

1) Instalacion y creacion de proyecto:

Comenzamos instalando la herramienta Angular CLI, para ejecutar los comandos de Angular, se puede realizar instalar de manera global para todo el equipo, simplemente hay que abrir una consola teclear el siguiente comando `npm install -g @angular/cli`.

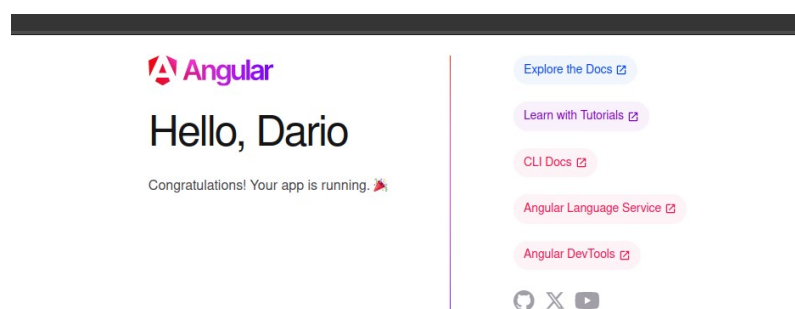
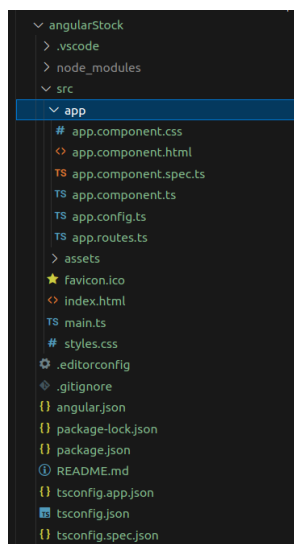
Ahora ya podemos generar un proyecto Angular. Para ello, completamos en cualquier ubicación de nuestro equipo el siguiente comando, `ng new <nombre del proyecto>` que incluye el nombre del directorio donde se generarán todos los archivos.

```
dario@dario-GL553VD:~/Escritorio/angular$ ng new angularStock
? Which stylesheet format would you like to use? CSS
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
CREATE angularStock/README.md (1066 bytes)
CREATE angularStock/.editorconfig (274 bytes)
CREATE angularStock/.gitignore (548 bytes)
CREATE angularStock/angular.json (2623 bytes)
CREATE angularStock/package.json (1044 bytes)
CREATE angularStock/tsconfig.json (903 bytes)
CREATE angularStock/tsconfig.app.json (263 bytes)
CREATE angularStock/tsconfig.spec.json (273 bytes)
CREATE angularStock/.vscode/extensions.json (130 bytes)
CREATE angularStock/.vscode/launch.json (470 bytes)
CREATE angularStock/.vscode/tasks.json (938 bytes)
CREATE angularStock/src/main.ts (250 bytes)
CREATE angularStock/src/favicon.ico (15086 bytes)
CREATE angularStock/src/index.html (298 bytes)
CREATE angularStock/src/styles.css (80 bytes)
CREATE angularStock/src/app/app.component.css (0 bytes)
CREATE angularStock/src/app/app.component.html (19903 bytes)
CREATE angularStock/src/app/app.component.spec.ts (934 bytes)
CREATE angularStock/src/app/app.component.ts (308 bytes)
CREATE angularStock/src/app/app.config.ts (227 bytes)
CREATE angularStock/src/app/app.routes.ts (77 bytes)
CREATE angularStock/src/assets/.gitkeep (0 bytes)
✓ Packages installed successfully.
Identidad del autor desconocido

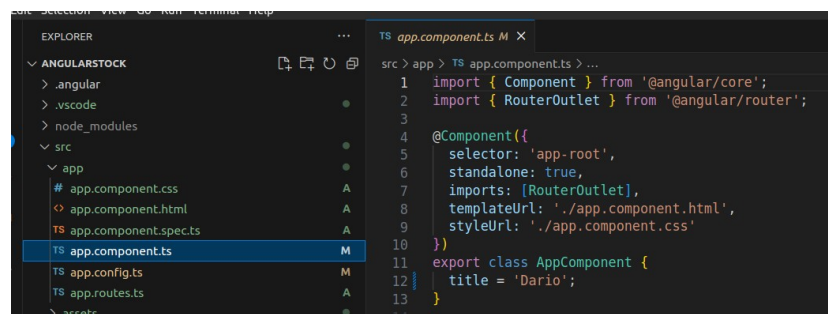
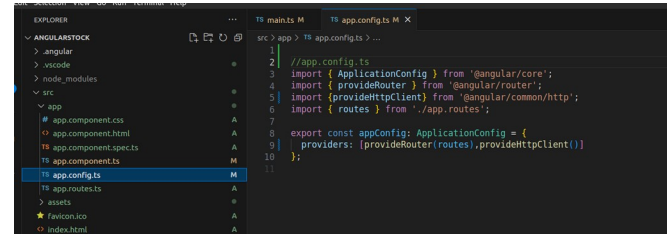
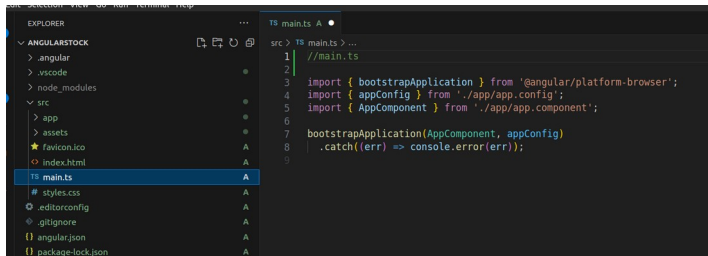
*** Por favor cuéntame quién eres.

Ejecuta
```

Antes de instalar los paquetes Angular nos pregunta que hojas de estilo queremos utilizar, elegimos css, luego nos pregunta si queremos 'server side rendering' y elegimos que no, a continuación se instalan los paquetes y tendremos los valores y la estructura por defecto que nos proporciona Angular y podremos ver una aplicación corriendo en nuestro navegador con el comando `ng serve`. En la última versión de Angular **desaparece** el módulo raíz, el archivo `app.module.ts`, los componentes se definen por defecto como `'standalone'` con el componente `app.component.ts` como raíz.

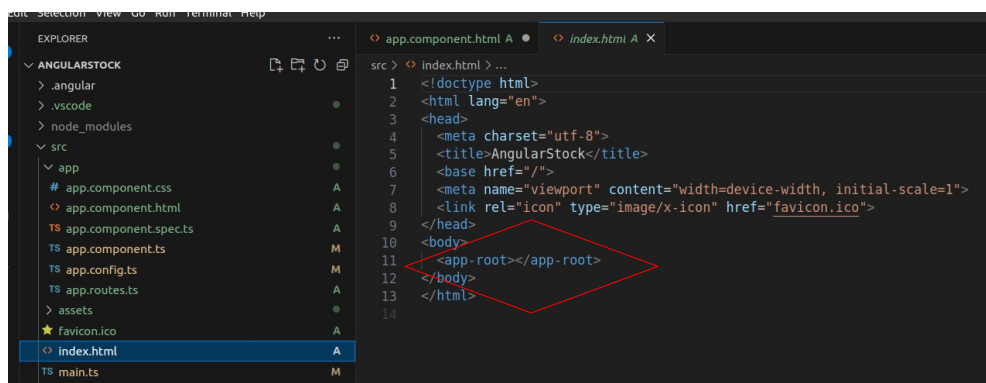


El archivo **main.ts** es el que arranca la aplicación importando las clases **AppComponent** (esta ubicada en **app.component.ts**) y el **appConfig**, (ubicada en **app.config.ts**), que nos proporcionara la configuración de los servicios que vamos a necesitar de manera global en nuestra aplicación y como ya sabemos que necesitaremos manejar rutas y peticiones http ,lo dejaremos configurado, **import {provideHttpClient} from '@angular/common/http'; import { routes } from './app.routes'.**



Observaremos que **app.component.ts** y todos los componentes que crearemos en el futuro están compuesto por el decorador **@Component** que se utiliza para definir un componente. Los componentes son bloques de construcción fundamentales en las aplicaciones Angular, los cuales encapsulan la lógica y la presentación de la interfaz de usuario de una parte específica de la aplicación. El decorador **@Component** toma un objeto de configuración como argumento. Este objeto contiene metadatos que describen el comportamiento y la apariencia del componente. Esto metadatos son:

selector: Identifica el componente en una plantilla HTML. Cuando Angular encuentra este selector en una plantilla, instanciará y renderizará el componente en ese lugar. En el caso de nuestro componente principal **app.component.ts** el selector es **'app-root'** y está insertado en **index.html**.



template o templateUrl: Define la plantilla HTML que representa la vista del componente. Puede ser una cadena de texto en línea o una ruta a un archivo HTML externo. Cada vez que creamos un componente se crea también este archivo, por eso vemos que en la estructura de arranque por defecto encontraremos el archivo **app.component.html**, que gracias al selector, terminara siendo renderizado en **index.html**, y siguiendo la logica de componentes padre/hijo renderizaremos todas las plantillas HTML que vayamos creando.

styles o styleUrls: Define los estilos CSS asociados con el componente. Pueden ser estilos en línea o rutas a archivos CSS externos. También se crea cuando creamos el componente

Imports[]: Aquí añadiremos los módulos externos declarados en la importaciones de cabecera que necesitamos para la lógica de la plantilla HTML de cada componente.

