces son una forma de definir la estructura de un objeto en TypeScript. Se pueden definir interfaces para objetos, funciones, clases y arreglos.

```
interface Person {
  name: string;
  age: number;
}

const person: Person = {
  name: 'John',
  age: 30
};

console.log(person); // { name: 'John', age: 30 }
```

En el ejemplo anterior, definimos una interfaz **Person** que tiene dos propiedades: **name** y **age**. Luego, creamos un objeto **person** que cumple con la estructura de la interfaz **Person**.

5.2 Tipos literales y tipos de unión

Los tipos literales son una forma de definir un tipo que solo puede tener un conjunto específico de valores. Por ejemplo, podemos definir un tipo literal para representar los días de la semana.

```
type Day = 'Monday' | 'Tuesday' | 'Wednesday' | 'Thursday' | 'Friday' | 'Saturday' | 'Sun

const day: Day = 'Monday';

console.log(day); // Monday
```

En el ejemplo anterior, definimos un tipo **Day** que solo puede tener los valores **Monday**, **Tuesday**, **Wednesday**, **Thursday**, **Friday**, **Saturday** o **Sunday**. Luego, creamos una variable **day** de tipo **Day** y le asignamos el valor **Monday**.

5.3 Tipos de unión

Los tipos de unión son una forma de combinar varios tipos en uno solo. Por ejemplo, podemos definir un tipo de unión para representar un número o una cadena.

```
type NumberOrString = number | string;

const value1: NumberOrString = 10;
const value2: NumberOrString = 'Hello';

console.log(value1); // 10
console.log(value2); // Hello
```

En el ejemplo anterior, definimos un tipo **NumberOrString** que puede ser un número o una cadena. Luego, creamos dos variables **value1** y **value2** de tipo **NumberOrString** y les asignamos un número y una cadena, respectivamente.

5.4 Tipos genéricos

Los tipos genéricos son una forma de definir tipos que pueden ser parametrizados con otros tipos. Por ejemplo, podemos definir una función genérica que toma un tipo **T** como argumento y devuelve un arreglo de ese tipo.

```
function toArray<T>(value: T): T[] {
  return [value];
}

const array1 = toArray(10);
const array2 = toArray('Hello');

console.log(array1); // [10]
console.log(array2); // ['Hello']
```

En el ejemplo anterior, definimos una función genérica **toArray** que toma un tipo **T** como argumento y devuelve un arreglo de ese tipo. Luego, creamos dos arreglos **array1** y **array2** llamando a la función **toArray** con un número y una cadena, respectivamente.

5.5 Tipos avanzados

Los tipos avanzados son una forma de definir tipos más complejos en TypeScript. Algunos tipos avanzados incluyen tipos condicionales, tipos inferidos, tipos de intersección y tipos de unión.

```
type Person = {
  name: string;
  age: number;
};

type Employee = {
  name: string;
  salary: number;
};

type Manager = Person & Employee;

const manager: Manager = {
  name: 'John',
  age: 30,
  salary: 50000
};

console.log(manager); // { name: 'John', age: 30, salary: 50000 }
```

En el ejemplo anterior, definimos tres tipos: **Person**, **Employee** y **Manager**. El tipo **Manager** es una intersección de los tipos **Person** y **Employee**, lo que significa que tiene todas las propiedades de ambos tipos. Luego, creamos un objeto **manager** que cumple con la estructura del tipo **Manager**.

5.6 Ejemplos Prácticos.

5.6.1 Ejemplo 1: Definir una interfaz para un objeto.

```
interface Animal {
  name: string;
  age: number;
}
```

En el ejemplo anterior, definimos una interfaz **Animal** que tiene dos propiedades: **name** y **age**.

5.6.2 Ejemplo 2: Definir un tipo literal para representar los meses del año.

```
type Month = 'January' | 'February' | 'March' | 'April' | 'May' | 'June' | 'July' | 'Augu

const month: Month = 'January';

console.log(month); // January
```

En el ejemplo anterior, definimos un tipo **Month** que solo puede tener los valores **January**, **February**, **March**, **April**, **May**, **June**, **July**, **August**, **September**, **October**, **November** o **December**.

5.6.3 Ejemplo 3: Definir un tipo de unión para representar un número o una cadena.

```
type NumberOrString = number | string;

const value1: NumberOrString = 10;

console.log(value1); // 10
```

En el ejemplo anterior, definimos un tipo **NumberOrString** que puede ser un número o una cadena.

5.6.4 Ejemplo 4: Definir una función genérica que toma un tipo T como argumento y devuelve un arreglo de ese tipo.

```
function toArray<T>(value: T): T[] {
  return [value];
}

const array1 = toArray(10);

console.log(array1); // [10]

const array2 = toArray('Hello');

console.log(array2); // ['Hello']

const array3 = toArray({ name: 'John', age: 30 });

console.log(array3); // [{ name: 'John', age: 30 }]
```

```
const array4 = toArray([1, 2, 3]);

console.log(array4); // [[1, 2, 3]]

const array5 = toArray(true);

console.log(array5); // [true]
```

En el ejemplo anterior, definimos una función genérica **toArray** que toma un tipo **T** como argumento y devuelve un arreglo de ese tipo.

5.6.5 Ejemplo 5: Definir un tipo de intersección para combinar dos tipos.

```
type Animal = {
  name: string;
  age: number;
};

type Pet = {
  name: string;
  breed: string;
};

type Dog = Animal & Pet;

const dog: Dog = {
  name: 'Buddy',
  age: 5,
  breed: 'Labrador'
};

console.log(dog); // { name: 'Buddy', age: 5, breed: 'Labrador' }
```

En el ejemplo anterior, definimos tres tipos: **Animal**, **Pet** y **Dog**. El tipo **Dog** es una intersección de los tipos **Animal** y **Pet**, lo que significa que tiene todas las propiedades de ambos tipos.

6 Reto

1. Define una interfaz **Product** con las siguientes propiedades:
    - **name**: string
    - **price**: number
    - **quantity**: number
2. Define un tipo literal **Category** para representar las categorías de productos:
    - **'Electronics'**
    - **'Clothing'**
    - **'Books'**
    - **'Food'**
3. Define un tipo de unión **ProductOrCategory** para representar un producto o una categoría.
4. Define una función genérica **createProduct** que toma un tipo **T** como argumento y devuelve un objeto de tipo **Product**.
5. Define un tipo de intersección **ProductWithCategory** para combinar un producto y una categoría.

6.1 Conclusiones

En este capítulo aprendimos sobre las interfaces y los tipos avanzados de TypeScript, asi tambien como los tipos literales y los tipos de unión. Estos conceptos nos permiten definir tipos más complejos y reutilizables en nuestros programas.

6.2 Referencias

- Interfaces
- Tipos literales
- Tipos de unión
- Tipos genéricos
- Tipos avanzados

7 Funciones en TypeScript.

Las funciones en TypeScript son muy similares a las funciones en JavaScript. Para crear una función en TypeScript, se utiliza la palabra clave **function** seguida del nombre de la función y los parámetros que recibe. A continuación, se muestra un ejemplo de una función en TypeScript:

```
function sumar(a: number, b: number): number {
    return a + b; // Devuelve la suma de los dos números.
}
```

En el ejemplo anterior, se define una función llamada **sumar** que recibe dos parámetros de tipo **number** y devuelve un valor de tipo **number**. La función suma los dos números y devuelve el resultado.

7.1 Parámetros opcionales y valores por defecto.

En TypeScript, se pueden definir parámetros opcionales y valores por defecto en una función. Para definir un parámetro opcional, se utiliza el signo de interrogación (?) después del nombre del parámetro. Para definir un valor por defecto, se utiliza el operador de asignación (=) seguido del valor por defecto. A continuación, se muestra un ejemplo de una función con parámetros opcionales y valores por defecto:

```
function saludar(nombre: string, mensaje: string = "Hola", hora?: string): string {
    if (hora) {
        return `${mensaje}, ${nombre}! Son las ${hora}.`;
    } else {
        return `${mensaje}, ${nombre}!`;
    }
}

let saludo1 = saludar("Juan"); // saludo1 = "Hola, Juan!"
```

En el ejemplo anterior, se define una función llamada **saludar** que recibe tres parámetros: **nombre**, **mensaje** y **hora**. El parámetro **mensaje** tiene un valor por defecto de “Hola” y el parámetro **hora** es opcional. La función devuelve un mensaje personalizado dependiendo de los parámetros que recibe.

7.2 Funciones de flecha.

En TypeScript, se pueden definir funciones de flecha utilizando la sintaxis **(parámetros) => expresión**. Las funciones de flecha son una forma más concisa de definir funciones y permiten omitir la palabra clave **function** y las llaves {}. A continuación, se muestra un ejemplo de una función de flecha en TypeScript:

```
let sumar = (a: number, b: number): number => a + b;

let resultado = sumar(10, 20); // resultado = 30
```

En el ejemplo anterior, se define una función de flecha llamada **sumar** que recibe dos parámetros de tipo **number** y devuelve un valor de tipo **number**. La función suma los dos números y devuelve el resultado.

7.3 Funciones como parámetros.

En el ejemplo anterior, se define una función de flecha llamada **sumar** que recibe dos parámetros de tipo **number** y devuelve un valor de tipo **number**. La función suma los dos números y devuelve el resultado.

7.4 Funciones como parámetros.

En TypeScript, se pueden pasar funciones como parámetros a otras funciones. Esto permite crear funciones de orden superior que toman funciones como argumentos y devuelven funciones como resultado. A continuación, se muestra un ejemplo de una función que recibe una función como parámetro:

```
function aplicarFuncion(funcion: (a: number, b: number) => number, a: number, b: number):
    return funcion(a, b);
}

let resultado = aplicarFuncion((a, b) => a + b, 10, 20); // resultado = 30
```

En el ejemplo anterior, se define una función llamada **aplicarFuncion** que recibe tres parámetros: **funcion**, **a** y **b**. El parámetro **funcion** es una función que recibe dos parámetros de tipo **number** y devuelve un valor de tipo **number**. La función **aplicarFuncion** llama a la función pasada como argumento con los parámetros **a** y **b** y devuelve el resultado.

7.5 Funciones genéricas.

En TypeScript, se pueden definir funciones genéricas que aceptan un tipo de dato como parámetro. Esto permite reutilizar la misma función con diferentes tipos de datos. A continuación, se muestra un ejemplo de una función genérica en TypeScript:

```
function imprimir<T>(valor: T): void {
    console.log(valor);
}

imprimir<number>(10); // Imprime 10
imprimir<string>("Hola"); // Imprime Hola
```

En el ejemplo anterior, se define una función genérica llamada **imprimir** que acepta un parámetro de tipo **T** y no devuelve ningún valor. La función imprime el valor pasado como argumento utilizando la función **console.log**. Se pueden llamar a la función **imprimir** con diferentes tipos de datos, como **number** y **string**.

8 Reto

Crea una función en TypeScript que reciba un arreglo de números y devuelva la suma de los números pares. La función debe tener la siguiente firma:

```
function sumarPares(numeros: number[]): number {
    // Implementa la función aquí.
}

let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]; // Suma de los números pares: 30
let resultado = sumarPares(numeros); // resultado = 30

console.log(`Suma de los números pares: ${resultado}`);
```

Ver solución

```
function sumarPares(numeros: number[]): number {
    return numeros.filter(numero => numero % 2 === 0).reduce((a, b) => a + b, 0);
}

let numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]; // Suma de los números pares: 30
let resultado = sumarPares(numeros); // resultado = 30

console.log(`Suma de los números pares: ${resultado}`);
```

9 Conclusiones.

En TypeScript, se pueden definir funciones de la misma forma que en JavaScript, pero con la ventaja de poder especificar los tipos de los parámetros y el tipo de retorno. Además, TypeScript permite definir parámetros opcionales, valores por defecto, funciones de flecha, funciones como parámetros y funciones genéricas. Estas características hacen que TypeScript sea un lenguaje más seguro y robusto para el desarrollo de aplicaciones.

En este tutorial, aprendiste cómo definir funciones en TypeScript y cómo utilizar las diferentes características que ofrece el lenguaje. Ahora estás listo para crear funciones más complejas y reutilizables en tus proyectos de TypeScript. ¡Sigue practicando y mejorando tus habilidades como desarrollador de TypeScript!

10 Clases y Herencia

En este capítulo se explicará cómo se pueden crear clases y cómo se pueden heredar de otras clases.

10.1 Clases y objetos.

Una clase es un modelo que define un conjunto de atributos y métodos que tendrán los objetos que se creen a partir de ella.

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

    constructor(nombre: string, edad: number) {
        this.nombre = nombre;
        this.edad = edad;
    }

    saludar() {
        console.log(`Hola, soy ${this.nombre} y tengo ${this.edad} años.`);
    }
}

let persona1 = new Persona('Juan', 30); // Se crea un objeto de la clase Persona
let persona2 = new Persona('Ana', 25); // Se crea otro objeto de la clase Persona

persona1.saludar(); // Hola, soy Juan y tengo 30 años.
persona2.saludar(); // Hola, soy Ana y tengo 25 años.
```

En el ejemplo anterior se ha creado una clase **Persona** que tiene dos atributos (**nombre** y **edad**) y un método (**saludar**). A partir de esta clase se han creado dos objetos (**persona1** y **persona2**) que tienen los mismos atributos y métodos.

10.2 Herencia

La herencia es un mecanismo que permite crear una nueva clase a partir de una clase existente. La nueva clase hereda los atributos y métodos de la clase existente y puede añadir nuevos atributos y métodos.

```
class Empleado extends Persona {
    salario: number;

    constructor(nombre: string, edad: number, salario: number) {
        super(nombre, edad);
        this.salario = salario;
    }

    trabajar() {
        console.log(`${this.nombre} está trabajando.`);
    }
}

let empleado1 = new Empleado('Pedro', 35, 2000);

empleado1.saludar(); // Hola, soy Pedro y tengo 35 años.
empleado1.trabajar(); // Pedro está trabajando.
```

En el ejemplo anterior se ha creado una clase **Empleado** que hereda de la clase **Persona**. La clase **Empleado** tiene un atributo adicional (**salario**) y un método adicional (**trabajar**).

10.3 Métodos estáticos

Los métodos estáticos son métodos que se pueden llamar sin necesidad de crear un objeto de la clase.

```
class Calculadora {
    static sumar(a: number, b: number) {
        return a + b;
    }

    static restar(a: number, b: number) {
        return a - b;
    }
}

console.log(Calculadora.sumar(5, 3)); // 8
console.log(Calculadora.restar(5, 3)); // 2
```

En el ejemplo anterior se ha creado una clase **Calculadora** con dos métodos estáticos (**sumar** y **restar**). Estos métodos se pueden llamar directamente a través de la clase sin necesidad de crear un objeto.

10.4 Modificadores de acceso (public, private, protected)

En este apartado se explicarán los modificadores de acceso que se pueden utilizar en TypeScript.

10.4.1 public

El modificador **public** indica que los atributos y métodos son accesibles desde cualquier parte del código.

