Tema 4 - Angular

Módulos.

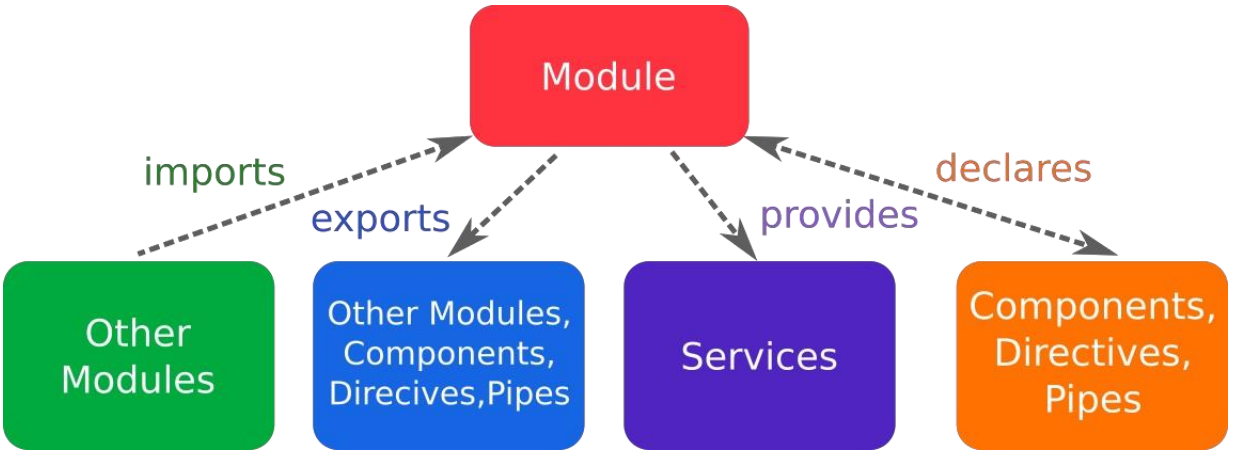
Desarrollo web en entorno cliente
IES Pere Maria Orts I Bosch

Índice

Módulos

Un módulo en Angular es una clase con el decorador @NgModule. Hasta ahora el único módulo que teníamos en la aplicación era el módulo principal (AppModule). Todas las partes de la aplicación (componentes, servicios, rutas, pipes) se declaraban en ese módulo. Sin embargo, cuando la aplicación es relativamente grande, se hace más compleja de mantener si no la separamos en diferentes módulos. Además, esto nos permite (en el caso de las rutas) usar una característica llamada carga en diferido (lazy loading), que mejora el rendimiento de la aplicación

En esta sección veremos como separar nuestra aplicación en módulos y qué criterios tener en cuenta para ello. También veremos como los módulos exportan características que pueden ser usadas por otros módulos (importación).



Array de inicialización (bootstrap)

El array llamado bootstrap de un módulo sólo se suele utilizar en el módulo de la aplicación AppModule, y sirve para indicar a Angular qué componente es el principal de la aplicación (AppComponent) y se cargará el primero. La plantilla de este componente es la que se incluirá en el elemento <app-root> de index.html.

```
@NgModule({
  ...
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Array declarations

Los componentes, directivas y pipes deben declararse dentro de un módulo (y sólo en uno). Pronto veremos como exportarlos desde el módulo donde se declaran para que puedan usarse en otros. Si no metemos alguno de los elementos mencionados antes en el array declarations de un módulo, Angular no sabrá que existen y no los compilará, lanzando un error cuando intentemos usarlos.