Por último encontramos el **'class component'** donde se ejecutará la lógica del componente y los valores y métodos que se pasaran a la plantilla HTML asociado a dicho componente.

2) Generación de componentes para datos de artículos y formulario para añadir nuevos valores.

Con el comando **'ng generate component <nombre del componente>'**, generaremos los siguientes componentes:

formulario.component.ts y sus asociados html y css.

```
▼ app
  ▼ formulario
    # formulario.component.css
    < formulario.component.html
    TS formulario.component.spec.ts
    TS formulario.component.ts
  ● dario@dario-GL553VD:~/Escritorio/angular/angularStock$ ng generate component formulario
    CREATE src/app/formulario/formulario.component.css (0 bytes)
    CREATE src/app/formulario/formulario.component.html (25 bytes)
    CREATE src/app/formulario/formulario.component.spec.ts (624 bytes)
    CREATE src/app/formulario/formulario.component.ts (250 bytes)
  ● dario@dario-GL553VD:~/Escritorio/angular/angularStock$ ng generate component lista-productos
    CREATE src/app/lista-productos/lista-productos.component.css (0 bytes)
```

lista-productos.component.ts y sus asociados html y css

```
▼ lista-productos
  # lista-productos.component.css
  < lista-productos.component.html
  TS lista-productos.component.spec.ts
  TS lista-productos.component.ts
  ● dario@dario-GL553VD:~/Escritorio/angular/angularStock$ ng generate component formulario
    CREATE src/app/formulario/formulario.component.css (0 bytes)
    CREATE src/app/formulario/formulario.component.html (25 bytes)
    CREATE src/app/formulario/formulario.component.spec.ts (624 bytes)
    CREATE src/app/formulario/formulario.component.ts (250 bytes)
  ● dario@dario-GL553VD:~/Escritorio/angular/angularStock$ ng generate component lista-productos
    CREATE src/app/lista-productos/lista-productos.component.css (0 bytes)
```

productos.component.ts y sus asociados html y css

```
▼ productos
  # productos.component.css
  < productos.component.html
  TS productos.component.spec.ts
  TS productos.component.ts
  ● dario@dario-GL553VD:~/Escritorio/angular/angularStock$ ng generate component productos
    CREATE src/app/productos/productos.component.css (0 bytes)
    CREATE src/app/productos/productos.component.html (24 bytes)
```

Luego de la creación de los componentes utilizaremos las etiquetas **<app-lista-productos></app-lista-productos>** **<app-productos></app-productos>** **<app-formulario> </app-formulario>**, para añadirlas a **app.component.html**, de esta forma indicamos en el HTML de nuestro componente principal que renderize los componentes que hemos creado, es decir que nos referimos a los HTML de los componentes utilizando el **'selector'** con la sintaxis **<selector></selector>**.

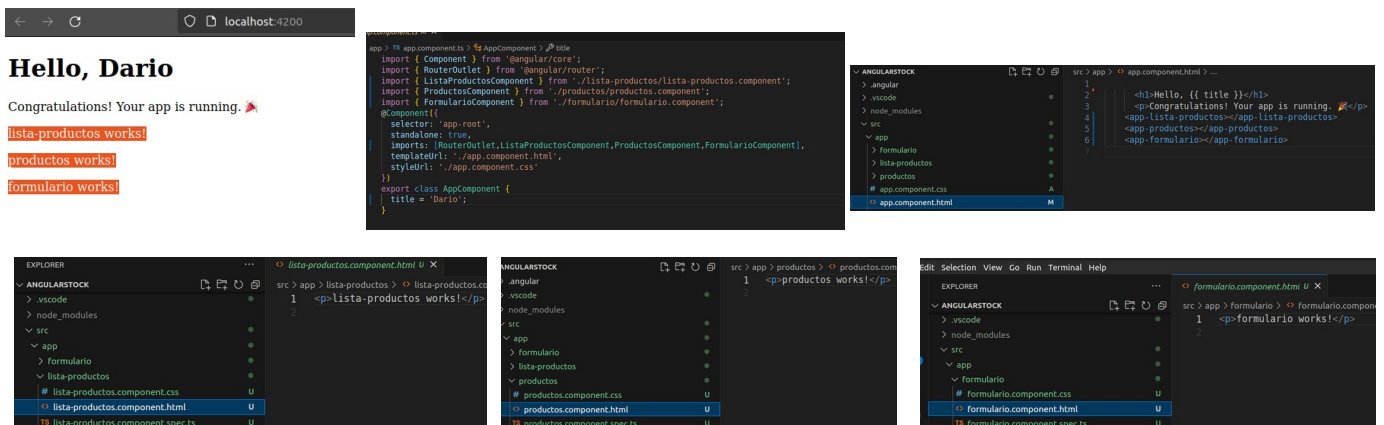
```
ANGULARSTOCK
src > app > lista-productos > TS lista-productos.component.ts > ...
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-lista-productos',
5   standalone: true,
6   imports: [],
7   templateUrl: './lista-productos.component.html',
8   styleUrls: ['./lista-productos.component.css']
9 })
10 export class ListaProductosComponent {
11 }
12
13 TS lista-productos.component.ts
```

```
ANGULARSTOCK
src > app > productos > TS productos.component.ts > ...
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-productos',
5   standalone: true,
6   imports: [],
7   templateUrl: './productos.component.html',
8   styleUrls: ['./productos.component.css']
9 })
10 export class ProductosComponent {
11 }
12
13 TS productos.component.ts
```

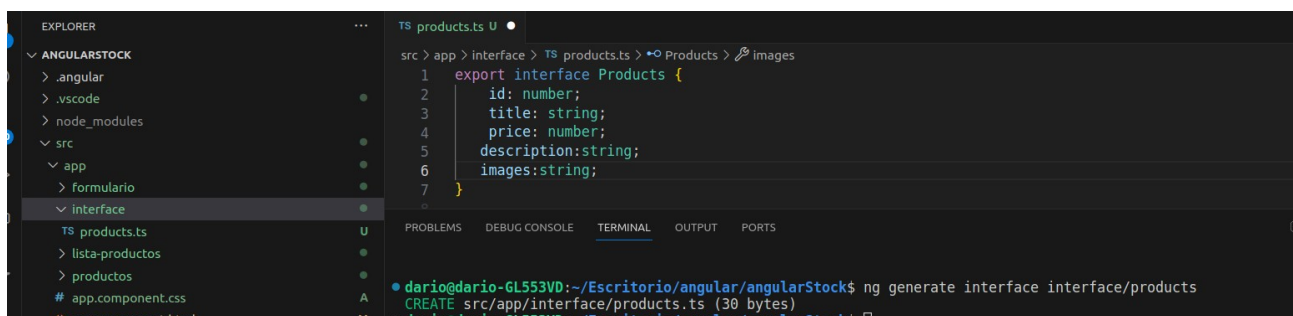
```
ANGULARSTOCK
src > app > formulario > TS formulario.component.ts > ...
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-formulario',
5   standalone: true,
6   imports: [],
7   templateUrl: './formulario.component.html',
8   styleUrls: ['./formulario.component.css']
9 })
10 export class FormularioComponent {
11 }
12
13 TS formulario.component.ts
```

Para la operación antes mencionada necesitamos antes hacer la importación de los componentes creados dentro **app.componet.ts**, escribimos entonces **import { nombredecomponente }**

from './directorio/archivodel componente' y luego lo añadimos dentro del array imports:[nombre del componente]. Comprobamos en el servidor con el comando `ng serve`.



