Semana 6

Curso de Angular

Semana 6
Directivas. Módulos.

Desarrollo de aplicaciones web con Angular
Cefire 2017/2018
Autor: Arturo Bernal Mayordomo

Index

Directivas

Las directivas son elementos o atributos creados por nosotros en Angular y que añaden o modifican funcionalidad a la plantilla HTML. Hay 3 tipos de directivas:

Directivas de componente

Una directiva de componente se define por la propiedad **selector** en el decorador @Component. Cuando un elemento con el mismo nombre se crea en la plantilla HTML, se instancia la clase del componente asociado y su plantilla se inserta dentro del mismo. Esto es lo que hemos estado haciendo hasta ahora.

```
@Component({
  selector: 'star-rating',
  templateUrl: './star-rating.component.html',
  styleUrls: ['./star-rating.component.css']
})
export class StarRatingComponent implements OnInit {
  ...
}
```

```
<star-rating *ngIf="product" [rating]="product.rating"
             (ratingChanged)="changeRating($event)"></star-rating>
```

Directivas de atributo

Las **directivas de atributo** son atributos creados para modificar el comportamiento de un elemento en la plantilla. Las directivas de Angular **NgClass** y **NgStyle** son ejemplos de esto.

Vamos a crear una directiva de este tipo. En este caso servirá para cambiar el color de fondo de un elemento cuando hagamos click en ella. Para generar la directiva, creamos un directorio llamado **directives** y ejecutamos dentro:

```
ng g directive setColor
```

```
@Directive({
  selector: '[setColor]'
})
export class SetColorDirective {
  constructor() { }
}
```

La clase de la directiva que hemos creado (SetColorDirective) debe incluirse en la sección **declarations** del módulo de la aplicación. Una vez creada se la podemos aplicar a cualquier elemento HTML. Por ejemplo:

```
<h1 [setColor]="'yellow'">{{title}}</h1>
```

Cuando colocamos el atributo de la directiva (en este caso **[setColor]**), esta se aplica al elemento en cuestión. Podemos pasarle parámetros extras y producir eventos (además del color) igual que hacíamos con los componentes anidados (**@Input**, **@Output**). Para procesar los eventos del elemento dentro de la directiva 'click', 'mouseenter', etc., usamos el decorador **@HostListener**.

app.component.ts

```
@Component({
  ...
})
export class AppComponent {
  title = 'app works!';
  color = 'yellow';
}
```

app.component.hmtl

```
<h1 [setColor]="color">{{title}}</h1>
<select [(ngModel)]="color">
  <option value="yellow">Yellow</option>
  <option value="red">Red</option>
  <option value="#06F">Blue</option>
</select>
```

set-color.directive.ts

```
@Directive({
  selector: '[setColor]'
})
export class SetColorDirective {
  private isSet: boolean;
  @Input('setColor') color: string;

  constructor(private elem: ElementRef) {
    this.isSet = false;
  }

  @HostListener('click') onClick() {
    if(this.isSet) {
      this.elem.nativeElement.style.backgroundColor = "";
    } else {
      this.elem.nativeElement.style.backgroundColor = this.color;
    }
    this.isSet = !this.isSet;
  }
}
```

https://angular.io/docs/ts/latest/guide/attribute-directives.html

Directivas estructurales

Las **directivas estructurales** se utilizan para modificar la estructura del DOM. Normalmente indican si un elemento debe aparecer en el DOM o no. No es muy común crear directivas estructurales, pero puede sernos bastante útil.

NgSwitch

Ejemplos de este tipo de directivas son **NgIf** → *ngIf y **NgFor** → *ngFor. Existe otra llamada **NgSwitch** → [ngSwitch] que no hemos visto y que permite crear una estructura **switch** en la plantilla HTML.

```
<span [ngSwitch]="property">
  <span *ngSwitchCase="'val1'">Value 1</span>
  <span *ngSwitchCase="'val2'">Value 2</span>
  <span *ngSwitchCase="'val3'">Value 3</span>
  <span *ngSwitchDefault>Other value</span>
</span>
```

NgIf (else)

La directivas estructurales necesitan del elemento **<ng-template>** para funcionar. Sin embargo, con el asterisco(*) delante no es necesario ponerlo, ya que Angular lo genera por nosotros. Este elemento <ng-template> desaparece cuando la directiva se procesa. Estas dos formas de escribir **ngIf** son equivalentes:

```
<div *ngIf="condition" >{{somevalue}}</div>
```

```
<ng-template [ngIf]="condition">
  <div>{{somevalue}}</div>
</ng-template>
```