```
@NgModule({
```

```
  declarations: [
    AppComponent,
    ProductListComponent,
    ProductItemComponent,
    ProductFilterPipe,
    StarRatingComponent,
    WelcomeComponent,
    ProductDetailComponent,
    ProductEditComponent
  ],
  ---
})
export class AppModule { }
```

Array exports

El array exports nos permite que el módulo actual comparta sus componentes, pipes o directivas. También se pueden exportar otros módulos (incluso sin importarlos previamente), esto se puede usar para crear un módulo que al importarlo, nos permita importar varios módulos de golpe, en lugar de hacerlo uno a uno.

Los servicios no deben ser exportados. Cuando los servicios se declaran (array providers) en cualquier módulo, se inyectan a nivel global siendo accesibles por toda la aplicación (y todos los módulos que la componen).

Array imports

Un módulo puede importar otros módulos usando el array imports. Algunos de los cuales serán módulos de Angular (FormsModule, HttpClientModule, ...), también módulos de terceros instalados (Bootstrap, Angular Material, ...), mientras que otros serán módulos creados por nosotros.

Cuando un módulo importa otro, tendrá acceso solamente a lo que dicho módulo haya exportado (es como su parte pública al exterior). Importa sólo los módulos que necesites.

```
@NgModule({
  ---
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot(APP_ROUTES)
  ],
  ---
})
export class AppModule { }
```

Array providers

El array providers nos permite registrar servicios, y otros valores u objetos globales que queramos añadir al contenedor de dependencias de la aplicación. Todos esos servicios estarán disponibles a nivel global para toda la aplicación. Por norma general, los servicios es mejor declararlos en el AppModule. Sólo es admisible cargarlos en otro módulo si estamos 100% seguros que no se va a usar en algún

componente/directiva/pipe/servicio de cualquier otro módulo.

Nunca declares el mismo servicio en 2 módulos diferentes.

```
@NgModule({
  ---
  providers: [
    Title,
    ProductService,
    ProductDetailGuard,
    CanDeactivateGuard,
    ProductDetailResolve
  ],
  ---
})
export class AppModule { }
```

Importante: Desde Angular 6.0, los servicios se pueden autoinyectar en la aplicación sin necesidad de declararlos en ningún módulo. Esto se hace con el atributo provideIn del decorador @Injectable. El valor de dicho atributo suele ser 'root', es decir, equivalente a declararlos en el módulo de la aplicación. También se puede especificar la clase de un módulo, lo que implicaría, que hasta que la aplicación no cargara dicho módulo, el servicio no estaría disponible.

Al crear un servicio mediante la herramienta Angular CLI, se crea con esta configuración, por lo que no debemos añadirlo al array providers.

```
@Injectable({
  providedIn: 'root'
})
```

https://angular.io/guide/providers

Dividiendo nuestra aplicación en módulos

División por características

Para organizar una aplicación en módulos hay que seguir ciertas reglas. Algunas de las que se recomiendan son las siguientes:

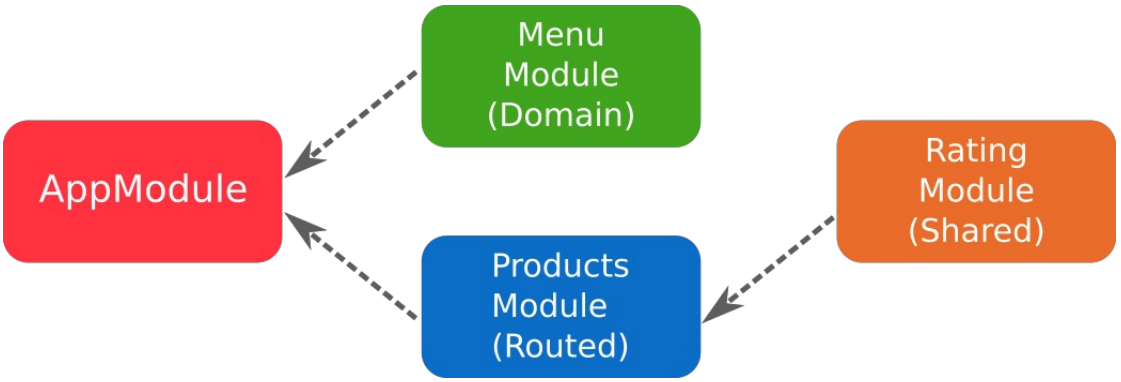
- Módulos de dominio: Módulos que se importan en el módulo de la aplicación. Sirven simplemente para separar código del módulo principal (aquí no meteríamos componentes que representen una ruta). Un ejemplo sería un módulo de Menú.

- Módulos de sección: Módulos que representan una sección de nuestra aplicación (un conjunto de rutas relacionadas entre sí). Gestionan un conjunto de páginas relacionadas. Por ejemplo un módulo de Productos podría contener los componentes ProductsList, ProductDetail, ProductEdit, ...

- Módulos de rutas: Módulo asociados con un "módulo de sección". Sirve para separar la gestión de las rutas.

- Módulo de servicio (CommonModule): Módulo o módulos que declaran una serie de servicios. Sólo se importan desde el módulo de la aplicación (AppModule) y tiene el mismo efecto que simplemente declarar los servicios ahí (sirve para separar código). Normalmente no se declaran componentes aquí. Ejemplo → HttpClientModule de Angular.

- Módulos de componentes: Módulos que exportan componentes, directivas y pipes que pueden ser usados a su vez en diferentes módulos de la aplicación. Por ejemplo el componente StarRatingComponent podría ir en uno de estos módulos (RatingModule).



https://angular.io/guide/module-types

Creando el módulo ProductsModule

Lo primero va a ser crear el módulo de productos (ProductsModule), donde

pondremos los componentes relacionados con las rutas de los productos. Todo lo que dependa de este módulo se meterá en un directorio llamado '/products'. Creamos un nuevo módulo ejecutando lo siguiente (desde app/src):