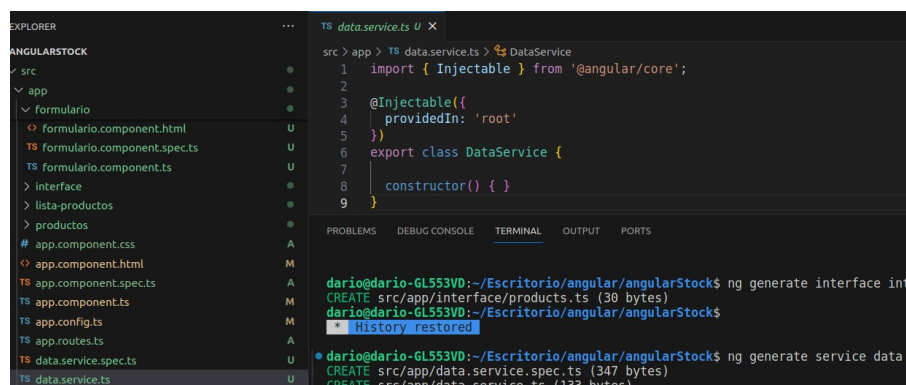
Creamos también una **'interface'** de Type Script que utilizaremos para el tipado del objeto que contendrá la información de nuestros artículos, ejecutamos el comando **ng generate interface archivo/nombre-de-interface**



3) Creación de servicio, rutas y lógica de componentes para mostrar y añadir artículos

Comenzaremos creando el ‘servicio’, este sera un archivo cuyo objetivo será realizar las peticiones HTTP al un back-end local, (el mismo que utilizamos en el ejercicio anterior pero con algunas modificaciones para que devuelva artículos de una tienda de ropa y accesorios, hemos copiado datos de ‘Platzi Fake Store’ para el array de datos que simula nuestra base de datos).

Utilizamos el comando `ng generate service <nombre>` que creara nuestro archivo de servicios llamado `DataService.ts`, `import { Injectable } from '@angular/core'`; permite que utilicemos la funcionalidad de inyección de dependencias utilizando el decorador `@Injectable` para marcar una clase como un servicio que puede ser inyectado en otras clases, Cuando utilizas el decorador `@Injectable({ providedIn: 'root' })`, le estamos indicando a Angular que este servicio estará disponible para ser inyectado en todos los componentes y servicios de nuestra aplicación Angular.



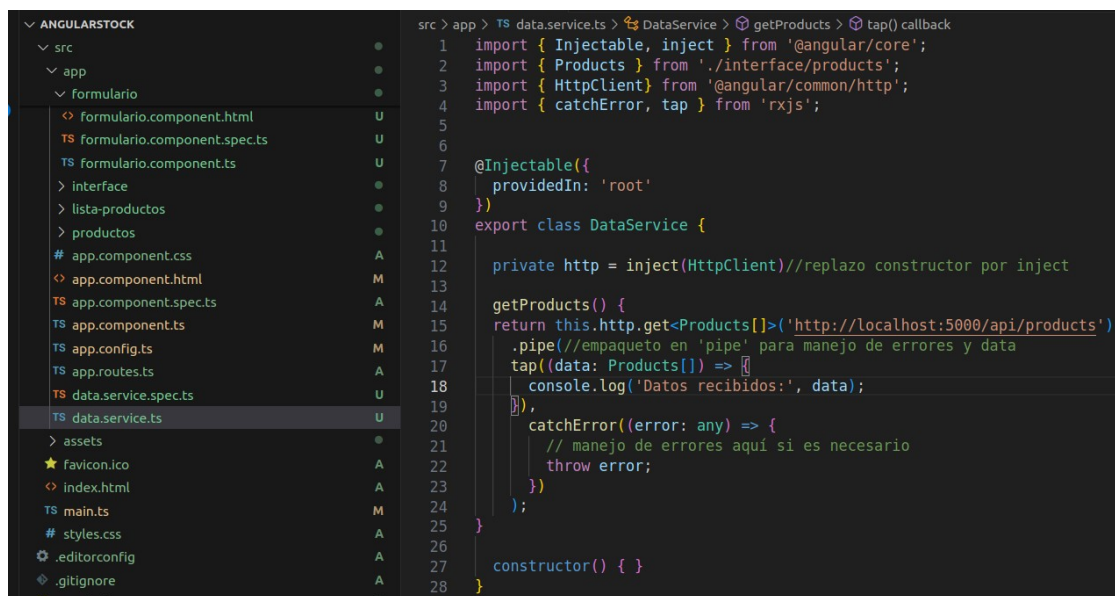
Para hacer que nuestro servicio ejecute las peticiones HTTP necesitaremos importar servicios y módulos que nos permitan hacerlo, aunque en Angular se puede utilizar `'fetch'` optaremos por usar el patrón `httpClient` que este framework pone a disposición y que para lo cual ya habíamos indicado su utilización cuando añadimos en `app.config.ts` (`provideHttpClient()`), entonces:

`import { Products } from './interface/products'`; importará la interface para el tipado de datos
`import { HttpClient } from '@angular/common/http'`; para poder realizar una instancia para implementar las peticiones.

Con la sintaxis `private http = inject(HttpClient)`, utilizamos la función `'inject()'` importada desde `'@angular/core'`, que nos permite realizar inyecciones de dependencias como hemos mencionado antes cuando mencionamos el decorador `@Injectable`, pero aquí se aplica a nuestra variable `http`, es decir que `inject()` sirve para aplicar el patrón de inyección de dependencias de Angular sobre una variable cada vez que necesitemos acceder a las funcionalidades proporcionadas por los servicios que hallamos creado o que el Angular nos proporcione, lo cual puede incluir métodos, propiedades y otros aspectos de la funcionalidad del servicio. Las inyecciones de dependencias también se pueden hacer en el `'constructor'` pero en la última versión de Angular ha añadido esta funcionalidad que simplifica el código.

Lo siguiente es escribir el código para la petición, creamos la función `'getProducts() {...}'`. Esta función se utilizara para realizar una solicitud HTTP GET a una API en el servidor local que devuelve una lista de productos. A diferencia de `'fetch'`, `'HttpClient'` no devuelve una promesa si no un `'observable'`, un observable es una fuente de datos que emite eventos o valores a lo largo del tiempo, y que permite a los `'subscriptores'` reaccionar a estos eventos de forma asíncrona. Los `'subscriptores'` son funciones o bloques de código que se pasan al método `subscribe()` del observable. Estas funciones o bloques de código definen cómo manejar los valores que emite el observable, ya sea para procesarlos, mostrarlos en la interfaz de usuario, realizar operaciones adicionales, etc.

Necesitamos añadir una nueva importación, `import { catchError, tap } from 'rxjs'`; estas funcionalidades nos permitirán encadenar el manejo de los datos como explicaremos mas adelante.



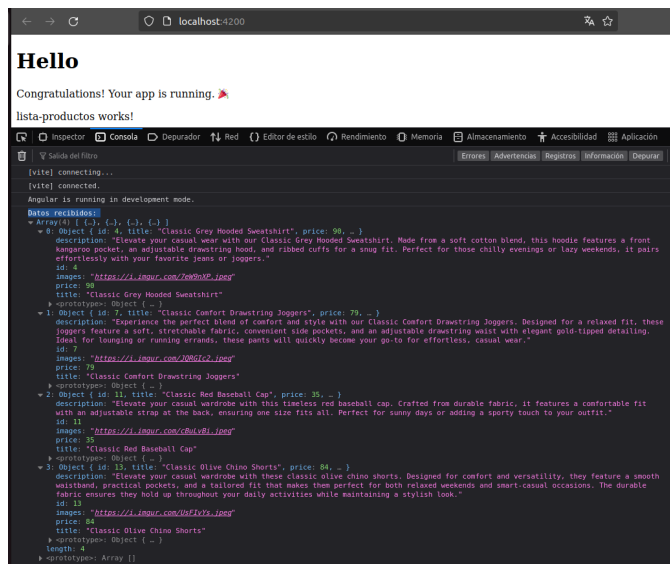
```
src > app > TS data.service.ts > DataService > getProducts > tap() callback
1  import { Injectable, inject } from '@angular/core';
2  import { Products } from './interface/products';
3  import { HttpClient } from '@angular/common/http';
4  import { catchError, tap } from 'rxjs';
5
6
7  @Injectable({
8    providedIn: 'root'
9  })
10 export class DataService {
11
12   private http = inject(HttpClient)//replazo constructor por inject
13
14   getProducts() {
15     return this.http.get<Products[]>('http://localhost:5000/api/products')
16       .pipe(//empaquetado en 'pipe' para manejo de errores y data
17         tap((data: Products[]) => {
18           console.log('Datos recibidos:', data);
19         }
20       ),
21       catchError((error: any) => {
22         // manejo de errores aqui si es necesario
23         throw error;
24       })
25     );
26   }
27   constructor() { }
28 }
```

`return this.http.get<Products[]>('http://localhost:5000/api/products');` Esta línea realiza una solicitud HTTP GET utilizando el servicio `HttpClient` de Angular (`this.http`). Se espera que la respuesta de la solicitud contenga un array de objetos de tipo `Products`. El método `get<Products[]>` es genérico y se utiliza para especificar el tipo de datos que se espera recibir en la respuesta. La URL a la que se hace la solicitud es `'http://localhost:5000/api/products'`.

