xacademy

Angular - Clase 2

Componentes, módulos y Routing

En esta clase veremos lo que son los componentes, su ciclo de vida, como reutilizar código en angular y cómo manejar los estados, comunicaciones y demás tareas pertinentes a los componentes.

También veremos qué son los módulos y cómo cambiar de página en Angular.

Componentes

En esencia, Angular es un framework basado en componentes. Pero ¿qué es un componente?

Al escribir una página web, nos encontramos con muchas secciones que deben ser repetidas, por ejemplo la cabecera de la página, en la cual se listan las secciones de nuestra página web. He aquí un ejemplo de la cabecera de la [página oficial de Angular]():

En caso de no hacer uso de una SPA, si queremos hacer modificaciones a la cabecera, tendremos que copiar y pegar dicha modificación en todos los archivos HTML que hagan uso de la cabecera.

Para resolver este inconveniente existen los Componentes. Podemos convertir dicha cabecera en un componente y utilizarlo en muchos lugares. De esta manera es mucho más fácil de actualizar una parte de la página pues el código se escribe una sola vez y luego se llama a dicho componente.

Pero para facilitar el ejemplo, vamos a crear un componente nosotros mismos.

En VS Code, vamos a ir a **`Terminal -> New Terminal`**

Dentro de esta ventana vamos a ingresar:

```
ng generate component clicker
```

Veremos que algunas cosas suceden:

Se generó una nueva carpeta llamada clicker, dentro de dicha carpeta se crearon 4 archivos. Un HTML, un TS, un SCSS y un spec.ts. También se actualizó el archivo `app.module.ts` para incluir el nuevo componente. Esto implica que el nuevo componente quedó registrado en el módulo.

Estructura de un componente

En primer lugar vamos a revisar el archivo `clicker.component.ts`

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

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

  constructor() { }

  ngOnInit(): void {
  }

}
```

El código se parece bastante al del archivo `app.component.ts`

Esto se debe a que en sí mismo ese archivo es un componente también.

Vamos a definir cada línea:

Esta línea simplemente importa las interfaces *Component* y *OnInit.*

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

Aquí tenemos nuestro *Decorador* `@Component.`

Este decorador define la clase “ClickerComponent” como un Componente. De esta manera Angular sabe que dicha clase debe ser tratada como un componente y no como una clase más.

```
@Component({
  selector: 'app-clicker',
  templateUrl: './clicker.component.html',
  styleUrls: ['./clicker.component.scss']
})
```

Al momento de definir un componente tenemos muchos parámetros que podemos modificar, pero los listados arriba son los más importantes en realidad:

`selector: 'app-clicker'`

Esta línea define el selector del componente como “app-clicker”. Esto implica que para usar nuestro componente en un archivo HTML debemos usar ese nombre. Más adelante veremos cómo.

`templateUrl: './clicker.component.html'`

Ésta línea define la URL donde se encuentra el archivo HTML vinculado a dicho componente.

`styleUrls: ['./clicker.component.scss']`

Ésta línea define el o las URL donde se encuentran el o los archivos CSS vinculados al componente.

```
export class ClickerComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

He aquí la definición de la clase del componente. Como podemos observar en la primera línea, la clase implementa la Interfaz *OnInit*, lo cual implica que se debe crear la función `ngOnInit()`, es por eso que la vemos definida abajo.

Por último la clase posee un constructor.

Editando nuestro primer componente

Para ver un componente en acción, vamos a trasladar el código del archivo `app.component.ts`, `app.component.html` y `app.component.scss` a los respectivos archivos del nuevo componente que creamos.

Una vez trasladado el código, podemos hacer uso del componente. Pero, ¿cómo lo hacemos?

Aquí entra en juego el primer parámetro del decorador *@Component*. Como podemos observar, posee un string el cual indica ***app-clicker***. Éste lo usaremos como un *TAG HTML* dentro de `app.component.html`

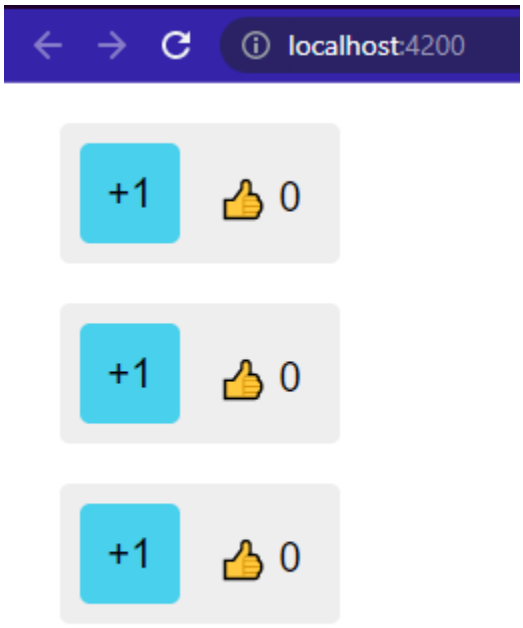
`app.component.html`

```
<app-clicker></app-clicker>
```

¡Esta es una nueva *TAG HTML* hecha por angular! A partir de ella, podemos colocar todo el contenido del archivo `clicker.component.html` en el lugar desde donde lo llamemos. Por supuesto que la lógica que ingresamos en el archivo TS también funciona correctamente.

A simple vista puede parecer que no hicimos mucho, ¿pero qué sucede si hacemos esto?