```
ng g module products --routing true
```

Veremos que refactorizar en módulos un proyecto que no está pensado así inicialmente puede ser relativamente costoso. Por ello, se recomienda ir creando los módulos necesarios en la fase más inicial posible del proyecto.

Este es el módulo que ha creado (products/products.module.ts):

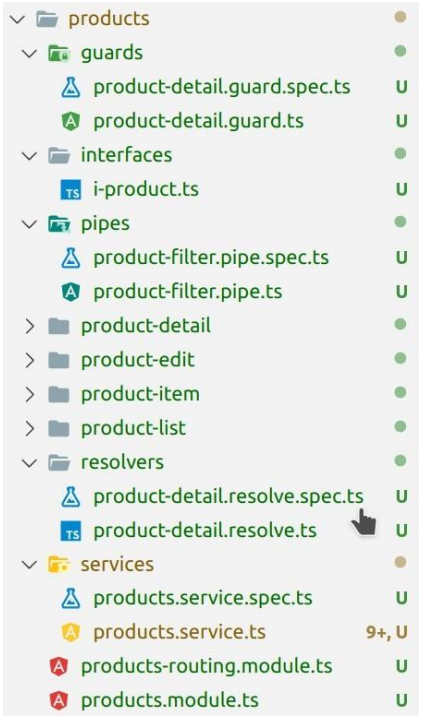
```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { ProductsRoutingModule } from './products-routing.module';

@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    ProductsRoutingModule
  ]
})
export class ProductsModule { }
```

El módulo CommonModule de Angular es equivalente a BrowserModule (pero este último sólo se debe importar en AppModule). Permite usar directivas como ngIf o ngFor en el módulo actual.

Vamos a situar todos los elementos relacionados con la gestión de productos en nuestro nuevo directorio.



Como se puede observar, si la aplicación comienza a crecer mucho y tiene diferentes secciones, el proyecto queda más organizado.

Ahora debemos corregir los imports (VS Code lo suele hacer por nosotros). Comprueba todos los archivos de código, muchas rutas habrán cambiado.

Lo primero que haremos será mover todas las rutas relacionadas con productos al módulo products-routing que se ha generado. Las rutas se declaran desde el método forChild en lugar de forRoot.

El objetivo de los módulos de rutas es separar la gestión de rutas y lo que ello conlleva (componentes de página, guards, resolvers, etc.) del módulo principal (ProductsModule).

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

const routes: Routes = [
  { path: 'products', component: ProductListComponent },
  // :id es un parámetro (id del producto)
  {
    path: 'products/:id',
    component: ProductDetailComponent,
    canActivate: [ProductDetailGuard],
    resolve: {
      product: ProductDetailResolve
    }
  },
  {
    path: 'products/edit/:id',
    canActivate: [ProductDetailGuard],
    canDeactivate: [LeavePageGuard],
    component: ProductEditComponent
  },
];

@NgModule({
  ...
  imports: [RouterModule.forChild(routes)],
  ...
})
export class ProductsRoutingModule { }
```

Después moveremos al módulo de productos todos los componentes, pipes, directivas, etc. que tengan que ver con productos y que ahora mismo se declaran en el módulo de la aplicación.