.pipe(): La función **pipe()** se utiliza para encadenar operadores de manipulación de observables. En este caso, se utiliza para aplicar operadores adicionales a la secuencia de observables que devuelve la solicitud HTTP.

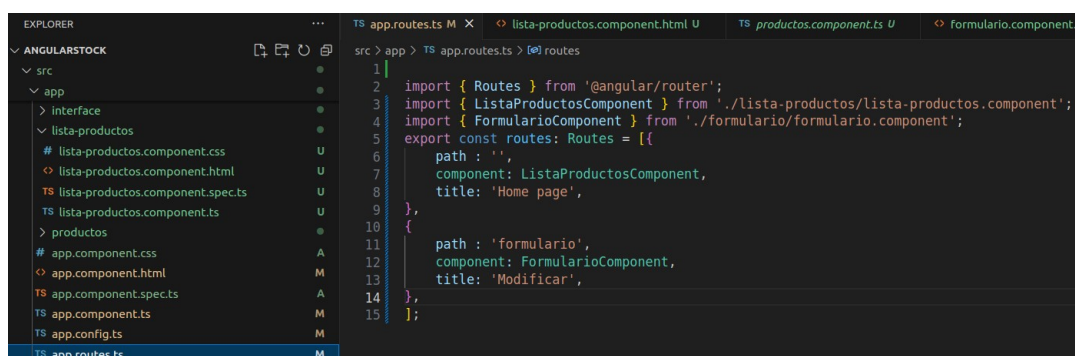
tap((data: Products[]) => { console.log('Datos recibidos:', data); }): El operador **tap()** permite realizar efectos secundarios en los datos emitidos por el observable sin alterarlos. En este caso, cada vez que se reciben datos (productos) en la respuesta, se imprime un mensaje en la consola con la información recibida.

catchError((error: any) => { throw error; }): El operador **catchError()** se utiliza para manejar errores que puedan ocurrir durante la solicitud HTTP. En este caso, si hay algún error durante la solicitud (por ejemplo, problemas de red o errores en el servidor), se lanzará una excepción (**throw error**) para que pueda ser manejada en el código que llama a esta función. Para testear si la petición funciona hemos añadido una llamada al servicio en **app.component.ts** para que cuando la aplicación se inicie imprima los datos en consola. Aquí no explicaremos este proceso, será detallado mas adelante cuando nos dispongamos a renderizar los datos ya que este bloque de código sera eliminado.



```
7 import { DataService } from './data.service';
8
9 @Component({
10   selector: 'app-root',
11   standalone: true,
12   imports: [RouterOutlet, ListaProductosComponent, ProductosComp
13   templateUrl: './app.component.html',
14   styleUrls: ['./app.component.css']
15 })
16 export class AppComponent {
17
18   private dataService: DataService = inject(DataService)
19   ngOnInit(): void {
20     this.dataService.getProducts().subscribe()
21   }
22 }
23
```

Antes de continuar añadiendo lógica en los componentes vamos a definir las rutas, recordemos que ya tenemos configurado en **app.config.ts** el 'provider' (**provideRouter(routes)**) que nos permite ejecutar las funcionalidades de Angular **Router** en nuestra aplicación. Tenemos que abrir el archivo **app.routes.ts**, (se crea automáticamente cuando generamos el proyecto). En este archivo se importa la interfaz llamada **Routes** desde el módulo **@angular/router**. Esta interfaz se utiliza para definir la estructura de las rutas en una aplicación Angular, también importamos los componentes que necesitemos desde sus respectivos archivos.



En el código definimos un array de objetos que representan las configuraciones de las rutas en una aplicación Angular. Cada objeto en el array, (del tipo **Routes**), corresponde a una ruta específica y contiene las siguientes propiedades:

path: Define la URL relativa de la ruta.

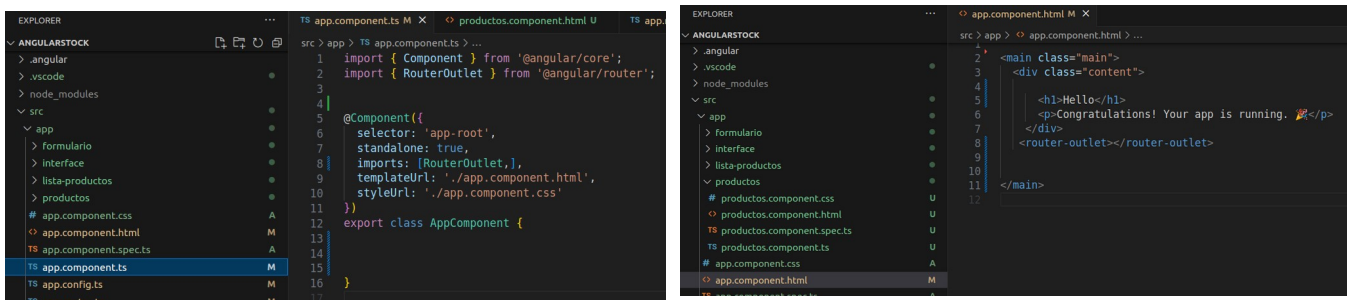
component: Especifica el componente que se cargará cuando se acceda a esta ruta.

title: Puede ser un título opcional asociado a la ruta.

Hemos definido que para la ruta raíz **'/'** se renderize el componente **'ListaProductosComponent'**

mientras que para la ruta **'/formulario'** renderize **'FormularioComponent'**.

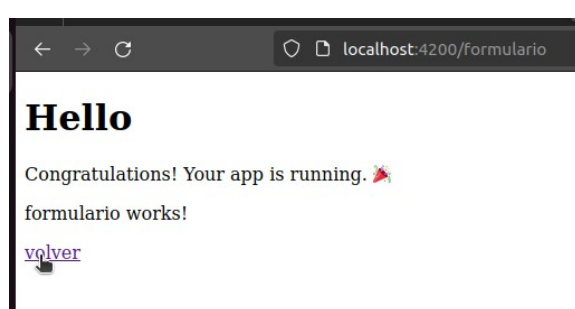
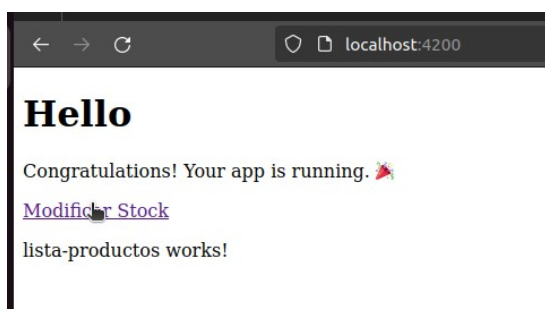
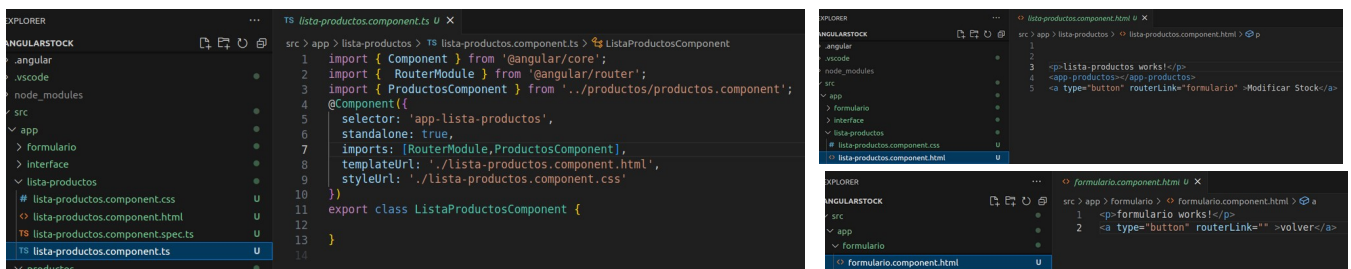
Modificamos **app.component.ts** eliminando el código que quede inutilizable, básicamente solo nos quedara la importación de la clase **RouterOutlet** proporcionada por el módulo de enrutamiento de Angular (**@angular/router**), esto nos da acceso a utilizar la etiqueta **<router-outlet></router-outlet>** en la plantilla del componente, **app.component.html**, esta etiqueta html actúa como marcador de salida del enrutamiento rederizando dinámicamente el componente que corresponda según la ruta que se indique en la url según nuestra configuración en **app.routes.ts**