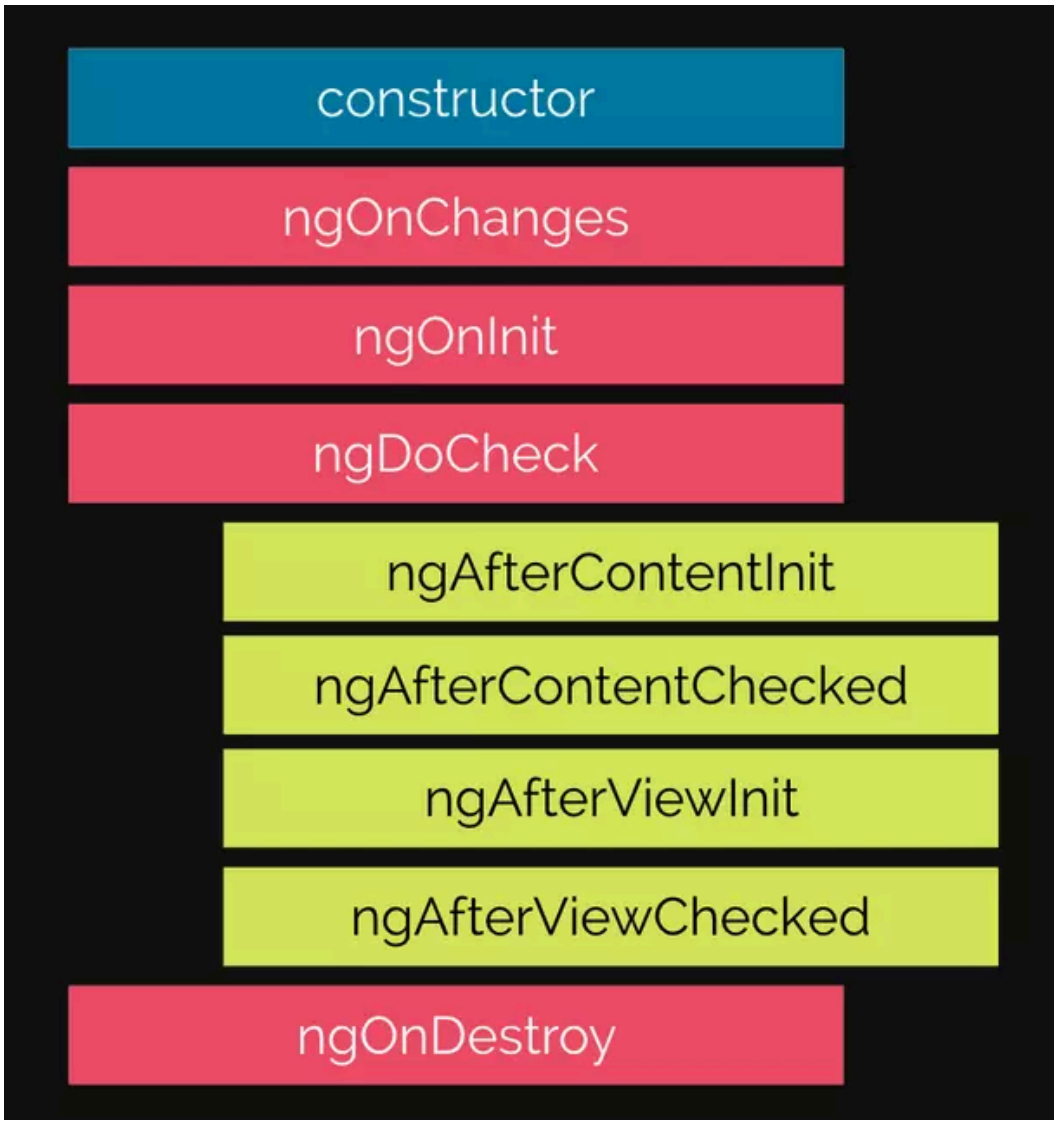
```
<app-clicker></app-clicker>
<app-clicker></app-clicker>
<app-clicker></app-clicker>
```



Hemos triplicado el código sin necesidad de escribir tres veces lo mismo. Es decir que hemos reutilizado el código y cada cambio realizado en el componente se refleja en todos los lugares donde se utilice dicho componente. Estas instancias del componente son independientes entre sí, es decir que cada uno es un contador distinto y no comparten información entre ellos. Prueba hacer click en los botones y verás que los valores se incrementan solo en el componente que estás clicando.

Ciclo de vida de un componente

Un componente posee varias “fases” desde el momento en que se crea hasta el momento que se destruye. Un componente se crea al inicio de una página, al cambiar de página a una distinta, o incluso se crean y destruyen de forma dinámica dentro de una misma página. Todo depende de cómo funcione la página en cuestión.



El listado anterior muestra el ciclo de vida completo de un componente. Pero a nosotros nos interesan solo 4. Los demás, si bien tienen su importancia y su uso, los mismos son para casos específicos o muy avanzados para este curso.

1. **Constructor:** Este es el constructor de la clase, este se encarga de la inicialización de la clase del componente, inyección de dependencias y tareas de inicialización de miembros de la clase, como variables.

2. **ngOnInit:** Este se encarga de inicializar datos que no forman parte del constructor, como los sistemas de comunicación del componente con otros componentes (veremos esto más adelante). Este es un buen momento para hacer tareas como llamadas a servicios externos o APIs.