```
class Coche {
    public marca: string;
    public modelo: string;

    constructor(marca: string, modelo: string) {
        this.marca = marca;
        this.modelo = modelo;
    }

    mostrar() {
        console.log(`Marca: ${this.marca}, Modelo: ${this.modelo}`);
    }
}

let coche = new Coche('Seat', 'Ibiza');
console.log(coche.marca); // Seat
console.log(coche.modelo); // Ibiza
coche.mostrar(); // Marca: Seat, Modelo: Ibiza
```

En el ejemplo anterior se ha creado una clase **Coche** con dos atributos (**marca** y **modelo**) y un método (**mostrar**). Los atributos y métodos son públicos, por lo que se pueden acceder desde cualquier parte del código.

10.4.2 private

El modificador **private** indica que los atributos y métodos son accesibles únicamente desde la propia clase.

```
class Coche {
    private marca: string;
    private modelo: string;

    constructor(marca: string, modelo: string) {
        this.marca = marca;
        this.modelo = modelo;
    }
```

```
    mostrar() {
        console.log(`Marca: ${this.marca}, Modelo: ${this.modelo}`);
    }
}

let coche = new Coche('Seat', 'Ibiza');
console.log(coche.marca); // Error
console.log(coche.modelo); // Error
coche.mostrar(); // Marca: Seat, Modelo: Ibiza
```

En el ejemplo anterior se ha creado una clase **Coche** con dos atributos (**marca** y **modelo**) y un método (**mostrar**). Los atributos y métodos son privados, por lo que no se pueden acceder desde fuera de la clase.

10.4.3 protected

El modificador **protected** indica que los atributos y métodos son accesibles desde la propia clase y desde las clases hijas.

```
class Vehiculo {
    protected marca: string;
    protected modelo: string;

    constructor(marca: string, modelo: string) {
        this.marca = marca;
        this.modelo = modelo;
    }

    mostrar() {
        console.log(`Marca: ${this.marca}, Modelo: ${this.modelo}`);
    }
}

class Coche extends Vehiculo {
    constructor(marca: string, modelo: string) {
        super(marca, modelo);
    }

    mostrar() {
        console.log(`Coche: Marca: ${this.marca}, Modelo: ${this.modelo}`);
    }
}

let coche = new Coche('Seat', 'Ibiza');
coche.mostrar(); // Coche: Marca: Seat, Modelo: Ibiza
```

En el ejemplo anterior se ha creado una clase **Vehiculo** con dos atributos (**marca** y **modelo**) y un método (**mostrar**). Los atributos y métodos son protegidos, por lo que se pueden acceder desde la propia clase y desde las clases hijas.

10.5 Getters y setters

Los getters y setters son métodos especiales que se utilizan para acceder y modificar los atributos de una clase.

```
class Persona {
    private _nombre: string;

    constructor(nombre: string) {
        this._nombre = nombre;
    }

    get nombre() {
        return this._nombre;
    }

    set nombre(nombre: string) {
        this._nombre = nombre;
    }
}

let persona = new Persona('Juan');
console.log(persona.nombre); // Juan
persona.nombre = 'Pedro';
console.log(persona.nombre); // Pedro
```

En el ejemplo anterior se ha creado una clase **Persona** con un atributo privado (**_nombre**) y un getter y un setter para acceder y modificar el atributo.

10.6 Herencia y polimorfismo

El polimorfismo es un concepto que permite que un objeto de una clase hija se comporte como un objeto de la clase padre.

```
class Vehiculo {
    protected marca: string;
    protected modelo: string;

    constructor(marca: string, modelo: string) {
        this.marca = marca;
        this.modelo = modelo;
```

```
    }

    mostrar() {
        console.log(`Marca: ${this.marca}, Modelo: ${this.modelo}`);
    }
}

class Coche extends Vehiculo {
    constructor(marca: string, modelo: string) {
        super(marca, modelo);
    }

    mostrar() {
        console.log(`Coche: Marca: ${this.marca}, Modelo: ${this.modelo}`);
    }
}

let vehiculo: Vehiculo = new Coche('Seat', 'Ibiza');
vehiculo.mostrar(); // Coche: Marca: Seat, Modelo: Ibiza
```

En el ejemplo anterior se ha creado una clase **Vehiculo** con un método **mostrar** y una clase **Coche** que hereda de la clase **Vehiculo** y sobrescribe el método **mostrar**. Se crea un objeto de la clase **Coche** y se asigna a una variable de tipo **Vehiculo**. Al llamar al método **mostrar** se ejecuta el método de la clase **Coche**.

10.7 Abstract classes

Las clases abstractas son clases que no se pueden instanciar directamente, sino que se utilizan como base para crear otras clases.

```
abstract class Figura {
    abstract area(): number;
}

class Circulo extends Figura {
    private radio: number;

    constructor(radio: number) {
        super();
        this.radio = radio;
    }

    area(): number {
        return Math.PI * this.radio * this.radio;
    }
}
```

```
let circulo = new Circulo(5);
console.log(circulo.area()); // 78.53981633974483
```

En el ejemplo anterior se ha creado una clase abstracta **Figura** con un método abstracto **area**. Se ha creado una clase **Circulo** que hereda de la clase **Figura** y sobrescribe el método **area**. Al crear un objeto de la clase **Circulo** se puede llamar al método **area**.

10.8 Interfaces

Las interfaces son un mecanismo que permite definir la estructura que deben tener los objetos.

```
interface Figura {
    area(): number;
}

class Circulo implements Figura {
    private radio: number;

    constructor(radio: number) {
        this.radio = radio;
    }

    area(): number {
        return Math.PI * this.radio * this.radio;
    }
}

let circulo = new Circulo(5);
console.log(circulo.area()); // 78.53981633974483
```

En el ejemplo anterior se ha creado una interfaz **Figura** con un método **area**. Se ha creado una clase **Circulo** que implementa la interfaz **Figura** y define el método **area**. Al crear un objeto de la clase **Circulo** se puede llamar al método **area**.

11 Reto

Crear una clase **Animal** con los siguientes atributos y métodos:

- Atributos:
  - nombre: string
  - edad: number
- Métodos:
  - constructor(nombre: string, edad: number): constructor que recibe el nombre y la edad del animal.
  - emitirSonido(): método que muestra un mensaje con el sonido del animal.

Crear una clase **Perro** que herede de la clase **Animal** con los siguientes atributos y métodos:

- Atributos:
  - raza: string
- Métodos:
  - constructor(nombre: string, edad: number, raza: string): constructor que recibe el nombre, la edad y la raza del perro.
  - emitirSonido(): método que muestra un mensaje con el sonido del perro.

Crear un objeto de la clase **Perro** y llamar al método **emitirSonido**.

Pista

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

    constructor(nombre: string, edad: number) {
        this.nombre = nombre;
        this.edad = edad;
    }

    emitirSonido() {
        console.log('Sonido del animal');
    }
}

class Perro extends Animal {
```

```
    raza: string;

    constructor(nombre: string, edad: number, raza: string) {
        super(nombre, edad);
        this.raza = raza;
    }

    emitirSonido() {
        console.log('Guau guau');
    }
}

let perro = new Perro('Toby', 5, 'Labrador');
perro.emitirSonido(); // Guau guau
```

11.1 Conclusiones

En este capítulo se ha explicado cómo se pueden crear clases y cómo se pueden heredar de otras clases. También se han explicado los modificadores de acceso, los métodos estáticos, los getters y setters, las clases abstractas y las interfaces.

Part II

Unidad 2: Profundización en TypeScript

12 Modularización y Espacios de Nombres

En este capitulo vamos a aprender acerca de la modularización y los espacios de nombres en TypeScript.

12.1 Modularización

La modularización es una técnica de programación que consiste en dividir un programa en módulos o partes más pequeñas. Esto permite que el código sea más fácil de mantener y de entender.

En TypeScript, podemos modularizar nuestro código utilizando los siguientes mecanismos:

- Módulos: nos permiten dividir nuestro código en archivos separados y reutilizarlo en otros archivos.
- Namespaces: nos permiten agrupar nuestro código en espacios de nombres para evitar conflictos de nombres.

13 Módulos: import y export

Los módulos en TypeScript nos permiten dividir nuestro código en archivos separados y reutilizarlo en otros archivos. Para exportar un módulo, utilizamos la palabra clave export, y para importar un módulo, utilizamos la palabra clave import.

13.1 Export

Para exportar un módulo en TypeScript, utilizamos la palabra clave export seguida del nombre de la variable, función o clase que queremos exportar.

```
// modulo.ts
export const PI = 3.1416;
export function suma(a: number, b: number): number {
  return a + b;
}

export class Persona {
  constructor(public nombre: string, public edad: number) {}
}
```

En el ejemplo anterior, exportamos una constante PI, una función suma y una clase Persona.

13.2 Import

Para importar un módulo en TypeScript, utilizamos la palabra clave import seguida del nombre del módulo que queremos importar.

```
// main.ts
import { PI, suma, Persona } from './modulo';

console.log(PI); // 3.1416
console.log(suma(2, 3)); // 5

const persona = new Persona('Juan', 30);
console.log(persona); // Persona { nombre: 'Juan', edad: 30 }
```

En el ejemplo anterior, importamos la constante PI, la función suma y la clase Persona del módulo modulo.ts.

13.3 Export por defecto

También podemos exportar un módulo por defecto en TypeScript utilizando la palabra clave default.

```
// modulo.ts
export default function saludar(nombre: string): string {
  return `Hola, ${nombre}!`;
}
```

En el ejemplo anterior, exportamos la función saludar por defecto.

```
// main.ts
import saludar from './modulo';

console.log(saludar('Juan')); // Hola, Juan!
```

En el ejemplo anterior, importamos la función saludar del módulo modulo.ts utilizando la palabra clave default.

14 Namespaces: definición y uso

Los namespaces en TypeScript nos permiten agrupar nuestro código en espacios de nombres para evitar conflictos de nombres. Para definir un namespace, utilizamos la palabra clave namespace seguida del nombre del namespace.

```
// namespace.ts
namespace Geometria {
  export const PI = 3.1416;
  export function areaCirculo(radio: number): number {
    return PI * radio ** 2;
  }
}
```

En el ejemplo anterior, definimos un namespace Geometria que contiene una constante PI y una función areaCirculo.

Para acceder a los miembros de un namespace, utilizamos la notación de punto.

```
// main.ts
console.log(Geometria.PI); // 3.1416
console.log(Geometria.areaCirculo(2)); // 12.5664
```

En el ejemplo anterior, accedemos a la constante PI y a la función areaCirculo del namespace Geometria utilizando la notación de punto.

15 Reto

Crea un módulo llamado operaciones.ts que contenga las siguientes funciones:

- suma(a: number, b: number): number
- resta(a: number, b: number): number
- multiplicacion(a: number, b: number): number
- division(a: number, b: number): number

Exporta las funciones y luego importalas en un archivo main.ts para probarlas.

Solución

```
// operaciones.ts
export function suma(a: number, b: number): number {
  return a + b;
}

export function resta(a: number, b: number): number {
  return a - b;
}

export function multiplicacion(a: number, b: number): number {
  return a * b;
}

export function division(a: number, b: number): number {
  return a / b;
}
```

```
// main.ts
import { suma, resta, multiplicacion, division } from './operaciones';

console.log(suma(2, 3)); // 5
console.log(resta(5, 3)); // 2
console.log(multiplicacion(2, 3)); // 6
console.log(division(6, 3)); // 2
```

16 Conclusión

En este capítulo aprendimos acerca de la modularización y los espacios de nombres en TypeScript. Los módulos nos permiten dividir nuestro código en archivos separados y reutilizarlo en otros archivos, mientras que los namespaces nos permiten agrupar nuestro código en espacios de nombres para evitar conflictos de nombres.

17 Decoradores

En este capítulo aprenderemos sobre los decoradores en TypeScript. Los decoradores son una característica experimental de JavaScript que permite agregar funcionalidades a clases y sus miembros. Los decoradores se utilizan para modificar o extender la funcionalidad de una clase o método sin modificar su código fuente.

Los decoradores se utilizan para agregar metadatos a clases y sus miembros. Los decoradores se pueden aplicar a clases, métodos, propiedades y parámetros de métodos. Los decoradores se utilizan para extender la funcionalidad de una clase o método sin modificar su código fuente.

```
@decorator
class MyClass {
    @decorator
    myMethod() {
        // code
    }
}
```

En el ejemplo anterior, **(decorator?)** es un decorador que se aplica a la clase **MyClass** y al método **myMethod**. Los decoradores se pueden aplicar a clases, métodos, propiedades y parámetros de métodos.

Los decoradores se utilizan para extender la funcionalidad de una clase o método sin modificar su código fuente. Los decoradores se utilizan para agregar metadatos a clases y sus miembros.

En TypeScript, los decoradores se definen como funciones que toman un argumento y devuelven una función que se aplica a la clase o miembro de la clase. Los decoradores se aplican a clases y miembros de la clase utilizando la sintaxis **(decorator?)**.

18 Introducción a los decoradores

En TypeScript, los decoradores se definen como funciones que toman un argumento y devuelven una función que se aplica a la clase o miembro de la clase. Los decoradores se aplican a clases y miembros de la clase utilizando la sintaxis **(decorator?)**.

Los decoradores se utilizan para extender la funcionalidad de una clase o método sin modificar su código fuente. Los decoradores se utilizan para agregar metadatos a clases y sus miembros.

Los decoradores se pueden aplicar a clases, métodos, propiedades y parámetros de métodos. Los decoradores se utilizan para extender la funcionalidad de una clase o método sin modificar su código fuente.

19 Decoradores de clases

Los decoradores de clases se utilizan para extender la funcionalidad de una clase sin modificar su código fuente. Los decoradores de clases se aplican a clases utilizando la sintaxis **(decorator?)**.

Los decoradores de clases se definen como funciones que toman un argumento y devuelven una función que se aplica a la clase. Los decoradores de clases se utilizan para agregar metadatos a clases.

```
function classDecorator<T extends { new (...args: any[]): {} }>(constructor: T) {
    return class extends constructor {
        newProperty = "new property";
        hello = "override";
    };
}

@classDecorator
class Greeter {
    property = "property";
    hello: string;
    constructor(m: string) {
        this.hello = m;
    }
}

console.log(new Greeter("world")); // Greeter { newProperty: 'new property', hello: 'over
```

En el ejemplo anterior, **classDecorator** es un decorador de clase que se aplica a la clase **Greeter**. El decorador de clase **classDecorator** agrega una nueva propiedad **newProperty** a la clase **Greeter**.

20 Decoradores de clases, métodos y propiedades

Los decoradores de clases, métodos y propiedades se utilizan para extender la funcionalidad de una clase, método o propiedad sin modificar su código fuente. Los decoradores de clases, métodos y propiedades se aplican a clases, métodos y propiedades utilizando la sintaxis **(decorator?)**.

```
function classDecorator<T extends { new (...args: any[]): {} }>(constructor: T) {
    return class extends constructor {
        newProperty = "new property";
        hello = "override";
    };
}

function methodDecorator(target: any, propertyKey: string, descriptor: PropertyDescriptor
    console.log("Method Decorator called");
}

function propertyDecorator(target: any, propertyKey: string) {
    console.log("Property Decorator called");
}

@classDecorator
class Greeter {
    @propertyDecorator
    property = "property";
    hello: string;
    constructor(m: string) {
        this.hello = m;
    }

    @methodDecorator
    greet() {
        return "Hello, " + this.hello;
    }
}

console.log(new Greeter("world").greet()); // Method Decorator called
```

En el ejemplo anterior, **classDecorator** es un decorador de clase que se aplica a la clase

Greeter. El decorador de clase **classDecorator** agrega una nueva propiedad **newProperty** a la clase **Greeter**.