La idea es que en la raíz de nuestra app se renderize **ListaProductosComponent** y su componente hijo **ProductosComponent**, para esto en **lista-productos.component.html** añadimos la etiqueta html que hace referencia a **ProductosComponent**, **<app-productos></app-productos>** mientras que en **ListaProductosComponent.ts** importamos **ProductosComponent** y **RouterModule**, el primero permite que **ListaProductosComponent** haga uso de **ProductosComponent**, la segunda importación nos permite utilizar las herramientas de enrutamiento de Angular, entonces añadimos por medio de un **<a>** (en **lista-productos.component.html**), un botón con la propiedad **'routerLink'** que al hacer click nos dirija a formulario, (**Modificar Stock**)

También modificamos **formularios.component.html**, de momento solo añadimos otro **<a>** que nos devuelva a la pagina de inicio, mas a delante lo modificaremos a para que podamos añadir un nuevo artículo y la navegación programática.

Testeamos en el navegador para ver como funciona el enrutamiento.



Ya con las rutas funcionando definimos en 'lista-productos.ts' dentro de la clase 'ListaProductosComponente' declaramos una array vacía 'protected listaProductos: Products[]=[]', que recibirá datos del tipo Products y que por ser 'protected' podrá ser accesible desde la clase hija ya que este componente pasara los datos a su componente hijo 'ProductosComponent' para que mediante la directiva *ngfor, (que es el equivalente en Angular a un 'for loop') renderize la data recibida desde 'DataService' como explicaremos a continuación. También declaramos una variable 'dataService' a la cual le inyectamos la dependencia 'DataService', private dataService: DataService = inject(DataService).

import { Component, inject } from '@angular/core'; Component viene por defecto ya que nos permite utilizar el @Component({..}) para configurar la metadata del componente, y añadimos 'inject' para que podamos utilizar la función inject(), para la inyección de dependencias.

import { ProductosComponent } from '../productos/productos.component'; con esta importación nuestro componente reconoce la existencia de 'ProductosComponent', debemos añadirlo también en el array 'imports' de @Component.

import { Products } from './interface/products'; importará la interfase para el tipado de datos.

import { DataService } from './data.service'; nos dará acceso a las funcionalidades de nuestro servicio creado en 'data.services.ts'

import { CommonModule } from '@angular/common'; Directivas comunes, se importan para exportar directivas comunes como NgIf, NgFor hacia nuestro html donde la aplicaremos sobre <pp-productos></app-productos>, debemos añadirlo también en el array 'imports' de @Component.

import { RouterModule } from '@angular/router'; proporciona las directivas y los servicios necesarios para configurar el enrutamiento, necesitaremos utilizar la propiedad 'routerLink'.

El método ngOnInit(...) se ejecuta cada vez que el componente es inicializado. Es donde realizamos la llamada a la API mediante la subscripción a 'DataService' que es el emisor del evento observable.

```
> app > lista-productos > TS lista-productos.component.ts > ListaProductosComponent
1 import { Component, inject } from '@angular/core';
2 import { RouterModule } from '@angular/router';
3 import { ProductosComponent } from '../productos/productos.component';
4 import { Products } from './interface/products';
5 import { DataService } from './data.service';
6 import { CommonModule } from '@angular/common';
7 @Component({
8   selector: 'app-lista-productos',
9   standalone: true,
10  imports: [RouterModule, ProductosComponent, CommonModule],
11  templateUrl: './lista-productos.component.html',
12  styleUrls: ['./lista-productos.component.css']
13 })
14 export class ListaProductosComponent {
15
16   protected listaProductos: Products[]=[]
17   private dataService: DataService = inject(DataService)
18
19   ngOnInit(): void {
20
21     this.dataService.getProducts().subscribe(next:(data)=> {
22       this.listaProductos = data
23     },
24     error: (err: any) => {console.error('Ocurrió un error al obtener los productos:', err);
25       alert('Ocurrió un error al obtener los productos')}}
26   )
27 }
28 }
```

this.dataService.getProducts().subscribe(.....), aquí indicamos que vamos a llamar a la función 'getProducts()' de 'dataService' y lo encadenamos con el método subscribe() que nos permitirá manipular los datos. Dentro del método subscribe(), hay un objeto con dos propiedades: next y error. Estas son las funciones de devolución de llamada que manejan la respuesta exitosa y la respuesta de error respectivamente.

En la función de devolución de llamada next:(data)=>{ this.listaProductos = data }, asigna a listaProductos: Products[]=[] los datos recibidos desde el back-end, (el evento observable).

En caso de un error, la función de devolución de llamada 'error' registra el error en la consola y muestra una alerta al usuario.


```
TS productos.component.ts U X
src > app > productos > TS productos.component.ts > ...
1 import { Component, Input } from '@angular/core';
2 import { Products } from '../interface/products';
3
4
5 @Component({
6   selector: 'app-productos',
7   standalone: true,
8   imports: [],
9   templateUrl: './productos.component.html',
10  styleUrls: ['./productos.component.css']
11 })
12 export class ProductosComponent {
13   @Input() products!: Products
14 }
15
```

```
FLORER
GULARSTOCK
src > lista-productos > TS lista-productos.component.ts > TS ListaProductosComponent
1 import { Component, inject } from '@angular/core';
2 import { RouterModule } from '@angular/router';
3 import { ProductosComponent } from '../productos/productos.component';
4 import { Products } from '../interface/products';
5 import { DataService } from '../data.service';
6 import { CommonModule } from '@angular/common';
7 @Component({
8   selector: 'app-lista-productos',
9   standalone: true,
10  imports: [RouterModule, ProductosComponent, CommonModule],
11  templateUrl: './lista-productos.component.html',
12  styleUrls: ['./lista-productos.component.css']
13 })
14 export class ListaProductosComponent {
15   protected listaProductos: Products[] = []
16   private dataService: DataService = inject(DataService)
17
18   ngOnInit(): void {
19     this.dataService.getProducts().subscribe(next: (data) => {
20       this.listaProductos = data
21     },
22     error: (err: any) => { console.error('Ocurrió un error al obtener los productos:', err);
23       alert('Ocurrió un error al obtener los productos') })
24   }
25 }
26
27 assets
28 favicon.ico
29 index.html
30
```

The screenshot shows the VS Code interface with the Explorer panel on the left and the Source Control panel on the right. The Explorer panel shows the file structure of a project named 'productos'. The Source Control panel shows the changes made to 'productos.component.html'. The code in the editor shows the HTML structure for the product list, including a header, a list of products, and a footer.

The image shows a VS Code editor with two panels. The left panel is the Explorer view, showing the file structure of a project. The right panel is the Source Control view, showing the diff for the file 'lista-productos.component.html'.