3. **ngAfterViewInit:** En este punto del ciclo, la vista del componente fue completamente inicializada, con vista nos referimos al HTML, es decir que el HTML fue renderizado. En este punto es útil hacer tareas específicas con componentes hijos, los cuales no estaban disponibles aún en las fases anteriores pues no se habían inicializado.

4. **ngOnDestroy:** Esta es la última fase del ciclo de vida de los componentes. Aquí suceden todas las tareas previas a eliminar por completo el componente. Podemos hacer tareas de cierre aquí como eventos que suceden al cambiar de página.

Veamos un ejemplo de los ciclos de vida en acción.

Cada fase del ciclo de vida tiene un momento de activación y podemos ejecutar código en cada fase. Para ello debemos implementar las interfaces correspondientes y declarar los nombres de las funciones del ciclo de vida de forma correcta. El código que ingresemos dentro se ejecutará.

```
export class AppComponent implements OnInit, AfterViewInit, OnDestroy{
  constructor() {
    console.log('Soy el constructor!');
  }
  ngAfterViewInit(): void {
    console.log('Soy el AfterViewInit!');
  }
  ngOnInit(): void {
    console.log('Soy el OnInit!');
  }
  ngOnDestroy(): void {
    console.log('Soy el OnDestroy!');
  }
}
```

Al ejecutar la página web nos encontraremos un LOG como el siguiente en la consola. (en general podemos acceder a ella con la tecla F12 en nuestro navegador)

```
Soy el constructor!
Soy el OnInit!
Soy el AfterViewInit!
Soy el OnDestroy!
>
```

Si eres perspicaz habrás notado que en el código, onInit se ejecuta antes que afterViewInit aunque este está declarado antes. Esto se debe a que no importa el orden de la declaración si no el momento en que sucede cada fase del ciclo de vida.

Comunicación entre componentes

Dentro de un componente podemos hacer uso de muchos componentes y dentro de los mismos llamar a muchos más. Esta jerarquía de componentes requiere que todos interactúen entre sí de forma adecuada y esto en muchas ocasiones implica que debemos pasar información entre componentes de padres a hijos y viceversa.

```
<componente-padre>
  <componente-hijo></componente-hijo>
</componente-padre>
```

Decorador @Input

El decorador @Input nos permite definir una variable la cual queda expuesta, por lo que podemos modificar su valor desde el lugar de donde llamamos al componente.

```
export class ClickerComponent{
  contador = 0;
  @Input() bonus = false;

  constructor() { }

  sumar(cantidad: number): void {
   this.contador = this.contador + cantidad;
  }
```

```
}
```

El decorador indica a Angular que la variable ***bonus*** puede ser vinculada a un valor que ingrese el padre, vamos a ver como hacer esto.

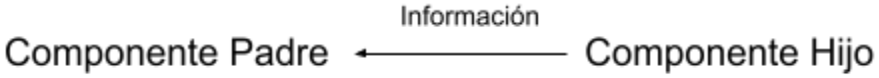
```
<app-clicker [bonus]="bonus"></app-clicker>
```

En el momento de llamar al componente hijo, debemos pasar el valor como si fuera un atributo pero entre corchetes. De esta manera el valor se envía desde el componente padre. Como podrás imaginar podemos pasarle variables que vamos modificando desde el componente padre y el valor se verá reflejado en el componente hijo.

Decorador @Output

El decorador *@Output* sirve para el mismo propósito pero a la inversa: en este caso es el componente hijo el que envía la información al padre.

@Output



Para ello, debemos hacer uso de Eventos (como los que usamos para hacer click).

```
export class ClickerComponent{
  contador = 0;
  @Input() bonus = false;
  @Output() valorIncrementado = new EventEmitter<number>();

  constructor() { }

  sumarValor() {
    this.contador = this.contador + this.valorClick;
    this.valorIncrementado.emit(this.valorClick);
  }
}
```

El decorador Output se usa para definir una variable con un EventEmitter, la cual toma un tipo de dato en su constructor, en este caso number.

Para enviar la información, debemos llamar a la función ***emit*** y en caso de que hayamos definido un tipo para el eventEmitter al momento de definirlo, debemos pasar el valor del evento. En este caso el valor es el valor de click que tenemos.

```
<app-clicker [bonus]="bonus"
(valorIncrementado)="sumarAlTotal($event)"></app-clicker>
```

Al momento de suscribirnos al evento, lo hacemos de la misma forma que hicimos con el evento click, entre paréntesis, entre las comillas debemos ingresar una instrucción. Llamamos a la función. El nombre de variable ***$event*** es el valor del evento que pasamos anteriormente.

app.componet.ts

```
export class AppComponent implements OnInit {

 bonus = false;
 total = 0;

 tablaPuntuaciones: {nombre: string, puntuacion: number}[] = [
   {nombre: 'Juan', puntuacion: 855},
   {nombre: 'Pedro', puntuacion: 803},
   {nombre: 'Luis', puntuacion: 720},
 ];

 sumarAlTotal(valor: number): void {
   this.total += valor;
 }

 ngOnInit(): void {
   window.setTimeout(() => {
     this.bonus = true;

     window.setTimeout(() => {
       this.bonus = false;
     }, 5000);

   }, 4000);
 }

}
```

clicker.component.html

```
<app-clicker [bonus]="bonus"
(valorIncrementado)="sumarAlTotal($event)"></app-clicker>
<app-clicker [bonus]="bonus"
(valorIncrementado)="sumarAlTotal($event)"></app-clicker>
<app-clicker [bonus]="bonus"
(valorIncrementado)="sumarAlTotal($event)"></app-clicker>

<p>Total: {{total}}</p>

<p>Puntuaciones más altas</p>

<table>
 <tr>
   <th>Nombre</th>
   <th>Puntuación</th>
 </tr>
 <tr *ngFor="let jugador of tablaPuntuaciones">
   <td>{{jugador.nombre}}</td>
   <td>{{jugador.puntuacion}}</td>
 </tr>
</table>
```