El decorador de propiedad **propertyDecorator** se aplica a la propiedad **property** de la clase **Greeter**. El decorador de método **methodDecorator** se aplica al método **greet** de la clase **Greeter**.

21 Decoradores de métodos y propiedades

Los decoradores de métodos y propiedades se utilizan para extender la funcionalidad de un método o propiedad sin modificar su código fuente. Los decoradores de métodos y propiedades se aplican a métodos y propiedades utilizando la sintaxis **(decorator?)**.

```
function methodDecorator(target: any, propertyKey: string, descriptor: PropertyDescriptor
    console.log("Method Decorator called");
}

function propertyDecorator(target: any, propertyKey: string) {
    console.log("Property Decorator called");
}

class Greeter {
    @propertyDecorator
    property = "property";
    hello: string;
    constructor(m: string) {
        this.hello = m;
    }

    @methodDecorator
    greet() {
        return "Hello, " + this.hello;
    }
}

console.log(new Greeter("world").greet()); // Method Decorator called
```

En el ejemplo anterior, el decorador de propiedad **propertyDecorator** se aplica a la propiedad **property** de la clase **Greeter**. El decorador de método **methodDecorator** se aplica al método **greet** de la clase **Greeter**.

22 Decoradores de parámetros

Los decoradores de parámetros se utilizan para extender la funcionalidad de un parámetro de método sin modificar su código fuente. Los decoradores de parámetros se aplican a parámetros de métodos utilizando la sintaxis **(decorator?)**.

```
function parameterDecorator(target: any, propertyKey: string, parameterIndex: number) {
    console.log("Parameter Decorator called");
}

class Greeter {
    greet(@parameterDecorator name: string) {
        return "Hello, " + name;
    }
}

console.log(new Greeter().greet("world")); // Parameter Decorator called
```

En el ejemplo anterior, el decorador de parámetro **parameterDecorator** se aplica al parámetro **name** del método **greet** de la clase **Greeter**.

23 Decoradores de fábrica

Los decoradores de fábrica son funciones que devuelven funciones que se aplican a clases, métodos, propiedades y parámetros de métodos. Los decoradores de fábrica se utilizan para extender la funcionalidad de una clase o método sin modificar su código fuente.

```
function factoryDecorator(value: string) {
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("Factory Decorator called with value: " + value);
    };
}

class Greeter {
    @factoryDecorator("value")
    property = "property";
    hello: string;
    constructor(m: string) {
        this.hello = m;
    }
}

console.log(new Greeter("world")); // Factory Decorator called with value: value
```

En el ejemplo anterior, **factoryDecorator** es un decorador de fábrica que devuelve una función que se aplica a la propiedad **property** de la clase **Greeter**. El decorador de fábrica **factoryDecorator** agrega metadatos a la propiedad **property** de la clase **Greeter**.

24 Decoradores de métodos de acceso

Los decoradores de métodos de acceso se utilizan para extender la funcionalidad de un método de acceso sin modificar su código fuente. Los decoradores de métodos de acceso se aplican a métodos de acceso utilizando la sintaxis **(decorator?)**.

```
function accessorDecorator(target: any, propertyKey: string, descriptor: PropertyDescript
    console.log("Accessor Decorator called");
}

class Greeter {
    private _name: string = "";

    @accessorDecorator
    get name(): string {
        return this._name;
    }

    set name(value: string) {
        this._name = value;
    }
}

const greeter = new Greeter();
greeter.name = "world";
console.log(greeter.name); // Accessor Decorator called
```

En el ejemplo anterior, el decorador de método de acceso **accessorDecorator** se aplica a los métodos de acceso **get** y **set** de la propiedad **name** de la clase **Greeter**.

25 Decoradores de métodos estáticos

Los decoradores de métodos estáticos se utilizan para extender la funcionalidad de un método estático sin modificar su código fuente. Los decoradores de métodos estáticos se aplican a métodos estáticos utilizando la sintaxis **(decorator?)**.

```
function staticMethodDecorator(target: any, propertyKey: string, descriptor: PropertyDesc
    console.log("Static Method Decorator called");
}

class Greeter {
    static name: string = "world";

    @staticMethodDecorator
    static greet() {
        return "Hello, " + Greeter.name;
    }
}

console.log(Greeter.greet()); // Static Method Decorator called
```

En el ejemplo anterior, el decorador de método estático **staticMethodDecorator** se aplica al método estático **greet** de la clase **Greeter**.

26 Reto

1. Crear un decorador de clase que agregue una propiedad **newProperty** a la clase **MyClass**.
2. Crear un decorador de método que se aplique al método **myMethod** de la clase **MyClass**.
3. Crear un decorador de propiedad que se aplique a la propiedad **myProperty** de la clase **MyClass**.
4. Crear un decorador de parámetro que se aplique al parámetro **myParameter** del método **myMethod** de la clase **MyClass**.

Solución

```
function classDecorator<T extends { new (...args: any[]): {} }>(constructor: T) {
    return class extends constructor {
        newProperty = "new property";
    };
}

function methodDecorator(target: any, propertyKey: string, descriptor: PropertyDescriptor
    console.log("Method Decorator called");
}

function propertyDecorator(target: any, propertyKey: string) {
    console.log("Property Decorator called");
}

function parameterDecorator(target: any, propertyKey: string, parameterIndex: number) {
    console.log("Parameter Decorator called");
}

@classDecorator
class MyClass {
    @propertyDecorator
    myProperty = "my property";

    constructor() {}

    @methodDecorator
    myMethod(@parameterDecorator myParameter: string) {
        return "Hello, " + myParameter;
    }
}
```

```
console.log(new MyClass().myMethod("world")); // Method Decorator called
```

27 Conclusiones

Los decoradores son una característica experimental de JavaScript que permite agregar funcionalidades a clases y sus miembros. Los decoradores se utilizan para modificar o extender la funcionalidad de una clase o método sin modificar su código fuente.

Los decoradores se utilizan para agregar metadatos a clases y sus miembros. Los decoradores se pueden aplicar a clases, métodos, propiedades y parámetros de métodos. Los decoradores se utilizan para extender la funcionalidad de una clase o método sin modificar su código fuente.

En TypeScript, los decoradores se definen como funciones que toman un argumento y devuelven una función que se aplica a la clase o miembro de la clase. Los decoradores se aplican a clases y miembros de la clase utilizando la sintaxis **(decorator?)**.

Los decoradores se utilizan para extender la funcionalidad de una clase o método sin modificar su código fuente. Los decoradores se utilizan para agregar metadatos a clases y sus miembros.

Los decoradores se pueden aplicar a clases, métodos, propiedades y parámetros de métodos. Los decoradores se utilizan para extender la funcionalidad de una clase o método sin modificar su código fuente.

28 Manejo de Errores

En este capítulo vamos a ver cómo manejar errores en TypeScript.

29 Introducción

En TypeScript, como en cualquier otro lenguaje de programación, los errores pueden ocurrir en cualquier momento. Estos errores pueden ser de varios tipos, como errores de sintaxis, errores de tiempo de ejecución, errores de lógica, etc.

En este capítulo vamos a ver cómo manejar errores en TypeScript.

30 Errores en TypeScript

En TypeScript, los errores pueden ocurrir en cualquier momento. Estos errores pueden ser de varios tipos, como errores de sintaxis, errores de tiempo de ejecución, errores de lógica, etc.

En TypeScript, los errores se pueden clasificar en dos categorías:

1. Errores de sintaxis: Estos errores ocurren cuando el código no sigue las reglas de sintaxis del lenguaje de programación. Por ejemplo, si olvidamos cerrar una llave o un paréntesis, TypeScript generará un error de sintaxis.
2. Errores de tiempo de ejecución: Estos errores ocurren cuando el código se ejecuta y algo inesperado sucede. Por ejemplo, si intentamos dividir un número por cero, TypeScript generará un error de tiempo de ejecución.

31 Manejo de errores en TypeScript

En TypeScript, podemos manejar errores de varias maneras. Algunas de las formas más comunes de manejar errores en TypeScript son:

1. Usando la declaración **try...catch**: La declaración **try...catch** nos permite capturar errores y manejarlos de manera controlada.
2. Usando la declaración **throw**: La declaración **throw** nos permite lanzar un error manualmente.
3. Usando la declaración **finally**: La declaración **finally** nos permite ejecutar un bloque de código después de que se haya completado el bloque **try** o **catch**.

32 Ejemplo de manejo de errores en TypeScript

Veamos un ejemplo de cómo manejar errores en TypeScript usando la declaración **try...catch**:

```
try {
    // Código que puede lanzar un error
    let x = 1 / 0;
} catch (error) {
    // Manejar el error
    console.log('Ocurrió un error:', error);
}
```

En este ejemplo, estamos intentando dividir un número por cero, lo cual generará un error de tiempo de ejecución. Usamos la declaración **try...catch** para capturar el error y manejarlo de manera controlada.

33 Lanzar errores manualmente en TypeScript

En TypeScript, podemos lanzar errores manualmente usando la declaración **throw**. Por ejemplo:

```
try {
    // Código que puede lanzar un error
    throw new Error('Este es un error manual');
} catch (error) {
    // Manejar el error
    console.log('Ocurrió un error:', error);
}
```

En este ejemplo, estamos lanzando un error manualmente usando la declaración **throw**. Luego, usamos la declaración **try...catch** para capturar el error y manejarlo de manera controlada.

34 Bloque finally en TypeScript

En TypeScript, podemos usar la declaración **finally** para ejecutar un bloque de código después de que se haya completado el bloque **try** o **catch**. Por ejemplo:

```
try {
    // Código que puede lanzar un error
    let x = 1 / 0;
} catch (error) {
    // Manejar el error
    console.log('Ocurrió un error:', error);
} finally {
    // Código que se ejecutará siempre
    console.log('Este código se ejecutará siempre');
}
```

En este ejemplo, estamos intentando dividir un número por cero, lo cual generará un error de tiempo de ejecución. Usamos la declaración **try...catch** para capturar el error y manejarlo de manera controlada. Luego, usamos la declaración **finally** para ejecutar un bloque de código después de que se haya completado el bloque **try** o **catch**.

35 Reto

Escribe un programa en TypeScript que maneje errores usando la declaración **try...catch**. El programa debe intentar dividir un número por cero y capturar el error usando la declaración **try...catch**. Luego, debe lanzar un error manualmente usando la declaración **throw** y capturar el error usando la declaración **try...catch**. Finalmente, debe ejecutar un bloque de código después de que se haya completado el bloque **try** o **catch** usando la declaración **finally**.

Solución

```
try {
    function divideByZero() {
        let x = 1 / 0;
    }
    let x = 1 / 0;
} catch (error) {
    // Manejar el error
    console.log('Ocurrió un error:', error);
}

try {
    function throwError() {
        throw new Error('Este es un error manual');
    }
    throwError();
} catch (error) {
    // Manejar el error
    console.log('Ocurrió un error:', error);
}

try {
    function divideByZero() {
        let x = 1 / 0;
    }
    divideByZero();
} catch (error) {
    // Manejar el error
    console.log('Ocurrió un error:', error);
} finally {
    // Código que se ejecutará siempre
    console.log('Este código se ejecutará siempre');
}
```

36 Conclusión

En este capítulo, aprendimos cómo manejar errores en TypeScript. Vimos cómo usar la declaración **try...catch** para capturar errores y manejarlos de manera controlada. También vimos cómo lanzar errores manualmente usando la declaración **throw** y cómo ejecutar un bloque de código después de que se haya completado el bloque **try** o **catch** usando la declaración **finally**.

37 Programación Asíncrona en TypeScript

En este capítulo vamos a ver cómo trabajar con código asíncrono en TypeScript. Vamos a ver cómo trabajar con promesas y cómo podemos usar **async** y **await** para trabajar con código asíncrono de una forma más sencilla.

38 Promesas y async/await

39 Promesas

Las promesas son un objeto que representa la terminación o el fracaso de una operación asíncrona. Una promesa es un objeto devuelto al cuál se adjuntan funciones callback, en lugar de pasar callbacks a una función.

```
const promesa = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('Se ha resuelto la promesa');
    }, 1000);
});

promesa.then((mensaje) => {
    console.log(mensaje);
});
```

En el ejemplo anterior, creamos una promesa que se resuelve después de 1 segundo. Cuando la promesa se resuelve, se ejecuta la función **then** que recibe el mensaje que se ha pasado al método **resolve**.

40 async/await

async y **await** son una forma de trabajar con promesas de una forma más sencilla. **async** se usa para declarar una función asíncrona y **await** se usa para esperar a que una promesa se resuelva.

```
const promesa = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('Se ha resuelto la promesa');
    }, 1000);
});

async function miFuncion() {
    const mensaje = await promesa;
    console.log(mensaje);
}

miFuncion(); // Se ejecuta la función asíncrona
```

En el ejemplo anterior, creamos una función asíncrona que espera a que la promesa se resuelva. Cuando la promesa se resuelve, se almacena el mensaje en la variable **mensaje** y se imprime por consola.

41 Tipado de funciones asíncronas

Podemos tipar las funciones asíncronas de la misma forma que tipamos las funciones normales. Podemos indicar el tipo de los parámetros y el tipo de retorno de la función.

```
const promesa = new Promise<string>((resolve, reject) => {
    setTimeout(() => {
        resolve('Se ha resuelto la promesa');
    }, 1000);
});

async function miFuncion(parametro: string): Promise<string> {
    const mensaje = await promesa;
    return mensaje + parametro;
}

miFuncion('Hola').then((mensaje) => {
    console.log(mensaje);
});
```

En el ejemplo anterior, la función **miFuncion** recibe un parámetro de tipo **string** y devuelve una promesa de tipo **string**.

42 Captura de errores

Podemos capturar los errores que se producen en una función asíncrona con un bloque **try/catch**.

```
const promesa = new Promise<string>((resolve, reject) => {
    setTimeout(() => {
        reject('Se ha producido un error');
    }, 1000);
});

async function miFuncion(parametro: string): Promise<string> {
    try {
        const mensaje = await promesa;
        return mensaje + parametro;
    } catch (error) {
        console.error(error);
        return error;
    }
}

miFuncion('Hola').then((mensaje) => {
    console.log(mensaje);
});
```

En el ejemplo anterior, la promesa se rechaza después de 1 segundo. Cuando la promesa se rechaza, se captura el error en el bloque **catch** y se imprime por consola.

43 Ejemplo

En el siguiente ejemplo vamos a ver cómo podemos trabajar con código asíncrono en TypeScript. Vamos a crear una función asíncrona que recibe un número y devuelve el doble de ese número después de 1 segundo.

```
const promesa = new Promise<number>((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 1000);
});

async function doble(numero: number): Promise<number> {
    try {
        const resultado = await promesa;
        return resultado * 2;
    } catch (error) {
        console.error(error);
        return error;
    }
}

doble(5).then((resultado) => {
    console.log(resultado);
});
```

En el ejemplo anterior, la función **doble** recibe un número y devuelve el doble de ese número después de 1 segundo. Cuando la promesa se resuelve, se imprime el resultado por consola.