Explorer View (Left):

- ANGULARSTOCK
 - src
 - app
 - formulario
 - formulario.component.ts
 - interface
 - lista-productos
 - lista-productos.component.css
 - lista-productos.component.html
 - lista-productos.component.spec.ts

Source Control View (Right):

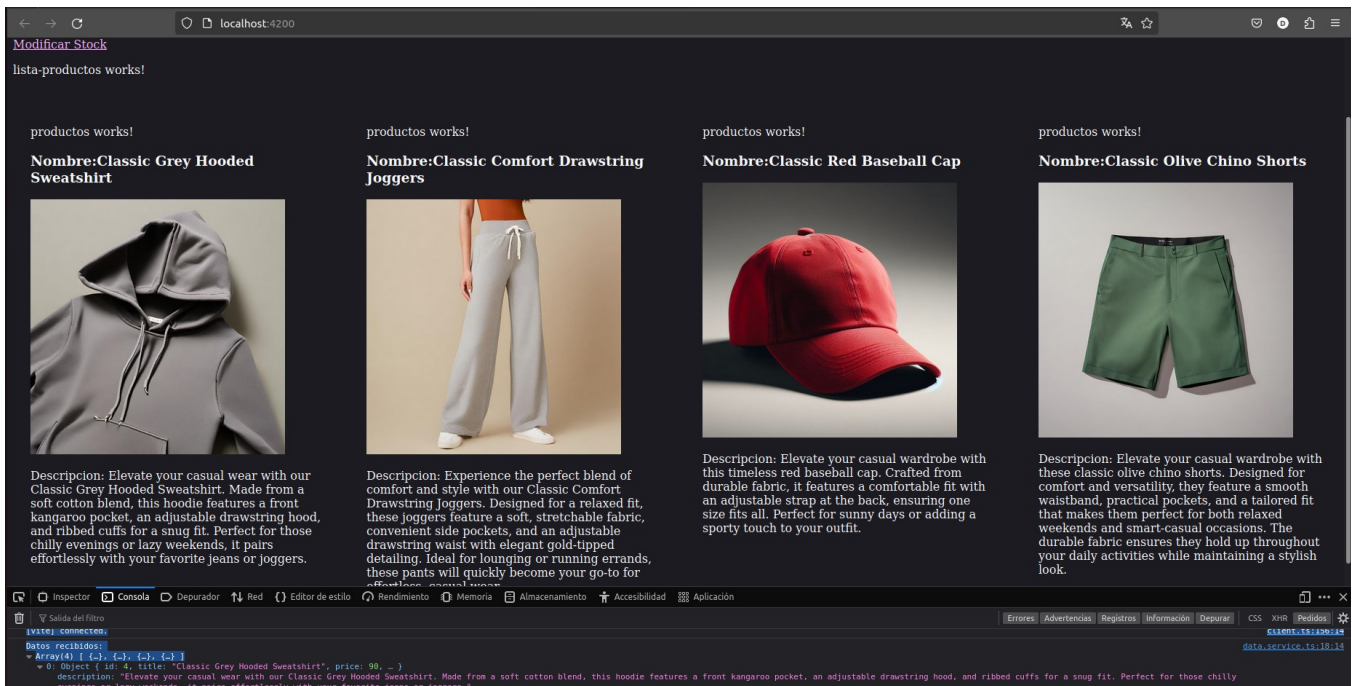
The diff shows changes to 'lista-productos.component.html'. Line 7 is highlighted, showing the addition of the <app-productos> tag with *ngFor and [products] attributes.

```

1
2
3 <a type="button" routerLink="formulario" >Modificar Stock</a>
4 <p>lista-productos works!</p>
5 <section class="results">
6   <app-productos *ngFor="let products of listaProductos"
7   [products]="products" ></app-productos>
8 </section>
9
10

```

Hemos añadido un poco de código css para una mejor visualización, serán explicado mas adelante después de finalizar con las funcionalidades del proyecto, escribimos en nuestra terminal `ng serve` y comprobamos en el navegador.



Ahora necesitamos añadir la lógica para poder añadir un nuevo artículo a nuestra lista y que después de enviar el formulario la navegación programática nos devuelva a la pagina de inicio. Primero necesitamos incluir en nuestro archivo `'data.services.ts'` la solicitud POST para enviar el artículo a la base de datos.

Creamos la función `'addProducts()'` que toma un parámetro `'body'` de tipo `'Products'`. La función realiza una solicitud para agregar productos a la URL de destino es `'http://localhost:5000/api/products'`.

```
addProducts(body: Products) { return this.http.post('http://localhost:5000/api/products', body)
  .pipe(
    catchError((error: any) => {
      // manejo de errores aquí si es necesario
      throw error;
    })
  );
}
```

El método `'post()'` que toma un parámetro `'body'` de tipo `'Products'` y devuelve un `'Observable'` que representa la respuesta de la solicitud HTTP. El operador `'pipe()'` se utiliza para encadenar operaciones observables. En este caso, se utiliza para encadenar una función de manejo de errores utilizando el operador `'catchError()'`.

Abrimos ahora `formulario.component.ts` y realizamos las importaciones:

`import { Component, inject } from '@angular/core';` `Component` viene por defecto ya que nos permite utilizar el `@Component({..})` para configurar la metadata del componente y añadimos `'inject'` para que podamos utilizar la función `inject()`, para la inyección de dependencias.

`import { Products } from './interface/products';` importará la interfase para el tipado de datos.

`import { DataService } from './data.service';` nos dará acceso a las funcionalidades de nuestro servicio creado en `'data.services.ts'`

`import { Router, RouterModule } from '@angular/router';` `'Router'` es una clase que proporciona las funcionalidades para la navegación entre las diferentes vistas o componentes de una aplicación

Angular. Permite la navegación programática a través de la aplicación, lo que significa que podemos dirigir a los usuarios a diferentes vistas en función de acciones específicas, como en nuestro caso al hacer clic en un botón para enviar el formulario. **'RouterModule'** es un módulo que proporciona las directivas y los servicios necesarios para configurar el enrutamiento. Este módulo es esencial para establecer la configuración de enrutamiento de la aplicación y definir las rutas que corresponden a diferentes componentes. Al importar **'Router'** y **'RouterModule'** estamos preparando la aplicación para manejar la navegación entre diferentes vistas o componentes. Estos elementos son fundamentales para construir aplicaciones de una sola página (SPA), donde la navegación se realiza sin recargar la página completa.

import { FormControl, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms'; Estos importes son fundamentales para trabajar con formularios reactivos en Angular. Permiten una gestión más avanzada de la validación, el seguimiento del estado del formulario y el manejo de los cambios en los valores de los campos.

'FormControl': es una clase que se utiliza para rastrear el valor y el estado de un control de formulario HTML individual, como un campo de entrada de texto. Los controles de formulario se utilizan para construir formularios reactivos en Angular.

'FormGroup': es una clase que representa un grupo de controles de formulario. Se utiliza para agrupar varios controles de formulario relacionados dentro de un formulario más grande.

'ReactiveFormsModule': Este módulo proporciona soporte para la construcción de formularios reactivos en Angular. Para utilizar los formularios reactivos, necesitas importar este módulo en tu aplicación.

'Validators': es una clase que proporciona varios validadores predefinidos que se pueden utilizar para validar los valores de los controles de formulario. Estos validadores se utilizan para realizar validaciones como requerido, mínimo, máximo, longitud, expresiones regulares, etc.

import { CommonModule } from '@angular/common'; Directivas comunes, se importan para exportar directivas comunes como **NgIf**, **NgFor**, entre otras, que se utilizan ampliamente en las plantillas de componentes Angular (archivos html) para controlar el flujo y la presentación de datos.

```
TS formulario.component.ts U X
src > app > formulario > TS formulario.component.ts > FormularioComponent
1 import { Component, inject } from '@angular/core';
2 import { Router, RouterModule } from '@angular/router';
3 import { DataService } from '../data.service';
4 import { FormControl, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms';
5 import { Products } from '../interface/products';
6 import { CommonModule } from '@angular/common';
7
8 @Component({
9   selector: 'app-formulario',
10  standalone: true,
11  imports: [RouterModule, ReactiveFormsModule, CommonModule],
12  templateUrl: './formulario.component.html',
13  styleUrls: ['./formulario.component.css']
14 })
```

En el array de la propiedad **'imports'** dentro de **@Component** añadimos **'RouterModule', 'ReactiveFormsModule, CommonModule'**, para que la plantilla html reconozca los módulos de la importación cabecera que serán manejados allí.