Módulos

Los módulos nos permiten separar partes del código en secciones definidas o clústers. Normalmente se utilizan para separar partes del sistema que se diferencian mucho entre sí. Por ejemplo, la sección de ventas y la sección de perfil de usuario en una página web.

Los módulos también se usan para la organización de la estructura de nuestra aplicación, separando el código más importante del código menos relevante e incluso código que se reutiliza en toda la aplicación.

La utilización de módulos nos permite mejorar la estructura, aumentando la facilidad de encontrar componentes en una aplicación y si se usa en conjunto con otras características de angular, podemos incluso mejorar la performance y aumentar la velocidad de la carga inicial de nuestra aplicación.

Los módulos por dentro

Un módulo por dentro es una clase con el decorador @ngModule. Posee 3 propiedades principales:

Declarations: Aquí registramos los componentes que forman parte del módulo. Por defecto los componentes son privados y no se pueden importar en otros módulos.

Imports: Aquí vamos a importar los módulos que queramos utilizar dentro de este módulo. En este caso se importa el common module que contiene muchas de las herramientas básicas de angular como por ejemplo los componentes.

Exports: Aquí pondremos los módulos o componentes que queremos exportar para que lo usen otros módulos, al importar este módulo.

```
@NgModule({
  declarations: [
    Componente1
  ],
  imports: [
    CommonModule,
    Modulo1
  ],
  exports: [
    Componente1
  ]
})
export class Modulo { }
```

Los módulos definen una unidad dentro de nuestra aplicación y todo lo que creemos dentro de ella, ya sean componentes, directivas, pipes, guards, interceptors, servicios o clases generadas por Angular debe estar registrados en un módulo.
Estos se declaran en la key *declarations* para su uso en la aplicación.
Ej:

```
declarations: [
  componente1,
  pipe1,
  directiva1,
],
```

NOTA: Recordemos que todo lo que declaremos debe ser importado en primer lugar.

En caso de que queramos que los componentes que creamos en este módulo sean utilizados en un módulo externo, debemos exportarlos a otro módulo, para ello usamos la property *exports.*
Ej:

```
exports: [
  Componente1,
  pipe1,
  directiva1,
]
```

Y en caso de que queramos en el módulo actual, utilizar componentes, directivas, pipes, servicios, etc. Debemos importar el módulo donde fueron declarados e importados. Esto lo hacemos en la property *Imports*.
Ej:

```
imports: [
  CommonModule,
  Modulo1
],
```

Organización

Con lo anterior en mente podemos crear una estructura general de nuestra aplicación donde solamente trabajaremos con los objetos que necesitamos.

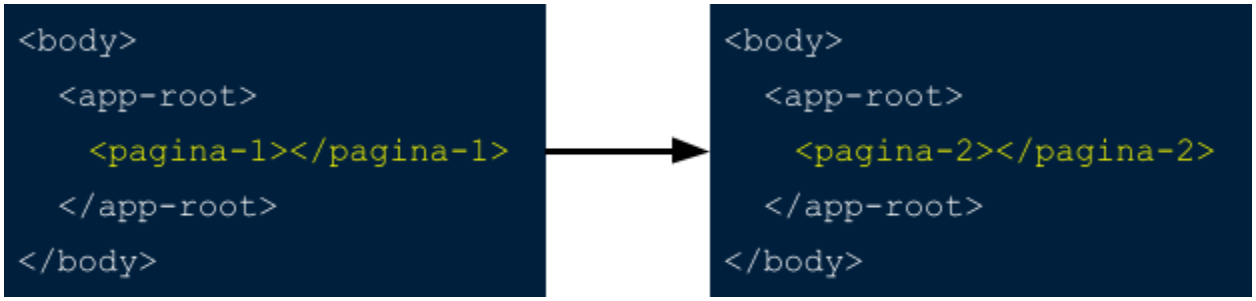
De esta manera, si nos encontramos en un “moduloA”, el “moduloB” el cual no tiene relación con el anterior no se cargará hasta que se lo requiera. Por lo tanto si no vamos al Módulo B en nuestra aplicación, este no nos estará ocupando ni recursos ni espacio de almacenamiento. Esto se llama “lazy loading”, o en español “carga perezosa” donde sólo vamos a descargar del servidor los archivos que necesitamos para mejorar la performance de la aplicación.

Routing

El Routing es una manera de simular lo que sucede en una MPA (Multiple Page Application) donde se carga una nueva página durante la navegación.

Pero como bien aprendimos en la clase anterior, una SPA posee solo una página web y no se puede refrescar el browser. Por lo que debemos optar por una nueva forma de manejar el contenido de la página.