44 Reto

Crea una función asíncrona que reciba un número y devuelva el cuadrado de ese número después de 1 segundo. Llama a la función con el número 5 y muestra el resultado por consola.

Solución

```
const promesa = new Promise<number>((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 1000);
});

async function cuadrado(numero: number): Promise<number> {
    try {
        const resultado = await promesa;
        return resultado * resultado;
    } catch (error) {
        console.error(error);
        return error;
    }
}

cuadrado(5).then((resultado) => {
    console.log(resultado);
});
```

45 Conclusión

En este capítulo hemos visto cómo trabajar con código asíncrono en TypeScript. Hemos visto cómo trabajar con promesas y cómo podemos usar **async** y **await** para trabajar con código asíncrono de una forma más sencilla. En el siguiente capítulo vamos a ver cómo podemos trabajar con módulos en TypeScript.

46 Event Binding en Angular

En esta unidad vamos a conocer cómo podemos reaccionar a eventos del usuario en Angular. Para ello, vamos a ver cómo podemos utilizar el event binding en Angular.

Para entender este tema vamos a crear un nuevo proyecto Angular. Para ello, abrimos una terminal y ejecutamos el siguiente comando:

```
ng new event-binding
```

Una vez creado el proyecto, nos movemos a la carpeta del proyecto:

```
cd event-binding
```

Y abrimos el proyecto en Visual Studio Code:

```
code .
```

46.1 Event Binding

El event binding en Angular nos permite reaccionar a eventos del usuario, como por ejemplo, un click, un doble click, un hover, etc. Para ello, vamos a ver un ejemplo sencillo.

Vamos a crear un nuevo componente llamado **contador**:

```
ng g c contador
```

Una vez creado el componente, nos movemos al archivo **contador.component.html** y escribimos el siguiente código:

```
<h1>Contador</h1>

<p>{{ contador }}</p>

<button>Incrementar</button>
<button>Decrementar</button>
```

En este código, tenemos un título, un párrafo que muestra el valor del contador y dos botones, uno para incrementar el contador y otro para decrementarlo.

Para poder reaccionar a los eventos de click de los botones, vamos a utilizar el event binding. Para ello, nos movemos al archivo **contador.component.html** y escribimos el siguiente código:

```
<h1>Contador</h1>

<p>{{ contador }}</p>

<button (click)="incrementar()">Incrementar</button>
<button (click)="decrementar()">Decrementar</button>
```

En este código, estamos utilizando el event binding para reaccionar al evento de click de los botones. Cuando se haga click en el botón de incrementar, se va a ejecutar el método **incrementar()** y cuando se haga click en el botón de decrementar, se va a ejecutar el método **decrementar()**.

Ahora, nos movemos al archivo **contador.component.ts** y escribimos el siguiente código:

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

@Component({
  selector: 'app-contador',
  standalone: true,
  imports: [],
  templateUrl: './contador.component.html',
  styleUrl: './contador.component.css'
})
export class ContadorComponent {

    contador: number = 0;

    incrementar() {
        this.contador++;
    }

    decrementar() {
        this.contador--;
    }
}
```

En este código, estamos creando una variable **contador** que va a almacenar el valor del contador y dos métodos **incrementar()** y **decrementar()** que se van a encargar de incrementar y decrementar el contador respectivamente.

> **Tip**
>
> No olvides agregar el componente **ContadorComponent** a la aplicación principal en el archivo **app.component.ts**.

```ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ContadorComponent } from './contador/contador.component';  ①

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    ContadorComponent                                                ②
  ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'event-binding';
}
```

① Importamos el componente **ContadorComponent**.
② Agregamos el componente **ContadorComponent** al arreglo de imports.

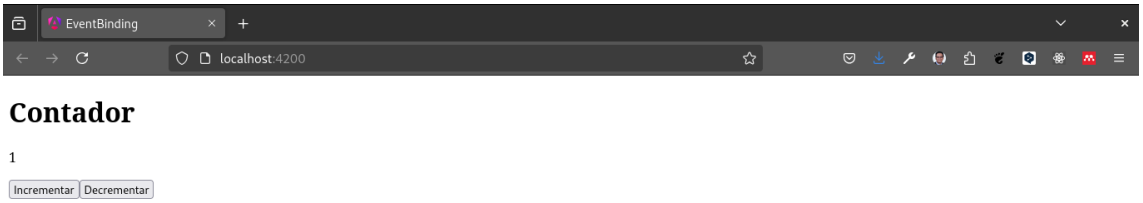
Tambien es necesario agregar el selector del componente **ContadorComponent** en el archivo **app.component.html**.

```html
<app-contador></app-contador>
```

Ahora, si ejecutamos la aplicación con el siguiente comando:

```
ng s -o
```

Podremos ver el contador en la pantalla y si hacemos click en los botones de incrementar y decrementar, el contador se va a incrementar y decrementar respectivamente.


EventBinding
localhost:4200
Contador
1
Incrementar
Decrementar

47 Reto

Crea un nuevo proyecto Angular y crea un nuevo componente llamado **calculadora**.

```
ng new calculadora
cd calculadora
ng g c calculadora
```

En este componente, crea una calculadora sencilla con dos inputs para ingresar los números y cuatro botones para realizar las operaciones de suma, resta, multiplicación y división. Utiliza el event binding para reaccionar a los eventos de click de los botones y mostrar el resultado de la operación en un párrafo.

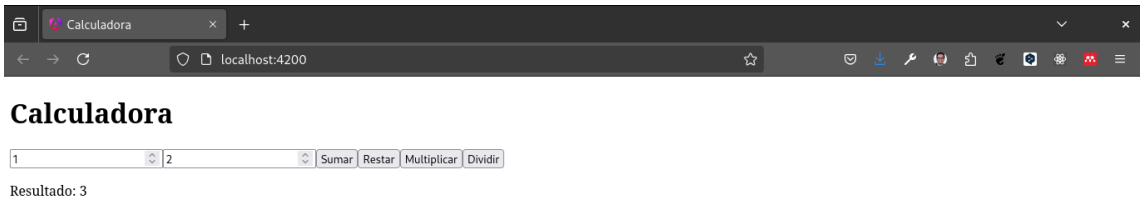
Posible solución

```
<h1>Calculadora</h1>

<input type="number" [(ngModel)]="numero1" placeholder="Número 1">
<input type="number" [(ngModel)]="numero2" placeholder="Número 2">

<button (click)="sumar()">Sumar</button>
<button (click)="restar()">Restar</button>
<button (click)="multiplicar()">Multiplicar</button>
<button (click)="dividir()">Dividir</button>

<p *ngIf="resultado !== undefined">Resultado: {{ resultado }}</p>
<p *ngIf="resultado === undefined">Realiza una operación</p>
```

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-calculadora',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './calculadora.component.html',
  styleUrls: ['./calculadora.component.css']
})
export class CalculadoraComponent {
  numero1: number | undefined;
  numero2: number | undefined;
```

```
  resultado: number | undefined;

  sumar() {
    this.resultado = (this.numero1 ?? 0) + (this.numero2 ?? 0);
  }

  restar() {
    this.resultado = (this.numero1 ?? 0) - (this.numero2 ?? 0);
  }

  multiplicar() {
    this.resultado = (this.numero1 ?? 0) * (this.numero2 ?? 0);
  }

  dividir() {
    this.resultado = (this.numero1 ?? 0) / (this.numero2 ?? 0);
  }
}
```

Si todo salio bien, deberías ver algo como esto:



48 Conclusión

En esta unidad aprendimos cómo podemos reaccionar a eventos del usuario en Angular utilizando el event binding. Vimos cómo podemos reaccionar a eventos de click de los botones y cómo podemos ejecutar métodos en el componente cuando se produce un evento. En el siguiente tema, vamos a ver cómo podemos reaccionar a eventos del usuario utilizando el event binding en Angular.

Part III

Unidad 3: TypeScript con Angular

49 Introducción a Angular

Figure 49.1: Angular

49.1 ¿Qué es Angular?

Angular es un framework de desarrollo de aplicaciones web desarrollado por Google. Angular es un framework de código abierto y gratuito que permite a los desarrolladores crear aplicaciones web de una sola página (SPA) de alta calidad y alto rendimiento.

50 Historia y evolución de Angular

Angular se lanzó por primera vez en 2010 por Google. Angular es un marco de desarrollo de aplicaciones web de código abierto y gratuito que permite a los desarrolladores crear aplicaciones web de una sola página (SPA) de alta calidad y alto rendimiento.

51 Características de Angular

Angular es un framework de desarrollo de aplicaciones web que ofrece una serie de características y funcionalidades que lo hacen único y poderoso. Algunas de las características más importantes de Angular son:

- **Componentes**: Angular es un framework basado en componentes, lo que significa que las aplicaciones web se construyen a partir de componentes reutilizables y modulares. Los componentes son bloques de construcción de una aplicación Angular y se pueden reutilizar en diferentes partes de la aplicación.
- **Directivas**: Angular ofrece una serie de directivas que permiten a los desarrolladores extender y modificar el comportamiento de los elementos HTML. Las directivas son atributos especiales que se pueden agregar a los elementos HTML para agregar funcionalidades adicionales.
- **Servicios**: Angular ofrece una serie de servicios que permiten a los desarrolladores compartir datos y funcionalidades entre diferentes partes de la aplicación. Los servicios son clases que se pueden inyectar en los componentes y otros servicios para compartir datos y funcionalidades.
- **Inyección de dependencias**: Angular utiliza un sistema de inyección de dependencias que permite a los desarrolladores inyectar dependencias en los componentes y servicios de la aplicación. Esto facilita la creación de aplicaciones modulares y reutilizables.
- **Rutas**: Angular ofrece un sistema de enrutamiento que permite a los desarrolladores crear aplicaciones de una sola página (SPA) con múltiples vistas. El sistema de enrutamiento de Angular permite a los desarrolladores definir rutas y vistas para diferentes partes de la aplicación.
- **Pipes**: Angular ofrece una serie de pipes que permiten a los desarrolladores transformar y formatear datos en las plantillas HTML. Los pipes son funciones que se pueden aplicar a los datos en las plantillas HTML para transformarlos y formatearlos de diferentes maneras.
- **Formularios**: Angular ofrece un sistema de formularios que permite a los desarrolladores crear formularios reactivos y dinámicos en las aplicaciones web. El sistema de formularios de Angular permite a los desarrolladores validar y gestionar los datos de los formularios de manera eficiente.
- **Testing**: Angular ofrece un sistema de pruebas que permite a los desarrolladores probar las aplicaciones web de manera eficiente. Angular proporciona herramientas y utilidades para escribir pruebas unitarias y de integración para las aplicaciones web.

52 Ventajas de Angular

Angular es un framework de desarrollo de aplicaciones web que ofrece una serie de ventajas y beneficios que lo hacen único y poderoso. Algunas de las ventajas más importantes de Angular son:

- **Productividad**: Angular es un framework de desarrollo de aplicaciones web que permite a los desarrolladores crear aplicaciones web de alta calidad y alto rendimiento de manera rápida y eficiente. Angular ofrece una serie de características y funcionalidades que facilitan el desarrollo de aplicaciones web.
- **Reutilización de código**: Angular es un framework basado en componentes, lo que significa que las aplicaciones web se construyen a partir de componentes reutilizables y modulares. Los componentes son bloques de construcción de una aplicación Angular y se pueden reutilizar en diferentes partes de la aplicación.
- **Mantenimiento**: Angular es un framework de desarrollo de aplicaciones web que facilita el mantenimiento de las aplicaciones web. Angular ofrece una serie de características y funcionalidades que facilitan la gestión y el mantenimiento de las aplicaciones web.
- **Escalabilidad**: Angular es un framework de desarrollo de aplicaciones web que permite a los desarrolladores crear aplicaciones web escalables y de alto rendimiento. Angular ofrece una serie de características y funcionalidades que facilitan la escalabilidad de las aplicaciones web.
- **Comunidad activa**: Angular es un framework de desarrollo de aplicaciones web que cuenta con una comunidad activa de desarrolladores y contribuidores. La comunidad de Angular es muy activa y ofrece soporte, documentación y recursos para los desarrolladores que trabajan con Angular.

53 Desventajas de Angular

Angular es un framework de desarrollo de aplicaciones web que ofrece una serie de ventajas y beneficios, pero también tiene algunas desventajas y limitaciones. Algunas de las desventajas más importantes de Angular son:

- **Curva de aprendizaje**: Angular es un framework de desarrollo de aplicaciones web que tiene una curva de aprendizaje empinada. Los desarrolladores que no están familiarizados con Angular pueden encontrar difícil aprender y dominar el framework.
- **Complejidad**: Angular es un framework de desarrollo de aplicaciones web que puede ser complejo y difícil de entender. Angular ofrece una serie de características y funcionalidades avanzadas que pueden ser difíciles de dominar para los desarrolladores principiantes.
- **Rendimiento**: Angular es un framework de desarrollo de aplicaciones web que puede tener problemas de rendimiento en aplicaciones web grandes y complejas. Angular ofrece una serie de características y funcionalidades que pueden afectar el rendimiento de las aplicaciones web.
- **Tamaño**: Angular es un framework de desarrollo de aplicaciones web que puede tener un tamaño grande y pesado. Angular ofrece una serie de características y funcionalidades que pueden aumentar el tamaño de las aplicaciones web.
- **Compatibilidad**: Angular es un framework de desarrollo de aplicaciones web que puede tener problemas de compatibilidad con otros frameworks y bibliotecas. Angular ofrece una serie de características y funcionalidades que pueden no ser compatibles con otros frameworks y bibliotecas.

54 Comparación con otros frameworks

Angular es un framework de desarrollo de aplicaciones web que ofrece una serie de características y funcionalidades que lo hacen único y poderoso. A continuación se muestra una comparación de Angular con otros frameworks de desarrollo de aplicaciones web populares:

- **React**: React es una biblioteca de JavaScript desarrollada por Facebook que se utiliza para crear interfaces de usuario interactivas y dinámicas. React es una biblioteca de JavaScript que se centra en la creación de componentes reutilizables y modulares. React es una biblioteca de JavaScript que se utiliza para crear aplicaciones web de una sola página (SPA) de alta calidad y alto rendimiento.
- **Vue**: Vue es un framework de desarrollo de aplicaciones web de código abierto y gratuito que se utiliza para crear aplicaciones web de una sola página (SPA) de alta calidad y alto rendimiento. Vue es un framework de desarrollo de aplicaciones web que se centra en la creación de componentes reutilizables y modulares. Vue es un framework de desarrollo de aplicaciones web que ofrece una serie de características y funcionalidades que facilitan el desarrollo de aplicaciones web.
- **Svelte**: Svelte es un framework de desarrollo de aplicaciones web de código abierto y gratuito que se utiliza para crear aplicaciones web de una sola página (SPA) de alta calidad y alto rendimiento. Svelte es un framework de desarrollo de aplicaciones web que se centra en la creación de componentes reutilizables y modulares. Svelte es un framework de desarrollo de aplicaciones web que ofrece una serie de características y funcionalidades que facilitan el desarrollo de aplicaciones web.