Dentro de **class FormularioComponent {...}**, declaramos las variables **'dataService'** y **'router'** inyectándole las correspondientes dependencias, (**private dataService: DataService = inject (DataService)** **private router: Router = inject(Router)**). Otra variable **'formSubmitted: boolean = false;'** de tipo booleano se usa para realizar un seguimiento de si el formulario se ha enviado o no, una última variable **'id: number = 0;'** se usará para asignar un identificador único a los nuevos productos que se agregarán, requisito necesario para el cuerpo de la solicitud POST que hemos diseñado, este valor será actualizado dentro del método **ngOnInit()** donde se llama al método

'getProducts()' del servicio 'dataService', utilizamos la longitud de la lista de productos obtenida 'data.length' más 1. Esto se hace para asegurar que 'id' sea único para el próximo producto que se agregará. (En el array que simula una base de datos en nuestro back-end, los id de la data que usamos como ejemplo en el primer artículo es 1, el segundo, 2 etc.)

```
export class FormularioComponent {
  private dataService:DataService = inject(DataService)
  private router: Router=inject(Router);
  formSubmitted: boolean = false;
  id: number = 0;

  ngOnInit(): void {

    this.dataService.getProducts().subscribe({next:(data)=> {
      this.id= data.length + 1
    }})
  }
}
```

Se declara una variable 'addForm' que es una instancia de 'FormGroup'. 'FormGroup' es una clase que representa un grupo de controles de formulario relacionados. Este bloque de código crea un formulario reactivo en Angular utilizando la clase 'FormGroup' y los controles 'FormControl'. En este caso, 'addForm' contendrá los controles para el formulario de agregar productos. Dentro de 'addForm', se definen varios controles utilizando la clase 'FormControl'. Cada control representa un campo específico del formulario. Los campos incluidos son:

title: Un control para el título del producto. Se inicializa con un valor vacío y se valida como obligatorio (Validators.required).

price: Un control para el precio del producto. Se inicializa con un valor null y se valida como obligatorio (Validators.required).

description: Un control para la descripción del producto. Se inicializa con un valor vacío y se valida como obligatorio (Validators.required).

images: Un control para las imágenes del producto. Se inicializa con un valor vacío y se valida como obligatorio (Validators.required).

Cada control se crea utilizando la clase 'FormControl', que rastrea el valor y el estado del campo de entrada HTML correspondiente. Se pueden aplicar validadores a cada control utilizando el segundo parámetro del constructor de 'FormControl'. Cada campo tiene sus propias validaciones definidas.

```
export class FormularioComponent {
  private dataService:DataService = inject(DataService)
  private router: Router=inject(Router);
  formSubmitted: boolean = false;
  id: number = 0;

  ngOnInit(): void {

    this.dataService.getProducts().subscribe({next:(data)=> {
      this.id= data.length + 1
    }})
  }
}

addForm = new FormGroup(
  {
    //id :new FormControl(null,Validators.required),
    title: new FormControl('',Validators.required),
    price: new FormControl(null,Validators.required),
    description: new FormControl('',Validators.required),
    images: new FormControl('',Validators.required),
  }
)
```


Definimos el método `addNewProduct()` que será llamado cuando hagamos click en el botón de envío de el formulario.

`this.formSubmitted = true;`: Esta línea establece la variable `formSubmitted` en `true`, lo que indica que el formulario ha sido enviado.

`if (this.addForm.valid) { ... }`: Aquí se comprueba si el formulario (`addForm`) es válido. Si el formulario es válido, se procede con la lógica dentro del bloque `if`.

Dentro del bloque `if`, se crea un objeto `body` que contiene los datos del producto a agregar. Estos datos se obtienen del formulario utilizando `this.addForm.value.id`, `'title'`, `'price'`, `'description'`, y `'images'` que son los campos del formulario.

Se llama al método `addProducts(body)` del servicio `dataService` que realiza una solicitud HTTP POST para enviar los datos del producto al servidor.

`Subscribe(...)` suscribe al observable devuelto por `addProducts(body)`. Dentro de la suscripción, hay dos funciones de devolución de llamada: `'next:'`, función que se ejecuta cuando la solicitud es exitosa. Muestra un mensaje de confirmación y `navega automáticamente` al componente raíz (`this.router.navigate([''])`), `'error:'`, la función se ejecuta y si hay algún error durante la solicitud, muestra un mensaje de alerta de error y también navega de forma programática al componente raíz. Por lo tanto, después de enviar el formulario y volver a la raíz que es donde se renderiza `'ListaProductosComponen'` se ejecutará `ngOnInit()` nuevamente y se volverá a llamar a la API para obtener los productos actualizados. Esto actualizará el array `'listaProductos'` que tenemos en nuestro componente `'ListaProductosComponen'` con los datos más recientes.

```
app > formulario > 14 formulario.component.ts > 15 formulario.component > 16 addNewProduct > 17 body
export class FormularioComponent {
  addForm = new FormGroup({
  })
  addNewProduct(){
    this.formSubmitted = true;
    if (this.addForm.valid){
      let body : Products = {
        "id":this.id,
        "title": this.addForm.value.title!,
        "price": this.addForm.value.price!,
        "description":this.addForm.value.description!,
        "images": this.addForm.value.images!
      }
      this.dataService.addProducts(body).subscribe({
        next:(data)=>(console.log(data),alert('Articulo añadido!!'), this.router.navigate([''])),
        error: (err: any) => (console.error(err), alert('algo salio mal!!'), this.router.navigate(['']))
      })
    }
  }
}
```

En resumen, el comportamiento que obtendremos es el resultado esperado del ciclo de vida de los componentes en Angular y del uso de `ngOnInit()` para inicializar el componente y obtener los datos necesarios.

Por ultimo necesitamos enlazar esta lógica con la plantilla html del componente, nos referimos a `formulario.componet.html`. Utilizaremos `'event binding'` usando `'()'` y `'property binding'` con `'[]'`.

```
formulario.component.html U
src > app > formulario > formulario.component.html > ...
1
2 |
3 <form action="" [formGroup]="addForm" (submit)="addNewProduct()">
4
5 <label for="product-title">Nombre</label>
6 <input type="text" formControlName="title">
7
8 <label for="product-price">Precio</label>
9 <input type="number" formControlName="price" >
10
11 <label for="product-description">Descripcion</label>
12 <input type="text" formControlName="description">
13
14
15 <label for="product-images">img URL</label>
16 <input type="text" formControlName="images">
17
18
19 <button class="" type="submit" >buscar</button>
20 </form>
21
22 <div *ngIf="formSubmitted && !addForm.valid" class="error-message">
23 | Por favor, complete el formulario correctamente.
24 </div>
25 <a type="button" routerLink="" >volver</a>
```

`<form action="" [formGroup]="addForm" (submit)="addNewProduct()">`: Esta etiqueta `<form>` define un formulario HTML. La propiedad `[formGroup]` está vinculada al formulario reactivo `'addForm'` definido en el componente. El evento `(submit)` está vinculado al método `addNewProduct()` del componente, que se activará cuando se envíe el formulario.

`'formControlName'`: Esta directiva se utiliza para vincular cada campo de entrada HTML (`<input>`) con el control correspondiente definido en el formulario reactivo. Esto permite que Angular rastree y valide los valores de los campos del formulario.

`*ngIf="formSubmitted && !addForm.valid"`: Esta **directiva estructural `*ngIf`** se utiliza para mostrar un mensaje de error si el formulario ha sido enviado (**formSubmitted es true**) pero no es válido (**!addForm.valid**). Esto significa que si el formulario se ha enviado pero no es válido según las reglas de validación definidas en el formulario reactivo, se mostrará un mensaje de error.

`volver`: Este enlace `<a>` se utiliza para proporcionar una opción para volver atrás. El atributo `routerLink` se usa para especificar la ruta a la que se debe navegar cuando se hace clic en el enlace. En este caso, es un string vacío, lo que significa que al hacer clic en este enlace, se navegará a la ruta raíz.