También necesitamos importar FormsModule para usar directivas como ngModule (HttpClientModule no hace falta importarlo ya que solo tiene servicios, por lo que al importarlo desde AppModule están disponibles en toda la aplicación).

Por ahora vamos también a incluir también (aunque no esté en el directorio del módulo) el componente StarRatingComponent ya que lo usan algunos componentes del módulo actual (Crearemos un módulo aparte para este componente más adelante). Este es el resultado:

```
...
@NgModule({
  declarations: [
    ProductListComponent,
    ProductFilterPipe,
    ProductItemComponent,
    ProductDetailComponent,
    ProductEditComponent,
    StarRatingComponent,
  ],
  imports: [
    CommonModule,
    ProductsRoutingModule,
    FormsModule
  ]
})
```

```
export class ProductsModule { }
```

Y así es como queda AppModule (quitando lo que ya no necesitamos):

```
const APP_ROUTES: Route[] = [
  { path: 'welcome', component: WelcomeComponent },
  // Ruta por defecto (vacía) -> Redirigir a /welcome
  { path: '', redirectTo: '/welcome', pathMatch: 'full' },
  // Ruta que no coincide con ninguna de las anteriores
  { path: '**', redirectTo: '/welcome', pathMatch: 'full' }
];

@NgModule({
  declarations: [
    AppComponent,
    WelcomeComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot(APP_ROUTES),
    ProductsModule // Importamos el módulo en la aplicación
  ],
  providers: [Title],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Creando el módulo de Menú

Vamos a modularizar un poco más nuestra aplicación creando un módulo para el menú (también crearemos un componente llamado menu-top dentro).

```
ng g module menu
```

```
ng g component menu/menu-top
```

Este módulo sólo tiene un componente (el menú superior). Se importará en AppModule y se usará directamente en AppComponent.

Mueve el elemento <nav> al nuevo componente. Este será el resultado:

```
# menu-top.component.ts
@Component({
  selector: 'menu-top',
  templateUrl: './menu-top.component.html',
  styleUrls: ['./menu-top.component.css']
})
export class MenuTopComponent implements OnInit {
  @Input() title;

  constructor() { }

  ngOnInit() { }
}
```

menu.module.ts

```
@NgModule({
  declarations: [
    MenuTopComponent
  ],
  imports: [
    CommonModule,
    RouterModule // Necesario para los enlaces → [routerLink]
  ],
  exports: [
    MenuTopComponent // Exportamos el componente
  ]
})
export class MenuModule { }
```

En AppModule, debemos importar este módulo y poner el selector <menu-top> donde anteriormente estaba el HTML del menú.

app.module.ts

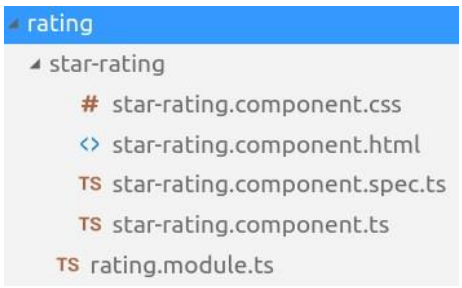
```
...
@NgModule({
  ...
  imports: [
    MenuModule,
    ProductsModule,
    BrowserModule,
    HttpClientModule,
    RouterModule.forRoot(APP_ROUTES)
  ],
  ...
})
export class AppModule { }
```

app.component.html

```
<menu-top [title]="'title'"></menu-top>

<div class="container">
  <router-outlet></router-outlet>
</div>
```

Creando el módulo de valoración (Rating)



Ahora vamos a crear un módulo para nuestro sistema de valoración. Por ahora sólo el componente star-rating, pero podríamos crear otros componentes como sliders, etc. La metodología es la misma que antes, pero en lugar de importar el módulo en AppModule, lo importaremos en cada módulo que necesite usar star-rating. Este es un ejemplo de módulo compartido.