Tomemos como ejemplo el gráfico de la clase anterior:



Como podemos observar, la TAG *<app-root>* es la encargada de ir modificando el contenido de la app. Dependiendo de donde estemos ubicados en la aplicación, el contenido del HTML se irá modificando para “emular” el cambio de página de una MPA. Pero lo que sucede por dentro es que el contenido de la tag root de angular cambia para ser primero una página y luego otra dependiendo de las necesidades del momento. De esta manera se manejan todas las SPAs.

En angular el encargado de estos “cambios de página” es el Router.

El Módulo Router

Cuando definimos la aplicación se nos preguntó si queríamos añadir “Angular Routing”, esto nos generó un archivo de routing básico el cual nos permite definir las “páginas” de nuestra aplicación y también las rutas de la misma, para que podamos movernos por diferentes partes de nuestra aplicación.

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

const routes: Routes = [];

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

Como podemos observar, lo que tenemos aquí es un módulo el cual importa el Módulo “RouterModule”, le aplica una variable (routes) y luego lo reexporta. En el módulo appModule, el cual es el módulo base de nuestra aplicación, se importará el módulo Router.

Rutas

Las rutas son como las “URL” de nuestra aplicación. Éstas nos permiten definir las páginas que deseamos tener. Veamos el siguiente ejemplo:

```
const routes: Routes = [
  {path: 'productos', component: ProductosComponent},
  {path: 'carrito', component: CarritoComponent},
  {path: 'caja', component: CajaComponent},
];
```

En este caso, para acceder a una sección solo debemos usar las URL

nuestraApp.com/productos
nuestraApp.com/carrito
nuestraApp.com/caja

Según qué URL introducimos en el navegador veremos en pantalla el componente de Productos, Carrito o Caja.

¿Cómo funcionan las Rutas?

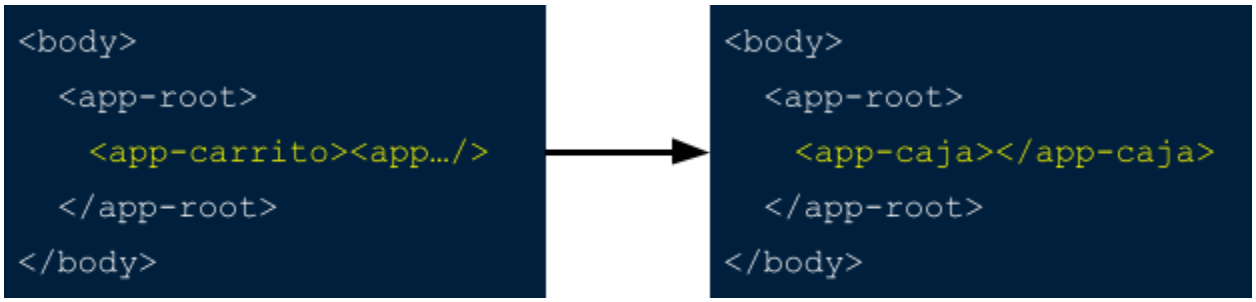
Las Rutas permiten definir las URL y le indican a angular que contenido mostrar según ellas.

En este caso, tenemos el módulo *app-routing.module.ts* el cual es el módulo de routing base de Angular. Más adelante podremos crear nuevos módulos para definir subrutas.

El módulo anteriormente mencionado, utiliza como base la tag *<app-root>* del HTML, y coloca el componente que le pasemos en la property “component” del objeto de rutas:

```
  {path: 'carrito', component: CarritoComponent},
```

De esta manera cuando estemos en el path ***nuestraApp.com/carrito*** se mostrará el componente de Carrito y si cambiamos a la Ruta de Caja, se eliminará el componente de carrito y se reemplazará con el de la Caja.



Lazy Loading

Como mencionamos anteriormente, los módulos se pueden utilizar para cargar contenido de forma dinámica, evitando así que se carguen archivos en memoria que aún no se necesitan. Esto se llama Lazy Loading y es uno de los actores fundamentales en la estructura de nuestra aplicación.

En lugar de levantar y mostrar un componente, lo que haremos es crear módulos “Feature” los cuales se van a encargar de sectorizar las partes fundamentales:

- Módulo Productos
- Módulo Carrito
- Módulo Caja

Luego vamos a apuntar con los path a dichos módulos, en lugar de a un componente. Y luego estos tendrán su propio módulo de Routing y apuntarán al componente pertinente en cada caso.

```
{
  path: 'productos',
  loadChildren: () => import('./features/productos/productos.module')
    .then(m => m.ProductosModule)
},
```

La Property loadChildren nos permite pasar una función la cual importa un módulo mediante una promesa y luego en la callback “then” podemos pasar a través del param el módulo a cargar.

Luego en el módulo de productos, en su propio lazy loading podremos definir las subrutas:

Productos.module.ts

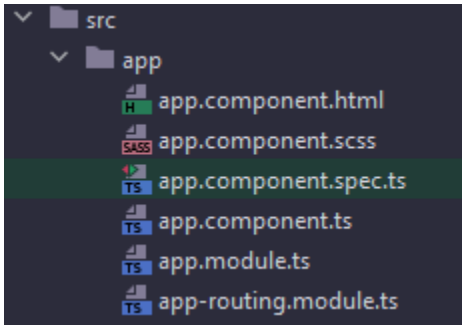
```
{path: '', component: ProductosComponent},
{path: 'ofertas', component: ProductosComponent},
{path: '**', redirectTo: ''},
```

Comodines:

Como podemos observar en la última línea, el path es “**” lo cual indica todo lo que no coincida. Esto será muy útil en caso de que haya un error al escribir la URL y se quiere redirigir al usuario a una página por defecto o bien a una página “404” que le indique que la página que busca no se encontró o la escribió de forma incorrecta. Le indicamos la URL a redirigir con la Property “RedirectTo”.