4) Redefinimos las plantillas HTML y añadimos estilos CSS

Para mejorar la UI, hemos modificado las plantillas html eliminando algunos elementos y añadiendo otros, terminamos de eliminar las etiquetas que Angular crea por defecto en las plantillas, añadido la etiqueta `<header>` para los encabezados y modificado el formulario reemplazando el `'<input>'` para la descripción de los artículos por `<textarea>`.

```
formulario.component.html U X
src > app > formulario > formulario.component.html > ...
1
2
3 <header>
4   <h1>Ingresar detalles <span> <br> artículo</span></h1>
5 </header>
6 <form action="" [formGroup]="addForm" (submit)="addNewProduct()"
7   class="shared-styles">
8
9   <label for="product-title">Nombre</label>
10  <input type="text" formControlName="title">
11
12  <label for="product-price">Precio</label>
13  <input type="number" formControlName="price" >
14
15  <label for="product-description">Descripción</label>
16  <textarea formControlName="description"></textarea>
17
18
19  <label for="product-images">img URL</label>
20  <input type="text" formControlName="images">
21
22
23  <button class="" type="submit" >Añadir</button>
24 </form>
25
26
27 <div *ngIf="formSubmitted && !addForm.valid" class="error-message">
28   Por favor, complete el formulario correctamente.
29 </div>
30 <a type="button" routerLink="" >volver</a>
```

```
formulario.component.html U X
src > app > formulario > formulario.component.html > form.shared-styles > label
1
2
3 <header>
4   <h1>Ingresar detalles <span> <br> artículo</span></h1>
5 </header>
6 <form action="" [formGroup]="addForm" (submit)="addNewProduct()"
7   class="shared-styles">
8
9   <label for="product-title">Nombre</label>
10  <input type="text" formControlName="title">
11
12  <label for="product-price">Precio</label>
13  <input type="number" formControlName="price" >
14
15  <label for="product-description">Descripción</label>
16  <textarea formControlName="description"></textarea>
17
18
19  <label for="product-images">img URL</label>
20  <input type="text" formControlName="images">
21
22
23  <button class="" type="submit" >Añadir</button>
24 </form>
25
26
27 <div *ngIf="formSubmitted && !addForm.valid" class="error-message">
28   Por favor, complete el formulario correctamente.
29 </div>
30 <a type="button" routerLink="" >volver</a>
```

```
app.component.html M X
src > app > app.component.html > main.main > route
1
2 <main class="main">
3   <router-outlet></router-outlet>
4 </main>
5
```

```
productos.component.html U
src > app > productos > productos.component.html > article
1
2
3 <article>
4   <h1>{{products.title}}</h1>
5   <img [src]="products.images" alt="foto del producto"
6     <{{products.title}}>
7   <h3>{{products.price}}</h3>
8   <p> {{products.description}}</p>
9 </article>
10
```

Los siguientes estilos son aplicados de forma global en el archivo `styles.css` que se encuentra en la raíz de nuestro proyecto:

`img{width: 350px;}`: Establece el ancho máximo de las imágenes en 350 píxeles.

`:root{color-scheme: light dark ;}`: Define el esquema de color para la página, permitiendo modos claro y oscuro.

`h1{text-align: center;};` Centra la alineación del texto de los elementos `<h1>`.

`a{position: absolute; right: 5%; top: 10px;};` Posiciona los elementos de `<a>` de forma absoluta, un 5% desde la derecha y 10 píxeles desde la parte superior de su elemento contenedor.

`form{text-align: center; padding: 10px;};` Centra la alineación del texto dentro de los elementos `<form>` y agrega 10 píxeles de relleno alrededor del contenido.

```
# styles.css M
src > # styles.css > ...
1  /* Estilos globales */
2  img{width: 350px;}
3
4  :root{color-scheme: light dark ;}
5
6  h1{text-align: center; }
7
8  a{position: absolute;
9    right: 5%;
10   top: 10px;
11  }
12
13  form{text-align: center;
14    padding: 10px;
15  }
16
```

```
# styles.css M
src > # styles.css > {} @media screen and (max-width: 700px) > % h1
17
18 .shared-styles {
19   display: grid;
20   column-gap: 14px;
21   row-gap: 14px;
22   margin-top: 50px;
23   justify-content: space-around;
24 }
25
26 .results {
27   grid-template-columns: repeat(auto-fill,
28     minmax(400px, 400px));
29 }
30
31 button{ margin-top: 10px;
32   width: fit-content;
33   margin: 0 auto;}
34
35 label,input{margin: 3px;}
36 @media screen and (max-width: 700px) {
37   h1 {
38     font-size: 1.5em;
39   }
40   a{font-size: smaller}
41 }
42 .error-message{text-align: center;
43   color: red; }
```

`.shared-styles {display: grid; column-gap: 14px; row-gap: 14px; margin-top: 50px; justify-content: space-around;};` Esto define un conjunto de estilos para elementos con la clase `shared-styles`. Los establece como una cuadrícula con espacios, de columna (`column-gap`), y fila (`row-gap`) específicos, margen (`margin-top`) y justificación (`justify-content: space-around`) para que se distribuyan el espacio sobrante en el contenedor de manera uniforme alrededor de los elementos lo que significa que los elementos, dentro de la cuadrícula tendrán un espaciado equitativo entre ellos y desde los bordes del contenedor.

```
<form action="" [formGroup]="addForm" (submit)="addNewProduct()"
  class="shared-styles">
```

`.results {grid-template-columns: repeat(auto-fill, minmax(400px, 400px));};` Esta parte define el patrón de repetición de las columnas (`auto-fill`) indicando que la cuadrícula debería crear tantas columnas como sea posible para llenar su contenedor, y `minmax(400px, 400px)` establece el ancho mínimo y máximo de cada columna. En este caso, cada columna tendrá un ancho mínimo y máximo de 400 píxeles. Este marcador de clase es el complemento a `.shared-styles`. Dentro de `lista-productos.component.html` usamos una combinación de ambos estilos, mientras que en `formulario.component.html` solo utilizamos `.shared-styles`.

```
<section class="results shared-styles">
  <app-productos *ngFor="let products of listaProductos"
    [products]="products" >
```

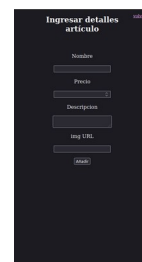
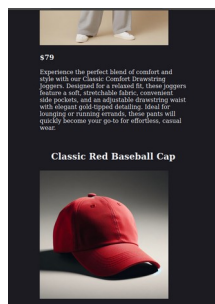
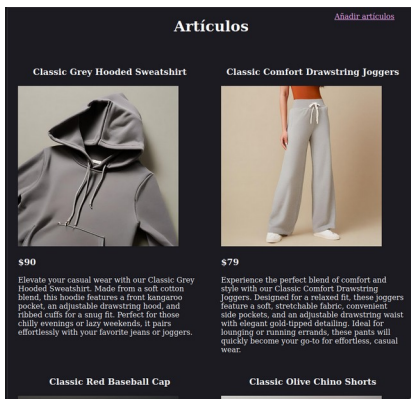
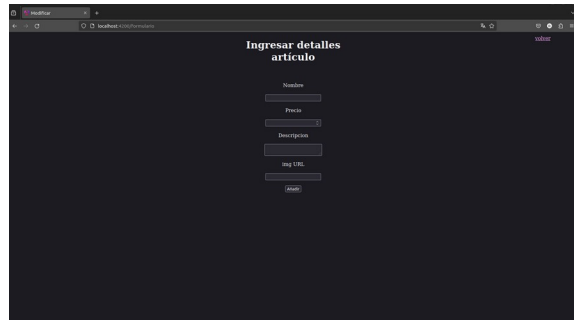
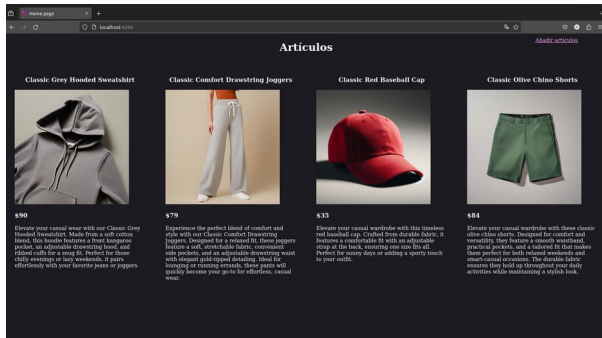
`.error-message{text-align: center; color: red;};` Estiliza los elementos con la clase `.error-message` para centrar el texto y colorearlo de rojo.

```
<div *ngIf="formSubmitted && !addForm.valid" class="error-message">
  Por favor, complete el formulario correctamente.
</div>
```

`button{ margin-top: 10px; width: fit-content; margin: 0 auto;};` Estiliza todos los elementos `<button>` para que tengan un margen superior de 10 píxeles, un ancho que se ajuste a su contenido, y los centra horizontalmente dentro de su elemento contenedor.

`label,input{margin: 3px;};` Aplica un margen de 3 píxeles a todos los elementos `<label>` y `<input>`.

@media screen and (max-width: 700px) {...}: Define estilos específicos para pantallas con un ancho máximo de 700 píxeles. Reduce el tamaño de fuente de los elementos **<h1>** y ajusta el tamaño de fuente de los elementos **<a>** cuando el ancho de la pantalla es de 700 píxeles o menos.



5) Demostración de funcionamiento(ver video)