```
ng g module rating
```

rating.module.ts

```
@NgModule({
  declarations: [
    StarRatingComponent
```

```
  ],
  imports: [
    CommonModule
  ],
  exports: [
    StarRatingComponent
  ]
})
export class RatingModule { }
```

```
# products.module.ts
@NgModule({
  ...
  imports: [
    RatingModule,
    CommonModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forChild(PRODUCT_ROUTES)
  ],
  ...
})
export class ProductsModule { }
```

Carga en diferido (lazy-loading) de módulos

Hasta el momento, nos hemos limitado a dividir nuestra aplicación en módulos. Esto permite organizar mejor el código y dividir más fácilmente el trabajo de desarrollo. Sin embargo, cuando hablamos de módulos que contienen rutas de la aplicación (ProductsModule), aún no estamos usando la característica más importante: la carga en diferido o lazy loading.

Este tipo de carga de módulos consiste en que, inicialmente, la aplicación sólo cargará los módulos estrictamente necesarios para mostrar la aplicación al usuario (una página de login, bienvenida, etc.), lo que reduce considerablemente el tamaño inicial y los tiempos de carga.

Activar la carga en diferido es muy sencillo, pero hay que seguir correctamente los pasos. Lo primero es que el módulo de aplicación (AppModule) no debe importar directamente los módulos que se cargan en diferido (borra ProductsModule del array de imports). Y debemos indicar al router de Angular que sea él quien cargue el módulo, en este caso cuando la ruta empieza por el prefijo 'products'.

```
const APP_ROUTES: Route[] = [
  { path: 'welcome', component: WelcomeComponent },
  {
    path: 'products',
    loadChildren: () => import('./products/products.module')
                          .then(m => m.ProductsModule)
  },
  { path: '', component: WelcomeComponent },
  { path: '**', component: WelcomeComponent }
];
```

Ahora, en las rutas de ProductsRoutingModule, ya no necesitamos el prefijo 'products/' ya que estará implícito, por lo que lo borramos:

```
const routes: Route[] = [
  { path: '', component: ProductListComponent },
  {
    path: ':id',
    component: ProductDetailComponent,
    canActivate: [ProductDetailGuard],
    resolve: {
      product: ProductDetailResolve
    }
  },
  {
    path: 'edit/:id',
    component: ProductEditComponent,
    canActivate: [ProductDetailGuard],
    canDeactivate: [LeavePageGuard]
  },
];
```

Ya está. Todo lo que contiene ProductsModule se empaquetará en un archivo JS aparte y se cargará la primera vez que visitemos una ruta que empiece por 'products'.

Definiendo la estrategia de carga en diferido (pre-loading)

El comportamiento por defecto de la carga en diferido es cargar el módulo sólo la primera vez que visitemos una ruta del mismo. Existe otra estrategia, llamada pre-loading o precarga, que empezará a cargar los módulos en segundo plano, inmediatamente después de cargar la aplicación inicial, sin esperar que visitemos una ruta relacionada. Esto mejorará aún más la experiencia del usuario ya que al visitar la ruta, todo lo relacionado habrá sido cargado en memoria previamente, sin perder la ventaja de que, inicialmente, para mostrar la aplicación, sólo se cargó lo estrictamente necesario.

Para activar esta característica, indicamos al router de Angular que aplique la estrategia PreloadAllModules. Por defecto la estrategia es NoPreloading.

```
@NgModule({
  ---
  imports: [
    MenuModule,
    BrowserModule,
    RouterModule.forRoot(APP_ROUTES, {preloadingStrategy: PreloadAllModules})
  ],
  ---
})
export class AppModule { }
```