55 Configuración del entorno de desarrollo (Angular CLI)

Para comenzar a trabajar con Angular, necesitamos configurar un entorno de desarrollo. Angular CLI (Command Line Interface) es una herramienta de línea de comandos que nos permite crear, construir y probar aplicaciones Angular de manera rápida y eficiente.

Para instalar Angular CLI, necesitamos tener Node.js y npm instalados en nuestro sistema. Node.js es un entorno de ejecución de JavaScript que nos permite ejecutar JavaScript en el servidor. npm es un administrador de paquetes de Node.js que nos permite instalar y administrar paquetes de Node.js.

Para instalar Angular CLI, ejecutamos el siguiente comando en la terminal:

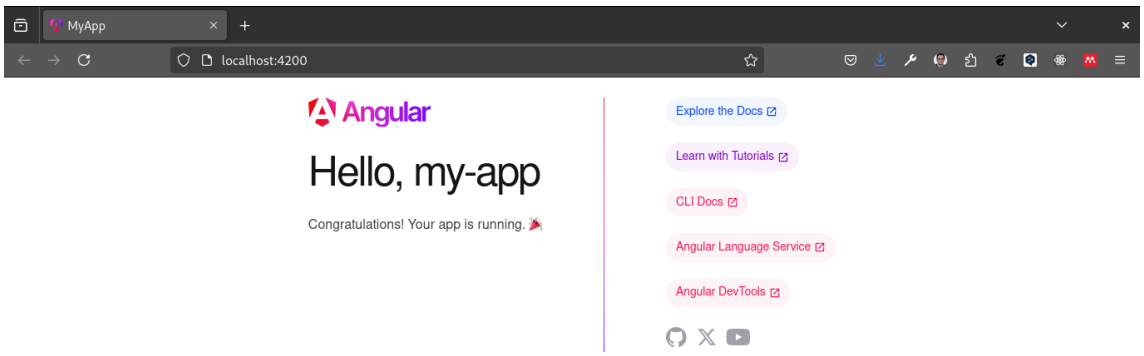
```
npm install -g @angular/cli
```

Una vez que Angular CLI se haya instalado correctamente, podemos crear un nuevo proyecto Angular ejecutando el siguiente comando en la terminal:

```
ng new my-app
```

Este comando creará un nuevo proyecto Angular llamado **my-app** en el directorio actual. Una vez que el proyecto se haya creado correctamente, podemos navegar al directorio del proyecto y ejecutar el siguiente comando para iniciar el servidor de desarrollo:

```
ng serve
```



Este comando iniciará el servidor de desarrollo de Angular en el puerto 4200. Podemos abrir un navegador web y navegar a http://localhost:4200 para ver nuestra aplicación Angular en acción.

56 Estructura de un proyecto Angular

Un proyecto Angular consta de varios archivos y directorios que definen la estructura de la aplicación. A continuación se muestra la estructura de un proyecto Angular típico:

- **e2e**: Este directorio contiene las pruebas de extremo a extremo de la aplicación.
- **node_modules**: Este directorio contiene los módulos de Node.js que se utilizan en la aplicación.
- **src**: Este directorio contiene el código fuente de la aplicación.
  - **app**: Este directorio contiene los componentes, servicios, directivas y pipes de la aplicación.
  - **assets**: Este directorio contiene los archivos estáticos de la aplicación, como imágenes, fuentes y estilos.
  - **environments**: Este directorio contiene los archivos de configuración de los entornos de la aplicación.
  - **index.html**: Este archivo es la página principal de la aplicación.
  - **main.ts**: Este archivo es el punto de entrada de la aplicación.
  - **styles.css**: Este archivo contiene los estilos globales de la aplicación.
- **angular.json**: Este archivo contiene la configuración de Angular CLI para el proyecto.

57 Hola Mundo en Angular

Vamos a modificar el archivo **app.component.ts** para mostrar un mensaje de "Hola Mundo" en nuestra aplicación Angular. Abre el archivo **app.component.ts** y modifícalo de la siguiente manera:

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  template: `
    <h1>Hello World!</h1> // <1>
  `,
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'my-app';
}
```

① Agregamos un mensaje de "Hola Mundo" en la plantilla HTML del componente.

Este código define un componente Angular llamado **AppComponent** que muestra un mensaje de "Hola Mundo" en la plantilla HTML. Ahora, si abrimos un navegador web y navegamos a http://localhost:4200, deberíamos ver el mensaje de "Hola Mundo" en nuestra aplicación Angular.

Otra opción sería modificar el archivo **app.component.html**, borramos el contenido y agregamos el siguiente código:

```
<h1>Hello World!</h1>
```

Dejamos el archivo **app.component.ts** de la siguiente manera:

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
```

```
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'my-app';
}
```

> **Tip**
>
> No olvides que puedes correr un servidor de desarrollo con el siguiente comando en la terminal.
>
> ```
> ng serve -o
> ```
>
> El flag **-o** abrirá automáticamente el navegador en la dirección http://localhost:4200.

Hello World!

> **Tip**
>
> Si estás trabajando con visual studio code, puedes instalar la extensión **Angular Language Service** para tener autocompletado y sugerencias en tu código.

58 Componentes

En este capítulo vamos a comprender como funcionan los componentes en Angular tomando en cuenta que cambian algunas cosas desde la versión 16, en este ejemplo utilizaremos la version 18.

Vamos a crear un nuevo proyecto de Angular utilizando el comando:

```
ng new angular-components
```

Luego de crear el proyecto, vamos a abrirlo en Visual Studio Code utilizando el comando:

```
cd angular-components
code .
```

58.1 Crear un componente

Para crear un componente en Angular, se puede hacer de dos formas, la primera es utilizando el comando:

```
ng generate component
```

o

```
ng g c
```

Otr opción que es la que recomiendo para entender los componentes es hacerlo de forma manual y para ello se debe crear una carpeta con el nombre del componente en la carpeta **src/app** y dentro de esta carpeta se deben crear los archivos **component.ts**, **component.html**, **component.css** y **component.spec.ts**.

> Tip
>
> Por convensión se acostumbra crear un directorio con el nombre del componente en minúsculas y dentro de este directorio se crean los archivos del componente. Y cada uno de los archivos del componente debe tener el mismo nombre del directorio, agregando la palabra **component** y la extensión correspondiente.

Para entender de mejor forma los componentes vamos a crear un componente llamado **header** y un componente llamado **footer**. El primero lo haremos de forma manual y el segundo lo haremos utilizando el comando **ng generate component**.

58.1.1 Crear el componente header de forma manual

Para crear el componente **header** de forma manual se deben seguir los siguientes pasos:

1. Crear la carpeta **header** en la carpeta **src/app**.
2. Dentro de la carpeta **header** crear los archivos **header.component.ts**, **header.component.html**, **header.component.css**.
3. En el archivo **header.component.ts** se debe agregar el siguiente código:

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

@Component({
  selector: 'app-header',
  standalone: true,
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.css']
})
export class HeaderComponent {
  title = 'Header';
}
```

> **Warning**
>
> A partir de la versión 16 se utiliza los componentes standalone, por lo que se debe agregar el atributo **standalone: true** en el decorador (**Component?**).

4. En el archivo **header.component.html** se debe agregar el siguiente código:

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

5. En el archivo **header.component.css** se debe agregar el siguiente código:

```
header {
  background-color: #333;
  color: white;
  text-align: center;
  padding: 10px;
}
```

6. Para utilizar el componente **header** en la aplicación se debe agregar el selector **app-header** en el archivo **app.component.html**:

```
<app-header></app-header>
```

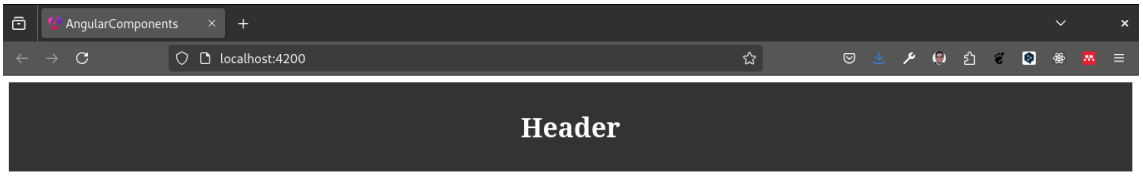
```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { HeaderComponent } from './header/header.component';                    ①

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    HeaderComponent                                                             ②
  ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'angular-components';
}
```

① Se importa el componente **HeaderComponent**.
② Se agrega el componente **HeaderComponent** en el arreglo de imports.

Si todo salio bien al correr el servidor de desarrollo con el comando:

```
ng s
```

Se debe ver el título **Header** en la parte superior de la página.



Ahora vamos a crear el componente **footer**.

58.1.2 Crear el componente footer

Para crear el componente **footer** de forma automática se debe utilizar el comando:

```
ng g c footer
```

Este comando crea la carpeta **footer** en la carpeta **src/app** y dentro de esta carpeta se crean los archivos **footer.component.ts**, **footer.component.html**, **footer.component.css** y **footer.component.spec.ts**.

Para utilizar el componente **footer** en la aplicación se debe agregar el selector **app-footer** en el archivo **app.component.html**:

```
<app-footer></app-footer>
```

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { HeaderComponent } from './header/header.component';
import { FooterComponent } from './footer/footer.component';                ①

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    HeaderComponent,
    FooterComponent                                                         ②
  ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'angular-components';
}
```

① Se importa el componente **FooterComponent**.
② Se agrega el componente **FooterComponent** en el arreglo de imports.

Modificamos un poco el archivo **footer.component.html** para que se vea de la siguiente forma:

```
<footer>
    <div class="container">
        <div class="row">
        <div class="col-md-12">
            <p class="text-center">© 2024 <a href="
            https://espe.edu.ec" target="_blank">Universidad de las Fuerzas Armadas ESPE<
```

```
        </div>
        </div>
    </div>
</footer>
```

Tambien el archivo **footer.component.css** para que se vea de la siguiente forma:

```
footer {
  background-color: #333;
  color: white;
  text-align: center;
  padding: 10px;
}
```

Si todo salio bien al correr el servidor de desarrollo con el comando:

```
ng s
```

Se debe ver el título **Footer** en la parte inferior de la página.

59 Reto

Crear un componente llamado **content** que tenga un título y un párrafo y utilizarlo en la aplicación.

Ver solución

1. Crear la carpeta **content** en la carpeta **src/app**.
2. Dentro de la carpeta **content** crear los archivos **content.component.ts**, **content.component.html**, **content.component.css**.
3. En el archivo **content.component.ts** se debe agregar el siguiente código:

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

@Component({
  selector: 'app-content',
  standalone: true,
  templateUrl: './content.component.html',
  styleUrls: ['./content.component.css']
})

export class ContentComponent {
  title = 'Content';
  paragraph = 'Lorem ipsum ';
}
```

4. En el archivo **content.component.html** se debe agregar el siguiente código:

```
<section>
  <h2>{{ title }}</h2>
  <p>{{ paragraph }}</p>
</section>
```

5. En el archivo **content.component.css** se debe agregar el siguiente código:

```
section {
  padding: 10px;
}
```

6. Para utilizar el componente **content** en la aplicación se debe agregar el selector **app-content** en el archivo **app.component.html**:

```
<app-content></app-content>
```

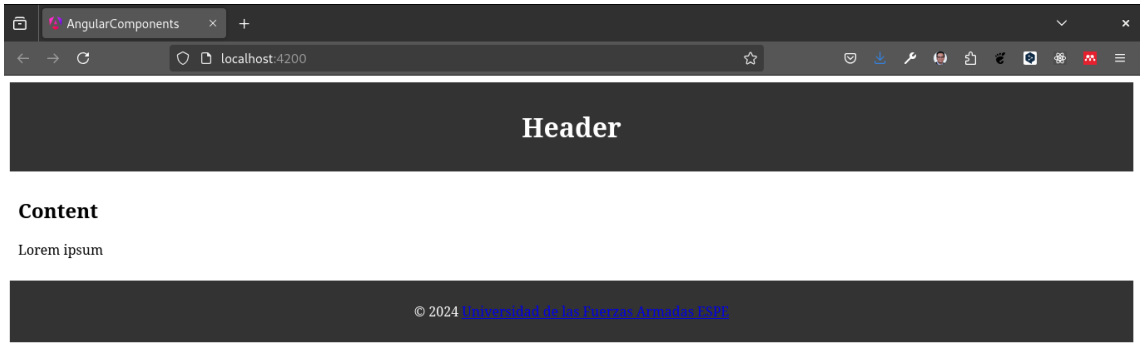
```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { HeaderComponent } from './header/header.component';
import { FooterComponent } from './footer/footer.component';
import { ContentComponent } from './content/content.component';          ①

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    HeaderComponent,
    FooterComponent,
    ContentComponent                                                     ②
  ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'angular-components';
}
```

① Se importa el componente **ContentComponent**.
② Se agrega el componente **ContentComponent** en el arreglo de imports.

Si todo salio bien al correr el servidor de desarrollo con el comando:

```
ng s
```

Se debe ver el título **Content** en la parte central de la página.


AngularComponents
localhost:4200
Header
Content
Lorem ipsum
© 2024 Universidad de las Fuerzas Armadas ESPE

60 Conclusión

En este capítulo aprendimos a crear componentes en Angular y a utilizarlos en la aplicación. En el siguiente capítulo vamos a aprender a utilizar los servicios en Angular.

61 Interpolación en Angular

La interpolación es una forma de mostrar datos en la vista de un componente. Se utiliza la sintaxis de doble llave **{{ }}** para mostrar datos en la vista.

Para este capítulo vamos a crear un nuevo proyecto de Angular. Para ello, abrimos una terminal y ejecutamos el siguiente comando:

```
ng new interpolacion
```

Nos ubicamos en el directorio del proyecto:

```
cd interpolacion
```

Y ejecutamos el servidor de desarrollo:

```
ng serve -o
```

61.1 Crear un componente

Vamos a crear un nuevo componente llamado **usuario**. Para ello, ejecutamos el siguiente comando en la terminal:

```
ng generate component usuario
```

61.2 Interpolación

Vamos a mostrar el nombre de un usuario en la vista del componente **usuario**. Para ello, abrimos el archivo **usuario.component.ts** y agregamos la siguiente propiedad:

```
nombre: string = 'Diego';
```

Ahora, abrimos el archivo **usuario.component.html** y agregamos el siguiente código:

```
<p>Nombre: {{ nombre }}</p>
```

Guardamos los cambios y observamos el navegador. Deberíamos ver el nombre del usuario en la vista.

Para probar nuestro proyecto, vamos a recordar lo que aprendimos en el capítulo anterior. Agregamos nuestro componente **usuario** al archivo **app.component.html**:

```
<app-usuario></app-usuario>
```

Guardamos y agregamos el componente a las importaciones de la aplicación principal. Abrimos el archivo **app.module.ts** y agregamos el siguiente código:

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { UsuarioComponent } from './usuario/usuario.component';          ①

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    UsuarioComponent                                                     ②
  ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'interpolacion';
}
```

① Importamos el componente **UsuarioComponent**.
② Agregamos el componente **UsuarioComponent** a las importaciones de la aplicación.

Si todo está correcto, deberíamos ver el nombre del usuario en la vista.