Nuestros propios módulos

Hora de pasar a la práctica. Luego de generar un nuevo proyecto, deberías tener solo un módulo: *app.module.ts*



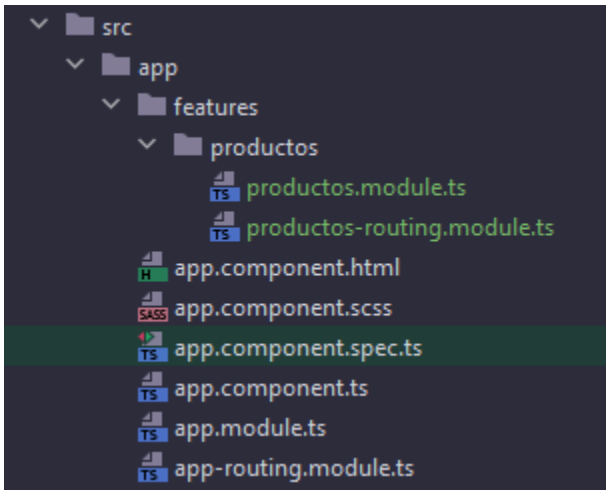
Este es el módulo base y el se encargará de cargar los demás módulos.

Primero vamos a crear el módulo de Productos. Para ello abriremos una anueva terminal de VSCode (o el IDE con el que trabajes) e introduciremos el siguiente comando:

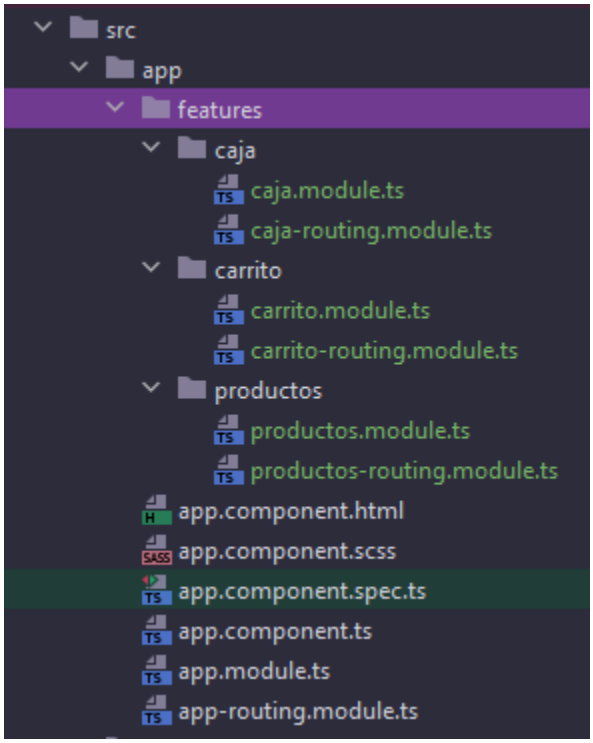
```
ng generate module features/productos --routing
```

Esto nos generará un nuevo módulo en la carpeta Features y subcarpeta Producto. (ya veremos más adelante por qué se llama Features)

Y con la opción “--routing” crearemos un módulo Routing para manejar las subrutas. Esto es opcional pues no todos los módulos que generemos las necesitan.



Hacemos lo mismo con los demás módulos:



Ahora vamos a crear las conexiones con dichos módulos:

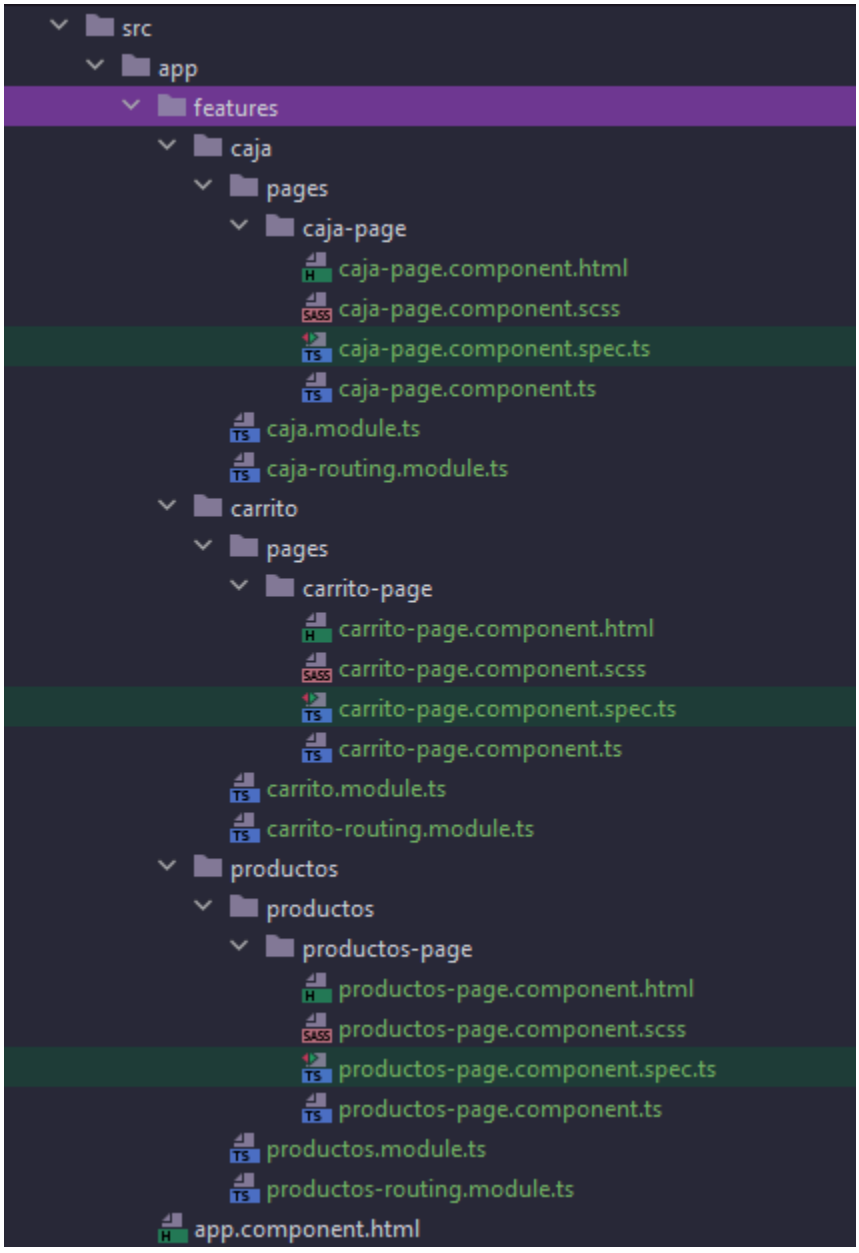
```
const routes: Routes = [
  {
    path: 'productos',
    loadChildren: () => import('./features/productos/productos.module').then(m => m.ProductosModule),
  },
  {
    path: 'carrito',
    loadChildren: () => import('./features/carrito/carrito.module').then(m => m.CarritoModule),
  },
  {
    path: 'caja',
    loadChildren: () => import('./features/caja/caja.module').then(m => m.CajaModule),
  }
];

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

Hemos generado las conexiones con los módulos, pero aún no podemos ver nada, dado que primero necesitamos crear componentes base para los mismos:

Vamos a crear componentes de Páginas principales para dichos módulos:

Vamos a ejecutar:

- ng generate component features/caja/pages/cajaPage
- ng generate component features/carrito/pages/carritoPage
- ng generate component features/productos/productos/productosPage



Una vez generados los componentes, vamos a crear las conexiones con dichos componentes.

Empecemos con el Módulo de Routing de Productos.

productos-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import {ProductosPageComponent} from './pages/productos-page/productos-page.component';
import {ProductoPageComponent} from './pages/producto-page/producto-page.component';

const routes: Routes = [
  {path: '', component: ProductosPageComponent},
  {path: 'ofertas', component: ProductosPageComponent},
  {path: 'producto/:id', component: ProductoPageComponent},
  {path: '**', redirectTo: ''}
];

2 usages  new *
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ProductosRoutingModule { }
```

Aquí tenemos varias cosas sucediendo en cada línea:

```
{path: '', component: ProductosPageComponent},
```

El path base está vacío por lo que usará como path el path anterior, el definido en *app-routing.module.ts.*

Por lo que para acceder a este path es necesario escribir como URL:

aplicacion.com/productos

```
{path: 'ofertas', component: ProductosPageComponent},
```

Ahora el path es “ofertas” por lo que para acceder será necesario escribir:

aplicacion.com/productos/ofertas

```
{path: 'producto/:id', component: ProductoPageComponent},
```

Podemos observar “/:id” en el path, esto quiere decir que podemos enviar un ID como param en la URL, este es opcional, pero probablemente sea requerido por la pagina pues no sabe qué producto quieres ver, por lo que te solicita el ID del producto para mostrar el detalle del mismo:

aplicacion.com/productos/producto/25

Por último vemos un selector comodín el cual nos reenviará a la URL ***aplicacion.com/productos*** por lo que veremos la página principal de productos.

Luego queda replicar esto a los demás módulos.

Estructura general

Como mencionamos anteriormente, los módulos también se usan para organizar a nivel estructural nuestra aplicación.

Vamos a poner manos a la obra con esto.

La estructura general adoptada por la mayoría de proyectos es la siguiente:

- Módulo general
    - Módulo **CORE**
        - Servicios
        - Interfaces
        - HTTPService
        - Interceptors
        - Guards
    - Módulos **FEATURE**: Pueden ser varios
        - Páginas de nuestra aplicación
    - Módulo **SHARED**
        - Componentes compartidos
        - Directivas
        - Pipes

Ya generamos 3 módulos FEATURE, los cuales se cargan mediante lazy loading, pero otros módulos deben ser cargados inmediatamente. Ésta es la función del módulo CORE.

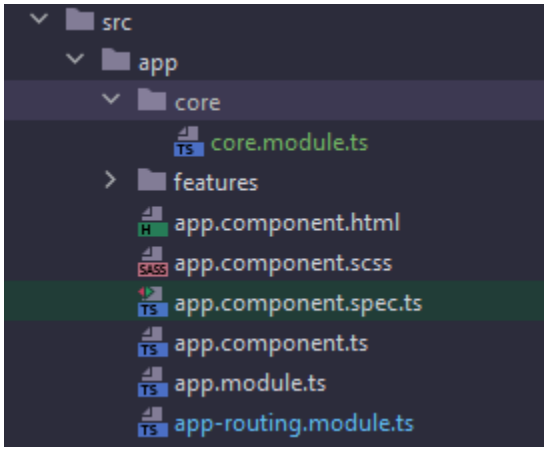
Módulo CORE

El Módulo CORE contiene elementos esenciales para la aplicación, los cuales intervienen en su funcionamiento general, seguridad, servicios y demás utilidades que son obligatorias.