Si queremos crear un bloque else, necesitamos otro elemento **ng-template**, sólo que esta vez no lo podemos omitir, ya que está fuera del elemento al que afecta la directiva ngIf. Necesitamos crear una referencia para este nuevo elemento, poniendo la almohadilla (#referencia), y usarla para definir la condición **else**.

```
<div *ngIf="show; else elseBlock">La condición es verdadera</div>
<ng-template #elseBlock>La condición es falsa</ng-template>
```

Creando una directiva estructural

Vamos a crear una directiva estructural que repita un elemento **n veces**. La directiva se llamará **repeatTimes**:

```
ng g directive repeat-times
```

Así es como la utilizaremos una vez implementada:

```
<p *apRepeatTimes="3; let n = current">
  {{n}} welcome works!
</p>
```

Para obtener el valor recibido (3) dentro de la directiva, tenemos que crear un *setter* que tenga el mismo nombre que la misma. Dentro de ese método debemos llamar a createEmbeddedView para dibujar el elemento que contiene la directiva (en este caso con un bucle lo dibujamos ***num*** veces). También podemos pasar valores a la plantilla dentro de un objeto, en este caso **current**. En la plantilla recogemos este valor en una variable llamada n con **let n = current**.

```
import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';

@Directive({
  selector: '[repeatTimes]'
})
export class RepeatTimesDirective {
  @Input()
  set repeatTimes(num: number) {
    this.viewContainer.clear(); // Si cambian los datos de entrada borramos

    for (let i = 0; i < num; i++) {
      this.viewContainer.createEmbeddedView(this.templateRef, {
        current: i + 1
      });
    }
  }

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) {}
```

1 welcome works!

2 welcome works!

3 welcome works!

```
}
```

Vamos a crear otra directiva estructural. Esta vez muy similar a **ngFor**, pero que también acepta una función para filtrar el array de elementos:

```
<p *apForFilter="let person from persons by filter">
  {{person | json}}
</p>
```

Este es el filtro que aplicamos:

```
filter = (p) => p.age >= 18;
```

{ "name": "Clara", "age": 22 }

{ "name": "John", "age": 36 }

Tenemos 2 diferencias con la anterior directiva. Los datos de entrada (persons y filter) vienen precedidos de las palabras **from** y **by**. Para obtenerlos debemos crear setters con dichos sufijos (**forFilterFrom** y **forFilterBy**). La otra diferencia es como enviamos cada persona a la plantilla cuando recorremos el array. En este caso, con esta sintaxis en la plantilla, se envía en un valor llamado **$implicit**.

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[forFilter]'
})
export class ForFilterDirective {
  @Input()
  set forFilterFrom(array: any[]) {
    this.items = array;
  }

  @Input()
  set forFilterBy(filter: (item: any) => boolean) {
    this.viewContainer.clear(); // Si cambian los datos de entrada borramos
    this.items.filter(filter).forEach(elem => {
      this.viewContainer.createEmbeddedView(this.templateRef, {
        $implicit: elem
      });
    });
  }

  items: any[] = [];

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) {}
}
```

ng-container

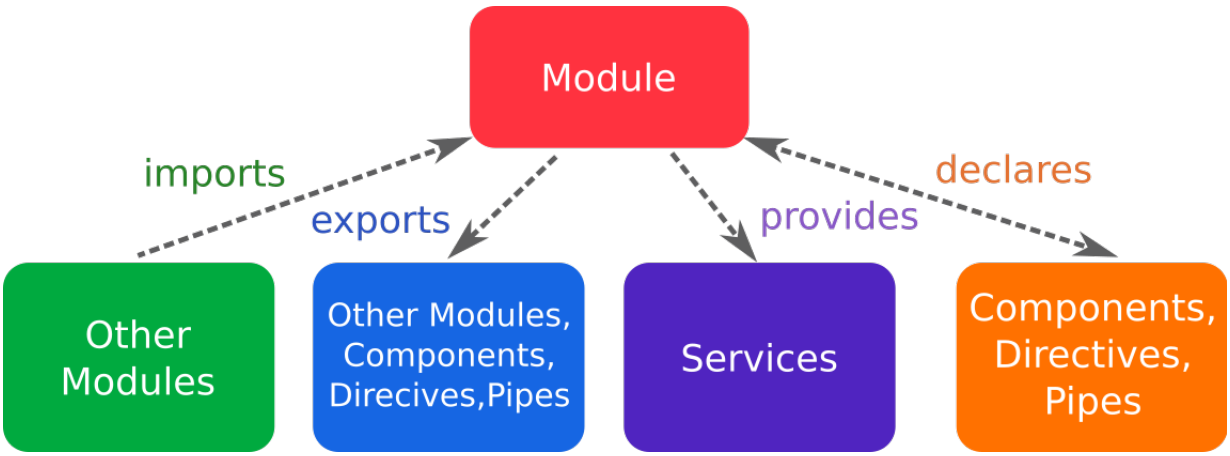
El elemento **ng-container** sirve para usar directivas estructurales. La diferencia frente a usarlas con un elemento normal, es que este desaparece cuando ha sido procesada. Es útil cuando quieres combinar 2 o más directivas estructurales ya que esto no se puede hacer en un mismo elemento.