> **Tip**
>
> Las propiedades solo pueden ser llamadas en las vistas de los componentes que las contienen. Si intentaramos llamar la propiedad **nombre** en el archivo **app.component.html**, obtendríamos un error.

Al utilizar typescript como lenguaje de programación las propiedades que definimos pueden ser de tipado estricto, es decir en el ejemplo anterior la propiedad **nombre** es de tipo **string**. Si intentamos asignar un valor de otro tipo a la propiedad obtendremos un error.

```
nombre: string = 10; // Error
```

En el ejemplo anterior se define la propiedad **nombre** como un número, pero se le asigna un valor de tipo **string**. Para corregir el error, debemos asignar un valor de tipo **string** a la propiedad **nombre**.

```
nombre: string = '10'; // Correcto
```

Recordemos los tipos de datos que podemos utilizar en typescript:

- **number**: Números enteros o decimales.
- **string**: Cadenas de texto.
- **boolean**: Valores booleanos (true o false).
- **any**: Cualquier tipo de dato.
- **Array**: Arreglos de datos.
- **Object**: Objetos.

Ahora vamos a intentar utilizar un objeto, en este caso un objeto de tipo **Persona**, para definirlo dentro del componente **usuario** y mostrar sus propiedades en la vista.

Modificamos el archivo **usuario.component.ts** y agregamos la siguiente propiedad:

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

@Component({
  selector: 'app-usuario',
  standalone: true,
  imports: [],
  templateUrl: './usuario.component.html',
  styleUrl: './usuario.component.css'
})
export class UsuarioComponent {
  persona: any = {                          ①
    nombre: 'Diego',                        ②
    edad: 36,                               ③
    direccion: {                            ④
    calle: 'Calle 123',                     ⑤
    ciudad: 'Quito'                         ⑥
    }
  }
}
```

① Definimos la propiedad **persona**.
② Definimos la propiedad **nombre**.
③ Definimos la propiedad **edad**.
④ Definimos la propiedad **direccion**.
⑤ Definimos la propiedad **calle**.
⑥ Definimos la propiedad **ciudad**.

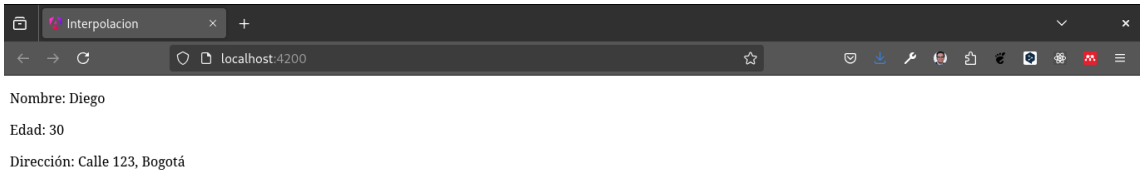
En el ejemplo anterior, definimos un objeto de tipo **Persona** con las propiedades **nombre**, **edad** y **direccion**. La propiedad **direccion** es un objeto que contiene las propiedades **calle** y **ciudad**.

Ahora, abrimos el archivo **usuario.component.html** y agregamos el siguiente código:

```
<p>Nombre: {{ persona.nombre }}</p>
<p>Edad: {{ persona.edad }}</p>
<p>Dirección: {{ persona.direccion.calle }}, {{ persona.direccion.ciudad }}</p>
```

En el archivo **usuario.component.html** mostramos las propiedades **nombre**, **edad** y **direccion** del objeto **persona**. La propiedad **direccion** es un objeto que contiene las propiedades **calle** y **ciudad**.

Si todo está correcto, deberíamos ver el nombre, la edad y la dirección de la persona en la vista.



Ahora si nos fijamos bien nuestro objeto persona es de tipo **any** lo que significa que puede ser cualquier tipo de dato, pero si queremos que nuestro objeto persona sea de tipo **Persona** debemos crear una interfaz que defina las propiedades del objeto.

Vamos a crear una interfaz llamada **Persona**. Para ello, abrimos el archivo **usuario.ts** y agregamos la siguiente interfaz:

```
export interface Persona {                    ①
  nombre: string;                             ②
  edad: number;                               ③
  direccion: {                                ④
    calle: string;                            ⑤
    ciudad: string;                           ⑥
  }
}
```

① Definimos la interfaz **Persona**.
② Definimos la propiedad **nombre**.
③ Definimos la propiedad **edad**.
④ Definimos la propiedad **direccion**.
⑤ Definimos la propiedad **calle**.
⑥ Definimos la propiedad **ciudad**.

Ahora, modificamos la propiedad **persona** para que sea de tipo **Persona**. Para ello, abrimos el archivo **usuario.component.ts** y modificamos la propiedad **persona** de la siguiente manera:

```
import { Component } from '@angular/core';
import { Persona } from './usuario';                    ①

@Component({
```

```
  selector: 'app-usuario',
  standalone: true,
  imports: [],
  templateUrl: './usuario.component.html',
  styleUrl: './usuario.component.css'
})
export class UsuarioComponent {
  persona: Persona = {                                                    ②
    nombre: 'Diego',
    edad: 36,
    direccion: {
      calle: 'Calle 123',
      ciudad: 'Quito'
    }
  }
}
```

> **Tip**
>
> ¿Qué conseguimos con esto?
> Al definir la propiedad **persona** como un objeto de tipo **Persona**, estamos asegurando que el objeto **persona** tenga las propiedades **nombre**, **edad** y **direccion**. Si intentamos agregar una propiedad que no está definida en la interfaz **Persona**, obtendremos un error.

Si todo está correcto, deberíamos ver el nombre, la edad y la dirección de la persona en la vista.

62 Reto

Crea un nuevo proyecto de Angular llamado **reto-interpolacion**. Crea un nuevo componente llamado **producto**. Define un objeto de tipo **Producto** con las propiedades **nombre**, **precio** y **descripcion**. Muestra las propiedades del objeto **producto** en la vista del componente **producto**.

Solución

1. Creamos un nuevo proyecto de Angular llamado **reto-interpolacion**.

```
ng new reto-interpolacion
```

2. Nos ubicamos en el directorio del proyecto.

```
cd reto-interpolacion
```

3. Creamos un nuevo componente llamado **producto**.

```
ng g c producto
```

4. Definimos una interfaz llamada **Producto** en el archivo **producto.ts**.

```
export interface Producto {
  nombre: string;
  precio: number;
  descripcion: string;
}
```

5. Modificamos el archivo **producto.component.ts** y definimos un objeto de tipo **Producto**.

```
import { Component } from '@angular/core';
import { Producto } from './producto';                    ①

@Component({
  selector: 'app-producto',
  standalone: true,
  imports: [],
  templateUrl: './producto.component.html',
  styleUrl: './producto.component.css'
})
export class ProductoComponent {
```

```
  producto: Producto = {                                                    ②
    nombre: 'Laptop',
    precio: 1000,
    descripcion: 'Laptop de última generación'
  }
}
```

```
<p>Nombre: {{ producto.nombre }}</p>
<p>Precio: {{ producto.precio }}</p>
<p>Descripción: {{ producto.descripcion }}</p>
```

7. Agregamos el componente **producto** al archivo **app.component.html**.

```
<app-producto></app-producto>
```

8. Agregamos el componente **producto** a las importaciones de la aplicación principal.

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ProductoComponent } from './producto/producto.component';        ①

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    ProductoComponent                                                     ②
  ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})

export class AppComponent {
  title = 'reto-interpolacion';
}
```

```
ng serve -o
```

Si todo está correcto, deberíamos ver el nombre, el precio y la descripción del producto en la vista.

Nombre: Laptop

Precio: 1000

Descripción: Laptop de última generación

63 Conclusión

La interpolación es una forma de mostrar datos en la vista de un componente. Se utiliza la sintaxis de doble llave **{{ }}** para mostrar datos en la vista. Las propiedades solo pueden ser llamadas en las vistas de los componentes que las contienen. Al utilizar typescript como lenguaje de programación las propiedades que definimos pueden ser de tipado estricto. Si queremos que un objeto sea de un tipo específico, podemos definir una interfaz que defina las propiedades del objeto. Al definir un objeto como una interfaz, estamos asegurando que el objeto tenga las propiedades definidas en la interfaz.

64 Event Binding en Angular

En esta unidad vamos a conocer cómo podemos reaccionar a eventos del usuario en Angular. Para ello, vamos a ver cómo podemos utilizar el event binding en Angular.

Para entender este tema vamos a crear un nuevo proyecto Angular. Para ello, abrimos una terminal y ejecutamos el siguiente comando:

```
ng new event-binding
```

Una vez creado el proyecto, nos movemos a la carpeta del proyecto:

```
cd event-binding
```

Y abrimos el proyecto en Visual Studio Code:

```
code .
```

64.1 Event Binding

El event binding en Angular nos permite reaccionar a eventos del usuario, como por ejemplo, un click, un doble click, un hover, etc. Para ello, vamos a ver un ejemplo sencillo.

Vamos a crear un nuevo componente llamado **contador**:

```
ng g c contador
```

Una vez creado el componente, nos movemos al archivo **contador.component.html** y escribimos el siguiente código:

```
<h1>Contador</h1>

<p>{{ contador }}</p>

<button>Incrementar</button>
<button>Decrementar</button>
```

En este código, tenemos un título, un párrafo que muestra el valor del contador y dos botones, uno para incrementar el contador y otro para decrementarlo.

Para poder reaccionar a los eventos de click de los botones, vamos a utilizar el event binding. Para ello, nos movemos al archivo **contador.component.html** y escribimos el siguiente código:

```
<h1>Contador</h1>

<p>{{ contador }}</p>

<button (click)="incrementar()">Incrementar</button>
<button (click)="decrementar()">Decrementar</button>
```

En este código, estamos utilizando el event binding para reaccionar al evento de click de los botones. Cuando se haga click en el botón de incrementar, se va a ejecutar el método **incrementar()** y cuando se haga click en el botón de decrementar, se va a ejecutar el método **decrementar()**.

Ahora, nos movemos al archivo **contador.component.ts** y escribimos el siguiente código:

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

@Component({
  selector: 'app-contador',
  standalone: true,
  imports: [],
  templateUrl: './contador.component.html',
  styleUrl: './contador.component.css'
})
export class ContadorComponent {

    contador: number = 0;

    incrementar() {
        this.contador++;
    }

    decrementar() {
        this.contador--;
    }
}
```

En este código, estamos creando una variable **contador** que va a almacenar el valor del contador y dos métodos **incrementar()** y **decrementar()** que se van a encargar de incrementar y decrementar el contador respectivamente.

> Tip
>
> No olvides agregar el componente **ContadorComponent** a la aplicación principal en el archivo **app.component.ts**.

```ts
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { ContadorComponent } from './contador/contador.component';  ①

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    ContadorComponent                                                ②
  ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'event-binding';
}
```

① Importamos el componente **ContadorComponent**.
② Agregamos el componente **ContadorComponent** al arreglo de imports.

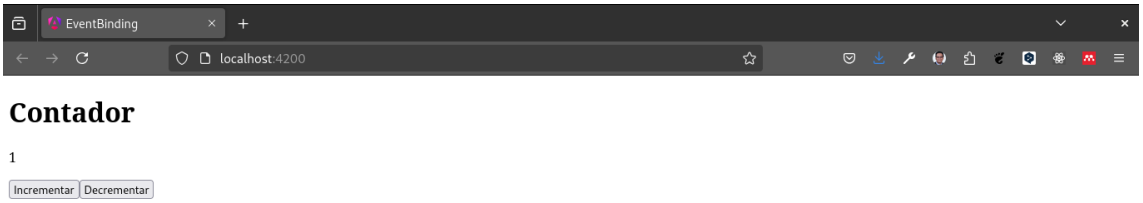
Tambien es necesario agregar el selector del componente **ContadorComponent** en el archivo **app.component.html**.

```html
<app-contador></app-contador>
```

Ahora, si ejecutamos la aplicación con el siguiente comando:

```
ng s -o
```

Podremos ver el contador en la pantalla y si hacemos click en los botones de incrementar y decrementar, el contador se va a incrementar y decrementar respectivamente.


EventBinding
localhost:4200
Contador
1
Incrementar
Decrementar

65 Reto

Crea un nuevo proyecto Angular y crea un nuevo componente llamado **calculadora**.

```
ng new calculadora
cd calculadora
ng g c calculadora
```

En este componente, crea una calculadora sencilla con dos inputs para ingresar los números y cuatro botones para realizar las operaciones de suma, resta, multiplicación y división. Utiliza el event binding para reaccionar a los eventos de click de los botones y mostrar el resultado de la operación en un párrafo.

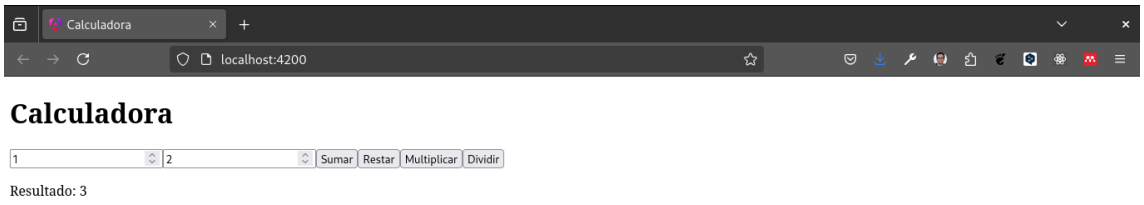
Posible solución

```
<h1>Calculadora</h1>

<input type="number" [(ngModel)]="numero1" placeholder="Número 1">
<input type="number" [(ngModel)]="numero2" placeholder="Número 2">

<button (click)="sumar()">Sumar</button>
<button (click)="restar()">Restar</button>
<button (click)="multiplicar()">Multiplicar</button>
<button (click)="dividir()">Dividir</button>

<p *ngIf="resultado !== undefined">Resultado: {{ resultado }}</p>
<p *ngIf="resultado === undefined">Realiza una operación</p>
```

```
import { CommonModule } from '@angular/common';
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-calculadora',
  standalone: true,
  imports: [CommonModule, FormsModule],
  templateUrl: './calculadora.component.html',
  styleUrls: ['./calculadora.component.css']
})
export class CalculadoraComponent {
  numero1: number | undefined;
  numero2: number | undefined;
```

```
  resultado: number | undefined;

  sumar() {
    this.resultado = (this.numero1 ?? 0) + (this.numero2 ?? 0);
  }

  restar() {
    this.resultado = (this.numero1 ?? 0) - (this.numero2 ?? 0);
  }

  multiplicar() {
    this.resultado = (this.numero1 ?? 0) * (this.numero2 ?? 0);
  }

  dividir() {
    this.resultado = (this.numero1 ?? 0) / (this.numero2 ?? 0);
  }
}
```

Si todo salio bien, deberías ver algo como esto:



66 Conclusión

En esta unidad aprendimos cómo podemos reaccionar a eventos del usuario en Angular utilizando el event binding. Vimos cómo podemos reaccionar a eventos de click de los botones y cómo podemos ejecutar métodos en el componente cuando se produce un evento. En el siguiente tema, vamos a ver cómo podemos reaccionar a eventos del usuario utilizando el event binding en Angular.

67 Bootstrap en Angular

En esta unidad aprenderemos a agregar Bootstrap a un proyecto Angular.

67.1 Introducción

Bootstrap es un framework de diseño que nos permite crear interfaces de usuario de forma rápida y sencilla. Bootstrap nos proporciona una serie de estilos CSS y componentes HTML que podemos utilizar en nuestros proyectos.

Para agregar Bootstrap a un proyecto Angular, podemos hacerlo de dos formas:

1. Descargando los archivos CSS y JS de Bootstrap y agregándolos a nuestro proyecto.
2. Utilizando la librería **ngx-bootstrap**.

En esta unidad aprenderemos a agrear Bootstrap mediante los archivos CSS y JS.

Creamos un proyecto Angular con el siguiente comando:

```
ng new proyecto-bootstrap
```

Nos ubicamos en la carpeta del proyecto:

```
cd proyecto-bootstrap
code .
```

67.2 Agregando Bootstrap a un proyecto Angular

Para agregar Bootstrap al proyecto vamos a utilizar el CDN de Bootstrap. Vamos a agregar los archivos CSS y JS de Bootstrap en el archivo **index.html** que se encuentra en la carpeta **src**.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>ProyectoBootstrap</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" re
```

```
</head>
<body>
  <app-root></app-root>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.
</body>
</html>
```

① Agregamos el archivo CSS de Bootstrap.
② Agregamos el archivo JS de Bootstrap.