Este módulo NO REQUIERE ROUTING, dado que solo contiene clases y código, más ningún tipo de HTML a mostrar.

Para generarlo, vamos a introducir:

```
ng generate module core
```



Finalmente vamos a registrar dicho módulo entre los imports del módulo base: *app.module.ts*

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

Ahora el módulo CORE será cargado ni bien se inicie la aplicación, dado que no fue referenciado mediante lazy loading si no que directamente en los imports del módulo base.

Módulos FEATURE

Los módulos FEAURE son aquellos que forman parte de la aplicación como características de la misma. Estos incluyen cosas como por ejemplo secciones de nuestra página, o grupos de funcionalidades, como PRODUCTOS.

A diferencia de los demás módulos, aquí puedes tener X cantidad de módulos FEATURE, dado que será igual a la cantidad de secciones de tu aplicación.

Módulo SHARED

El Módulo SHARED es aquel que almacenará información que será utilizada en toda la aplicación, tal como componentes reutilizables, como un HEADER o FOOTER, directivas o pipes.

A diferencia del módulo core, este NO debe ser importado en app.module.ts, si no que debe ser importado en módulos feature siempre que se necesite.

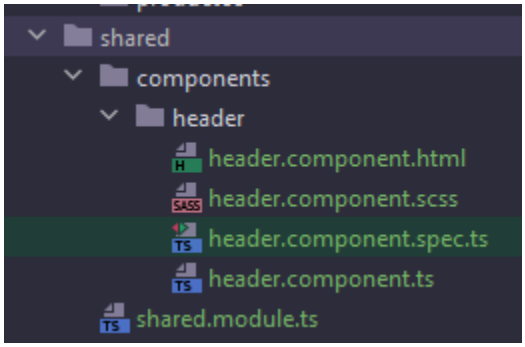
```
ng generate module shared
```

En este módulo, se hará mucho uso de la property EXPORTS, dado que vamos a exportar todos los componentes, directivas o pipes que se requieran en otros lugares.

Router

Finalmente, el enrutamiento no puede suceder desde la barra de URLs, dado que esto dispara una carga de página y no queremos que esto suceda.

Primero vamos a crear un Header en el módulo SHARED.



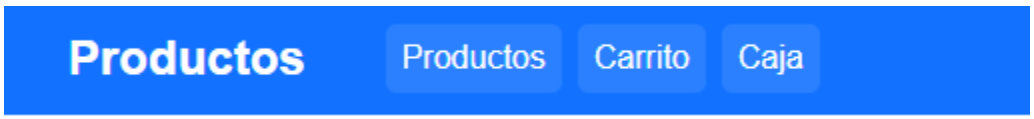
Recordemos exportarlo en el módulo shared.

```
exports: [
  HeaderComponent,
]
```

Luego vamos a crear un header simple

```
<header>
 <h1>{{titulo}}</h1>
 <nav>
   <ul>
     <li>Productos</li>
     <li>Carrito</li>
     <li>Caja</li>
   </ul>
 </nav>
</header>
```

Con un poco de magia de CSS vamos a dejarlo así



Y lo vamos a colocar en todas las páginas (previamente importando el módulo SHARED en cada módulo FEATURE)

```
<app-header titulo="Caja"></app-header>
```

Ahora debemos hacer que cada vez que le hagamos click a uno de los botones, este genere una redirección a una URL específica la cual hayamos registrado previamente. Haremos uso de la directiva *routerLink*:

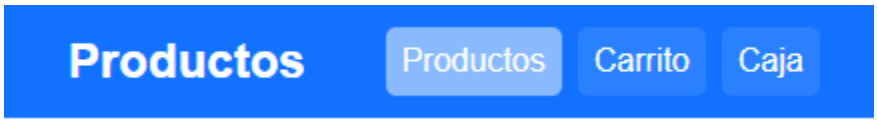
```
<header>
  <h1>{{titulo}}</h1>
  <nav>
    <ul>
      <li routerLink="/productos" routerLinkActive="active">Productos</li>
      <li routerLink="/carrito" routerLinkActive="active">Carrito</li>
      <li routerLink="/caja" routerLinkActive="active">Caja</li>
    </ul>
  </nav>
</header>
```

Ahora nuestros botones generarán una redirección a las URL que indicamos en el link.

Nótese que se utilizó una barra antes de la URL, esto implica que es una URL completa que reemplazará la totalidad de la URL. Si no la colocaremos, esta sería una url relativa al lugar donde se encuentra.

Si estamos en ***"productos"*** y usamos ***"/ofertas"***, la nueva URL es ***"/ofertas"***, si usamos solo ***"ofertas"***, la nueva URL es ***"productos/ofertas".***

También tenemos una directiva llamada *routerLinkActive,* ésta asigna una clase, la cual le pasamos como param, cuando la URL que definimos es la Activa, es decir estamos en esa URL.



Ahora con esa clase, podemos darle un color más claro al botón cuando estamos en “Productos”