```
<ng-container *ngFor="let person of persons"> <!-- Desaparece -->
  <ng-container *ngIf="person.age >= 18"> <!-- Desaparece -->
    <p>{{person | json}}</p> <!-- Sólo quedará este elemento -->
  </ng-container>
</ng-container>
```

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 bootstrap en un módulo sólo se suele utilizar en el módulo de la aplicación **AppModule**, y sirve para indicar a Angular qué componente es el principal (**AppComponent**) y se cargará el primero. La plantilla de este componente es la que se cargará 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 un módulo 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, e incluso otros módulos independientemente de si los importa previamente o no (por ejemplo un módulo creado para agrupar la importación de varios módulos en un sólo sitio).

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 (esto incluye 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 en la aplicación. Todos los 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 { }
```

Nota: 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.

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

Dividiendo nuestra aplicación en módulos

División por características

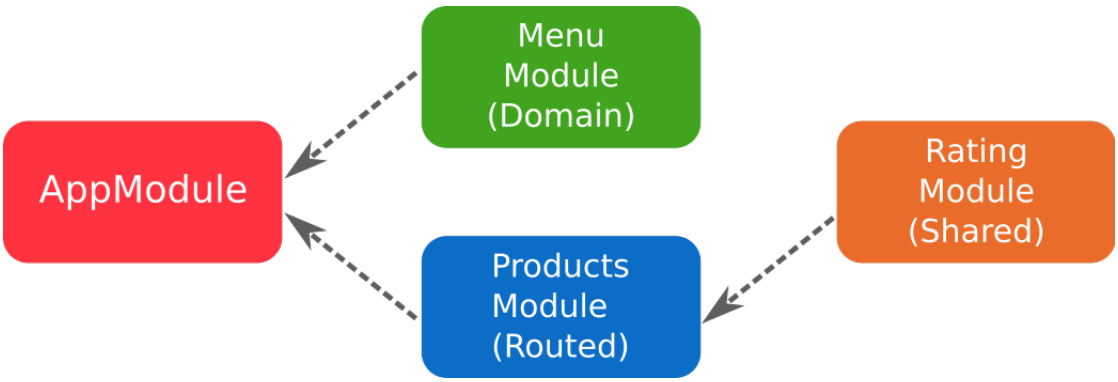
Para organizar una aplicación en módulos hay que seguir ciertas reglas. Algunas de las que se recomiendan son las siguientes:

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

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

- **Módulos compartidos**: 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
```

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

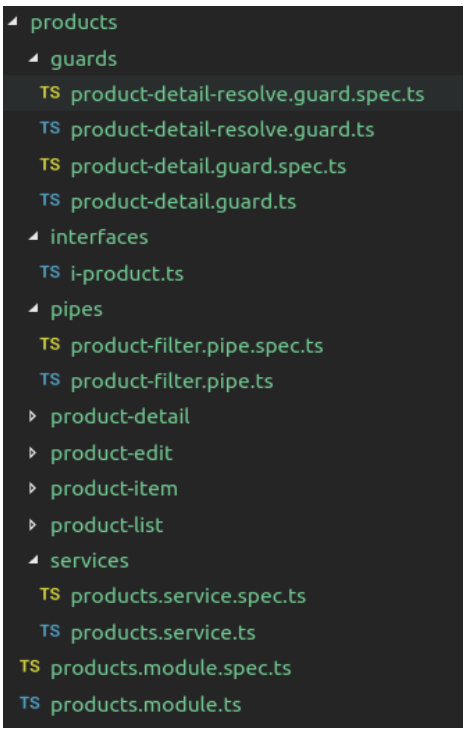
Este es el módulo que ha creado (**products/products.module.ts)**:

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

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
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. **Muy importante**: Sólo debemos declarar los servicios en un módulo separado de AppModule si estamos 100% seguros que no se van a necesitar en ningún componente que no forme parte del módulo donde se declara.



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 (comprueba todos los archivos de código, muchas rutas habrán cambiado).

No vamos a exportar componentes en este módulo, por lo que no podremos usarlos fuera del mismo, como por ejemplo en las rutas de AppModule. Para resolver este, vamos a importar RouterModule en el módulo de productos y vamos a crear rutas dentro del mismo. Las rutas se declaran con el método **forChild** en lugar de **forRoot**.

También necesitamos importar**FormsModule** para usar directivas como **ngModule** y **HttpClientModule** para el servicio de productos (no pasa nada por importarlos en varios módulos). Por ahora vamos también a declarar (aunque no esté en el directorio del módulo) el componente StarRatingComponent ya que lo usan algunos componentes del módulo actual (Crearemos un módulo para este componente más adelante). Este es el resultado:

```
export const PRODUCT_ROUTES: Route[] = [
  { path: 'products', component: ProductListComponent },
  {
    path: 'products/:id', component: ProductDetailComponent,
    canActivate: [ProductDetailGuard],
```

```
    resolve: {
      product: ProductDetailResolve
    }
  },
  {
    path: 'products/edit/:id', component: ProductEditComponent,
    canActivate: [ProductDetailGuard], canDeactivate: [CanDeactivateGuard]
  },
];

@NgModule({
  declarations: [
    ProductListComponent,
    ProductItemComponent,
    ProductFilterPipe,
    ProductDetailComponent,
    ProductEditComponent,
    StarRatingComponent
  ],
  imports: [
    CommonModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forChild(PRODUCT_ROUTES)
  ],
  providers: [
    ProductService,
    ProductDetailGuard,
    ProductDetailResolve
  ]
})
export class ProductsModule { }
```

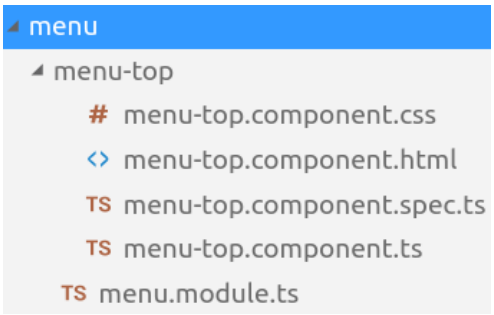
Y así es como queda **AppModule** (quitando lo que ya no necesitamos):

```
export const APP_ROUTES: Route[] = [
  { path: 'welcome', component: WelcomeComponent },
  { path: '', component: WelcomeComponent },
  { path: '**', component: WelcomeComponent }
];

@NgModule({
  declarations: [
    AppComponent,
    WelcomeComponent,
  ],
  imports: [
    ProductsModule, // Importamos el módulo en la aplicación
    BrowserModule,
    RouterModule.forRoot(APP_ROUTES)
  ],
  providers: [
    Title,
    CanDeactivateGuard,
  ],
  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).



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
  ],
  exports: [
    MenuTopComponent // We need to export the component
  ]
})
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

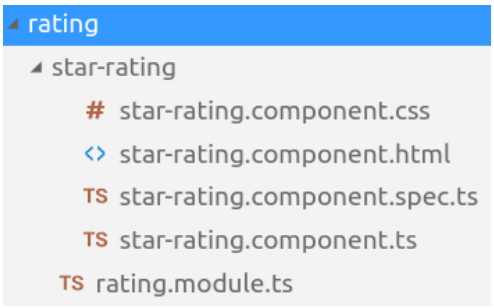
```
...
@NgModule({
  ...
  imports: [
    MenuModule,
    ProductsModule,
    BrowserModule,
    RouterModule.forRoot(APP_ROUTES)
  ],
  ...
})
export class AppModule { }
```

 # 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 sistemas como barras deslizantes, etc. La metodología es la misma que antes pero en lugar de importarlo en AppModule lo importaremos en ProductsModule que alguno de sus componentes necesitan usar star-rating. Este es un ejemplo de módulo compartido, pensado para importarlo en cualquier módulo que lo necesite.

```
# 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 iniciales.

Activar la carga en diferido es muy sencillo, pero hay que seguir correctamente los pasos. Lo primero es que el módulo de aplicación no debe importar directamente los módulos que se cargan así (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: './products/products.module#ProductsModule'},
 { path: '', component: WelcomeComponent },
 { path: '**', component: WelcomeComponent }
];
```

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

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

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 { }
```