67.3 Creando un componente

Vamos a crear un componente llamado **navbar** que contendrá un menú de navegación.

```
ng g c navbar
```

67.4 Agregando el componente a la aplicación principal

Vamos a agregar el componente **navbar** al archivo **app.component.html**.

```
<app-navbar></app-navbar>
```

Agregamos tambien el componente **navbar** al archivo **app.module.ts**.

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import { NavbarComponent } from './navbar/navbar.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    RouterOutlet,
    NavbarComponent
  ],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'proyecto-bootstrap';
}
```

67.5 Creando el menú de navegación

Vamos a agregar el siguiente código al archivo **navbar.component.html**.

```
<ul class="nav nav-pills nav-fill gap-2 p-1 small bg-primary rounded-5 shadow-sm" id="pil
  <li class="nav-item" role="presentation">
    <button class="nav-link active rounded-5" id="home-tab2" data-bs-toggle="tab" type="b
  </li>
  <li class="nav-item" role="presentation">
    <button class="nav-link rounded-5" id="profile-tab2" data-bs-toggle="tab" type="butto
  </li>
  <li class="nav-item" role="presentation">
    <button class="nav-link rounded-5" id="contact-tab2" data-bs-toggle="tab" type="butto
  </li>
</ul>
```

67.6 Agregando un contenedor

Vamos a agregar un contenedor a la aplicación para centrar el contenido.

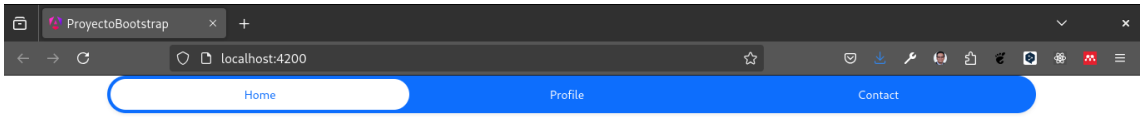
```
<div class="container">
  <app-navbar></app-navbar>
</div>
```

67.7 Ejecutando la aplicación

Vamos a ejecutar la aplicación con el siguiente comando:

```
ng serve -o
```

Si todo salio bien deberíamos ver el menú de navegación en la aplicación.


ProyectoBootstrap
localhost:4200
Home
Profile
Contact

68 Reto

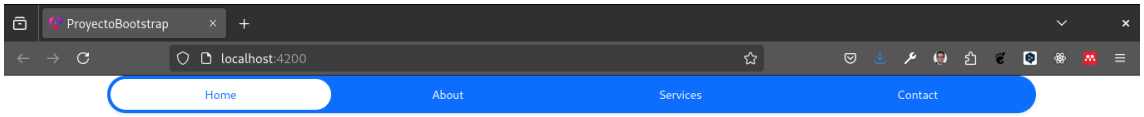
Agrega un menú de navegación a la aplicación que contenga los siguientes elementos:

1. Home
2. About
3. Services
4. Contact

Utiliza los estilos de Bootstrap para darle un aspecto agradable al menú.

Posible solución

```
<ul class="nav nav-pills nav-fill gap-2 p-1 small bg-primary rounded-5 shadow-sm" id="pil
  <li class="nav-item" role="presentation">
    <button class="nav-link active rounded-5" id="home-tab2" data-bs-toggle="tab" type="b
  </li>
  <li class="nav-item" role="presentation">
    <button class="nav-link rounded-5" id="about-tab2" data-bs-toggle="tab" type="button"
  </li>
  <li class="nav-item" role="presentation">
    <button class="nav-link rounded-5" id="services-tab2" data-bs-toggle="tab" type="butt
  </li>
  <li class="nav-item" role="presentation">
    <button class="nav-link rounded-5" id="contact-tab2" data-bs-toggle="tab" type="butto
  </li>
</ul>
```


ProyectoBootstrap
localhost:4200
Home
About
Services
Contact

69 Conclusión

Hemos aprendido a agregar Bootstrap a un proyecto Angular. Bootstrap nos permite crear interfaces de usuario de forma rápida y sencilla. En la siguiente unidad aprenderemos a agregar estilos personalizados a un proyecto Angular.

Part IV

Unidad 5: Creación de Proyectos con Angular

70 Tutorial: Crear un CRUD en Angular

En este capítulo aprenderás cómo crear un CRUD (Crear, Leer, Actualizar, Eliminar) en Angular. Cubriremos cómo configurar un proyecto Angular, cómo crear componentes y servicios para manejar las operaciones CRUD, cómo implementar la funcionalidad de listar, agregar, editar y eliminar elementos, y cómo configurar el enrutamiento en Angular.

70.1 Setup del Proyecto

Crear un nuevo proyecto Angular utilizando Angular CLI

2. Instalar Angular CLI (si no lo tienes instalado):

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

3. Crear un nuevo proyecto Angular:

```
ng new angular-crud
cd angular-crud
```

4. Iniciar el servidor de desarrollo:

```
ng serve
```

Navega a http://localhost:4200/ para ver la aplicación en funcionamiento.

70.2 Configurar el proyecto

Configurar el proyecto para utilizar un servicio backend simulado con json-server o in-memory-web-api

Para este tutorial, usaremos json-server.

> Tip
>
> ¿Qué es json-server?
> Es una herramienta que nos permite **crear** un **servidor RESTful simulado** a partir de un archivo **JSON**. Es útil para prototipar aplicaciones y realizar pruebas sin tener que configurar un servidor backend real.

1. Instalar json-server:

```
npm install -g json-server
```

Crear un archivo **db.json** en la raíz del proyecto con datos iniciales:

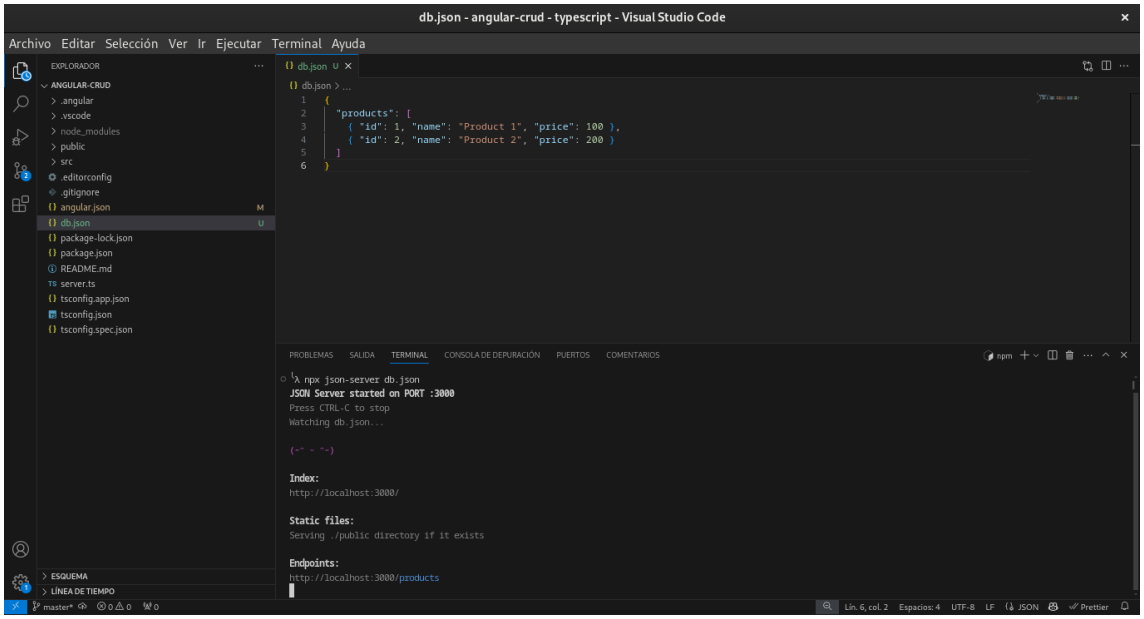
```
{
  "products": [
    { "id": 1, "name": "Product 1", "price": 100 },
    { "id": 2, "name": "Product 2", "price": 200 }
  ]
}
```

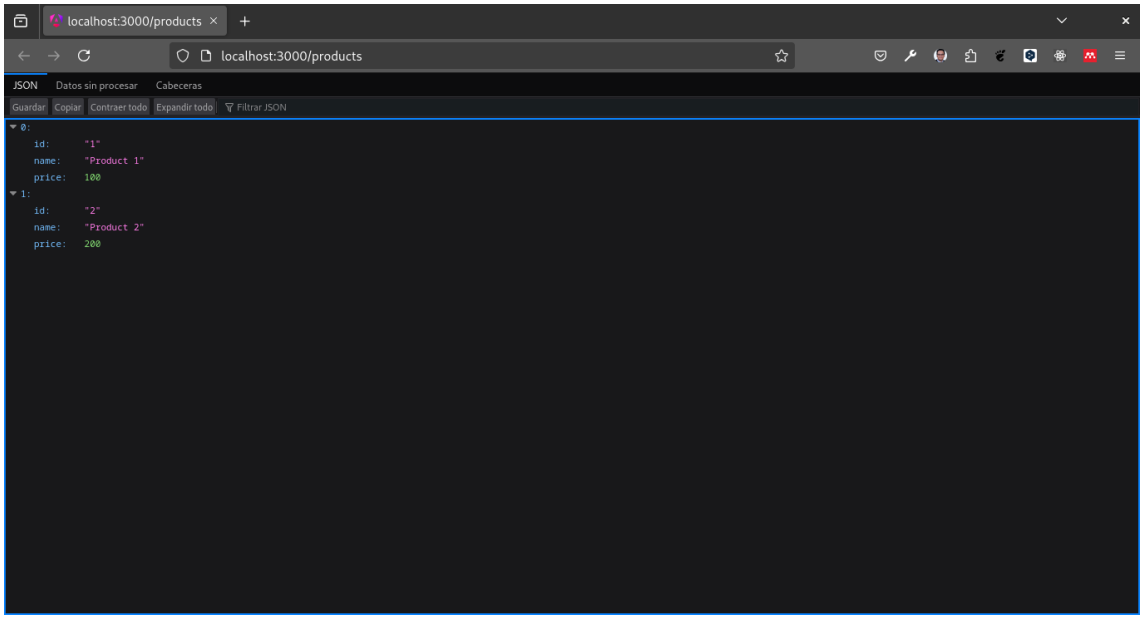
2. Iniciar json-server:

```
npx json-server db.json
```

> **Tip**
>
> **¿Qué es npx?**
> Es un ejecutable que viene con npm 5.2.0 o superior. Se utiliza para ejecutar paquetes instalados localmente en el proyecto sin tener que instalarlos globalmente.

El servidor estará disponible en http://localhost:3000.





70.3 Componentes y Servicios

Crear componentes para listar, agregar, editar y eliminar elementos

1. Generar los componentes:

```
ng generate component product-list
ng generate component product-add
ng generate component product-edit
ng generate component product-delete
```

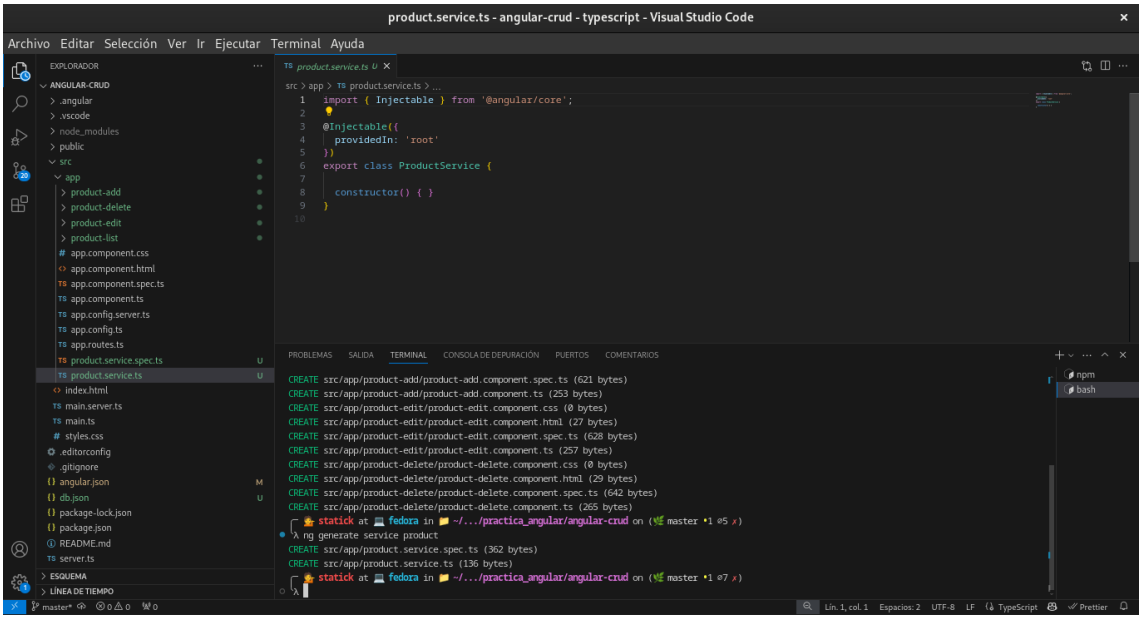
70.4 Creación de Operaciones CRUD

Crear un servicio Angular para manejar las operaciones CRUD contra el backend simulado.

> Tip
>
> **¿Qué es un servicio en Angular?**
> Un servicio en Angular es una clase que se utiliza para **compartir datos y funcionalidades** entre componentes. Se utiliza para **manejar operaciones asíncronas** como llamadas HTTP, y para **compartir datos** entre componentes.

1. Generar el servicio:

```
ng generate service product
```



Modificar el servicio **product.service.ts** para manejar las operaciones CRUD:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

interface Product {
    id: number;
    name: string;
    price: number;
}

@Injectable({
    providedIn: 'root'
})
export class ProductService {
    private apiUrl = 'http://localhost:3000/products';

    constructor(private http: HttpClient) {}

    getProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(this.apiUrl);
    }

    getProduct(id: number): Observable<Product> {
    return this.http.get<Product>(`${this.apiUrl}/${id}`);
    }

    addProduct(product: Product): Observable<Product> {
    return this.http.post<Product>(this.apiUrl, product);
    }
```

```
    updateProduct(product: Product): Observable<Product> {
    return this.http.put<Product>(`${this.apiUrl}/${product.id}`, product);
    }

    deleteProduct(id: number): Observable<void> {
    return this.http.delete<void>(`${this.apiUrl}/${id}`);
    }
}
```

En el servicio **product.service.ts** se definen las operaciones CRUD para manejar los productos. Se utilizan los métodos **get**, **post**, **put** y **delete** del servicio HttpClient para realizar las operaciones CRUD.

70.5 Operaciones CRUD

Implementar la funcionalidad para listar todos los productos

1. Modificar el componente **product-list.component.ts**:

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { ProductService } from '../product.service';
import { CommonModule } from '@angular/common';

interface Product {
  id: number;
  name: string;
  price: number;
}

@Component({
  selector: 'app-product-list',
  standalone: true,
  imports: [FormsModule, CommonModule],
  providers: [ProductService],
  templateUrl: './product-list.component.html',
  styleUrl: './product-list.component.css'
})

export class ProductListComponent {

deleteProduct(arg0: number) {
throw new Error('Method not implemented.');
}

editProduct(arg0: number) {
```

```
throw new Error('Method not implemented.');
}
  products: Product[] = [];

  constructor(private productService: ProductService) {}

  ngOnInit(): void {
    this.productService.getProducts().subscribe((data: Product[]) => {
      this.products = data;
    });
  }
}
```

En el componente **product-list.component.ts** se define la clase **ProductListComponent** que se encarga de listar los productos. Se utiliza el método **getProducts** del servicio **ProductService** para obtener los productos y se almacenan en la propiedad **products**.

2. Modificar el template **product-list.component.html**:

```
<h2>Product List</h2>
<ul>
    <li *ngFor="let product of products">
    {{ product.name }} - ${{ product.price }}
    <button (click)="editProduct(product.id)">Edit</button>
    <button (click)="deleteProduct(product.id)">Delete</button>
    </li>
</ul>
<button routerLink="/add">Add Product</button>
```

En el template **product-list.component.html** se utiliza la directiva **ngFor** para iterar sobre la lista de productos y mostrarlos en una lista. Se utilizan los botones **Edit** y **Delete** para editar y eliminar los productos respectivamente.

70.6 Implementar la funcionalidad para agregar un nuevo producto

1. Modificar el componente **product-add.component.ts**:

```
import { Component } from '@angular/core';
import { ProductService } from '../product.service';
import { Router } from '@angular/router';
import { FormsModule } from '@angular/forms';

interface Product {
  id: number;
  name: string;
```

```
  price: number;
}

@Component({
  selector: 'app-product-add',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './product-add.component.html',
  styleUrl: './product-add.component.css'
})
export class ProductAddComponent {
  product: Product = { id: 0, name: '', price: 0 };

  constructor(private productService: ProductService, private router: Router) {}

  addProduct(): void {
    this.productService.addProduct(this.product).subscribe(() => {
      this.router.navigate(['/']);
    });
  }
}
```

En el componente **product-add.component.ts** se define la clase **ProductAddComponent** que se encarga de agregar un nuevo producto. Se utiliza el método **addProduct** del servicio **ProductService** para agregar el producto y se redirige al componente **ProductListComponent**.

2. Modificar el template **product-add.component.html**:

```
<h2>Add Product</h2>
<form (ngSubmit)="addProduct()">
    <label for="name">Name</label>
    <input id="name" [(ngModel)]="product.name" name="name" required>
    <label for="price">Price</label>
    <input id="price" [(ngModel)]="product.price" name="price" required>
    <button type="submit">Add</button>
</form>
```

En el template **product-add.component.html** se define un formulario para agregar un nuevo producto. Se utilizan las directivas **ngModel** y **ngSubmit** para enlazar los campos del formulario con la propiedad **product** y para manejar el evento de envío del formulario.

70.7 Implementar la funcionalidad para editar un producto existente

1. Modificar el componente **product-edit.component.ts**:

```
import { Component, Inject } from '@angular/core';
import { ProductService } from '../product.service';
import { ActivatedRoute, Router } from '@angular/router';
import { FormsModule } from '@angular/forms';

interface Product {
  id: number;
  name: string;
  price: number;
}

@Component({
  selector: 'app-product-edit',
  standalone: true,
  providers: [ProductService],
  imports: [FormsModule],
  templateUrl: './product-edit.component.html',
  styleUrl: './product-edit.component.css'
})
export class ProductEditComponent {

  product: Product = { id: 0, name: '', price: 0 };

  constructor(
    private productService: ProductService,
    @Inject(ActivatedRoute) private route: ActivatedRoute,
    @Inject(Router) private router: Router
  ) {}

  ngOnInit(): void {
    const id = Number(this.route.snapshot.paramMap.get('id'));
    this.productService.getProduct(id).subscribe((data: Product) => {
      this.product = data;
    });
  }

  updateProduct(): void {
    this.productService.updateProduct(this.product).subscribe(() => {
      this.router.navigate(['/']);
    });
  }
}
```

En el componente **product-edit.component.ts** se define la clase **ProductEditComponent** que se encarga de editar un producto existente. Se utiliza el método **getProduct** del servicio **ProductService** para obtener el producto a editar y se utiliza el método **updateProduct** para actualizar el producto.

2. Modificar el template **product-edit.component.html**:

```
<h2>Edit Product</h2>
<form (ngSubmit)="updateProduct()">
    <label for="name">Name</label>
    <input id="name" [(ngModel)]="product.name" name="name" required>
    <label for="price">Price</label>
    <input id="price" [(ngModel)]="product.price" name="price" required>
    <button type="submit">Update</button>
</form>
```

En el template **product-edit.component.html** se define un formulario para editar un producto existente. Se utilizan las directivas **ngModel** y **ngSubmit** para enlazar los campos del formulario con la propiedad **product** y para manejar el evento de envío del formulario.

70.8 Implementar la funcionalidad para eliminar un producto

1. Modificar el componente product-delete.component.ts para agregar la funcionalidad de eliminación:

```
import { Component, Inject } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { ProductService } from '../product.service';
import { Router, ActivatedRoute } from '@angular/router';
import { CommonModule } from '@angular/common';

interface Product {
  id: number;
  name: string;
  price: number;
}

@Component({
  selector: 'app-product-delete',
  standalone: true,
  providers: [ProductService],
  imports: [FormsModule, FormsModule, CommonModule],
  templateUrl: './product-delete.component.html',
  styleUrl: './product-delete.component.css'
})
export class ProductDeleteComponent {
  product: Product | null = null;

  constructor(
    private productService: ProductService,
```

```
    @Inject(ActivatedRoute) private route: ActivatedRoute,
    private router: Router
  ) {}

  ngOnInit(): void {
    const id = Number(this.route.snapshot.paramMap.get('id'));
    this.productService.getProduct(id).subscribe((data: Product) => {
      this.product = data;
    });
  }

  deleteProduct(): void {
    if (this.product) {
      this.productService.deleteProduct(this.product.id).subscribe(() => {
        this.router.navigate(['/']);
      });
    }
  }

  cancel(): void {
    this.router.navigate(['/']);
  }
}
```

En el componente **product-delete.component.ts** se define la clase **ProductDeleteComponent** que se encarga de eliminar un producto. Se utiliza el método **getProduct** del servicio **ProductService** para obtener el producto a eliminar y se utiliza el método **deleteProduct** para eliminar el producto.

2. Modificar el template **product-delete.component.html**:

```
<h2>Delete Product</h2>

<div *ngIf="product">
  <p>Are you sure you want to delete the product "{{ product.name }}"?</p>
  <button (click)="deleteProduct()">Yes, Delete</button>
  <button (click)="cancel()">Cancel</button>
</div>

<div *ngIf="!product">
  <p>Loading...</p>
</div>
```

En el template **product-delete.component.html** se muestra un mensaje de confirmación para eliminar el producto. Se utilizan los botones **Yes, Delete** y **Cancel** para confirmar o cancelar la eliminación del producto.

70.9 Routes

Para configurar las rutas de la aplicación, modificar el archivo **app.routes.ts**:

```
import { Routes } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.component';
import { ProductAddComponent } from './product-add/product-add.component';
import { ProductEditComponent } from './product-edit/product-edit.component';
import { ProductDeleteComponent } from './product-delete/product-delete.component';

export const routes: Routes = [
  { path: 'product-list', component: ProductListComponent },
  { path: 'product-add', component: ProductAddComponent },
  { path: 'product-edit/:id', component: ProductEditComponent },
  { path: 'product-delete/:id', component: ProductDeleteComponent },
  { path: '', redirectTo: '/product-list', pathMatch: 'full' },
//   { path: '**', component: PageNotFoundComponent } // Asegúrate de haber creado este c
];
```

En el archivo **app.routes.ts** se definen las rutas de la aplicación. Se utiliza la propiedad **path** para definir la URL de la ruta y la propiedad **component** para definir el componente asociado a la ruta. Se utiliza la ruta **''** para redirigir al componente **ProductListComponent** cuando no se especifica ninguna ruta.

> **Tip**
>
> Tomar en cuenta que no ha sido implementado el componente **PageNotFoundComponent**.

Modificar el archivo **app.component.ts** para importar las rutas:

```
import { Component } from '@angular/core';
import { RouterOutlet, RouterLink, RouterLinkActive, RouterModule } from '@angular/router
import { CommonModule } from '@angular/common';

import { ProductDeleteComponent } from './product-delete/product-delete.component';
import { ProductListComponent } from './product-list/product-list.component';
import { ProductEditComponent } from './product-edit/product-edit.component';
import { ProductAddComponent } from './product-add/product-add.component';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [
    CommonModule,
    RouterOutlet,
    RouterLink,
    RouterLinkActive,
```

```
    RouterModule,
    ProductListComponent,
    ProductAddComponent,
    ProductEditComponent,
    ProductDeleteComponent
  ],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-crud';
}
```

En el archivo **app.component.ts** se importan las rutas definidas en el archivo **app.routes.ts**.

Modificar el archivo **app.component.html** para agregar las rutas:

```
<h1>{{ title }}</h1>
<nav>
  <a class="button" routerLink="/product-list" routerLinkActive="activebutton" ariaCurren
  <a class="button" routerLink="/product-add" routerLinkActive="activebutton" ariaCurrent
  <a class="button" routerLink="/product-edit" routerLinkActive="activebutton" ariaCurren
  <a class="button" routerLink="/product-delete" routerLinkActive="activebutton" ariaCurr
</nav>
<router-outlet></router-outlet>
```

En el archivo **app.component.html** se definen los enlaces de navegación para las rutas de la aplicación. Se utilizan las directivas **routerLink** y **routerLinkActive** para enlazar las rutas con los botones de navegación y para resaltar el botón activo.

> **Tip**
>
> Se utiliza router-outlet para mostrar el componente asociado a la ruta actual.

Ahora vamos a modificar el archivo **main.ts** para importar las rutas:

```
import { bootstrapApplication } from '@angular/platform-browser';
import { provideHttpClient } from '@angular/common/http'; //<1
import { provideRouter } from '@angular/router';

import { AppComponent } from './app/app.component';
import { routes } from './app/app.routes';                    ②

bootstrapApplication(AppComponent, {
  providers: [
    provideHttpClient(),                                      ③
    provideRouter(routes)                                     ④
```

```
  ]
}).catch(err => console.error(err));
```

② En el archivo **main.ts** se importan las rutas definidas en el archivo **app.routes.ts**.
③ En el archivo **main.ts** se proporcionan los servicios **HttpClient** y **Router**.
④ En el archivo **main.ts** se proporcionan las rutas de la aplicación.

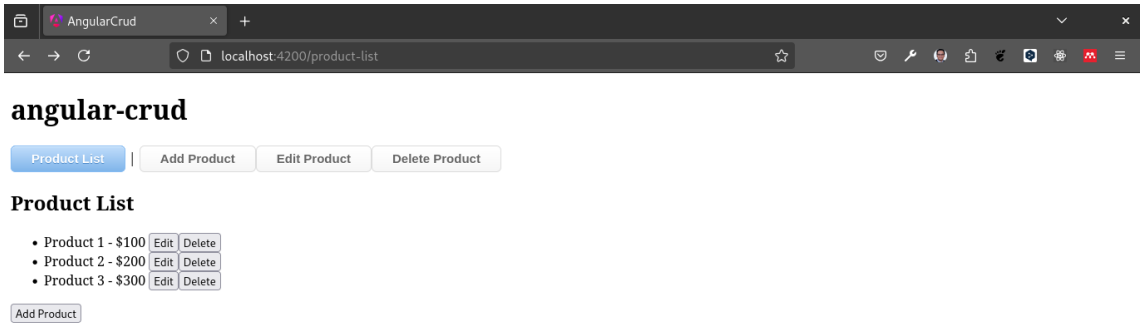
En el archivo **main.ts** se importan las rutas definidas en el archivo **app.routes.ts** y se proporcionan los servicios **HttpClient** y **Router**.

Finalmente modificar el archivo **app.module.css** para agregar estilos:

```
.button {
    box-shadow: inset 0 1px 0 0 #ffffff;
    background: #ffffff linear-gradient(to bottom, #ffffff 5%, #f6f6f6 100%);
    border-radius: 6px;
    border: 1px solid #dcdcdc;
    display: inline-block;
    cursor: pointer;
    color: #666666;
    font-family: Arial, sans-serif;
    font-size: 15px;
    font-weight: bold;
    padding: 6px 24px;
    text-decoration: none;
    text-shadow: 0 1px 0 #ffffff;
    outline: 0;
  }
  .activebutton {
    box-shadow: inset 0 1px 0 0 #dcecfb;
    background: #bddbfa linear-gradient(to bottom, #bddbfa 5%, #80b5ea 100%);
    border: 1px solid #84bbf3;
    color: #ffffff;
    text-shadow: 0 1px 0 #528ecc;
  }
```

En el archivo **app.module.css** se definen los estilos para los botones de navegación.

Si todo salio bien deberías ver algo como esto:


AngularCrud
localhost:4200/product-list
angular-crud
Product List
Add Product
Edit Product
Delete Product
Product List
Product 1 - $100 Edit Delete
Product 2 - $200 Edit Delete
Product 3 - $300 Edit Delete
Add Product

71 Reto

Realizar los siguientes ajustes:

1. Los componentes edit y delete no están completos, intente repararlos y hacer que funcionen correctamente.
2. Crear el componente PageNotFoundComponent y configurar la ruta en el archivo **app.routes.ts**.
3. Modifique los estilos de la aplicación para que se vea más atractiva.

72 Recursos

- Código del Proyecto

73 Conclusión

En este tutorial, aprendiste cómo crear un CRUD en Angular. Cubrimos cómo configurar un proyecto Angular, cómo crear componentes y servicios para manejar las operaciones CRUD, cómo implementar la funcionalidad de listar, agregar, editar y eliminar elementos, y cómo configurar el enrutamiento en Angular. Ahora puedes aplicar estos conceptos para crear tus propias aplicaciones CRUD en Angular.