Tema 4 - Angular

Angular Router.

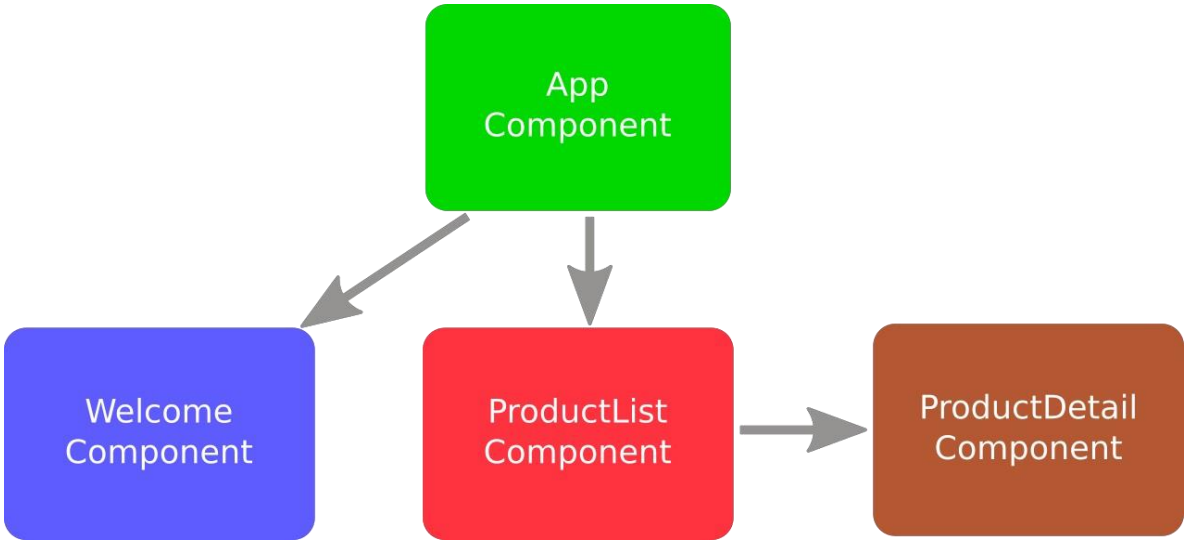
Desarrollo web en entorno cliente
IES Pere Maria Orts I Bosch

Índice

Conceptos básicos

Como explicamos al principio, Angular es un framework para crear SPA (Single Page Application, o Aplicación de Una Página). Esto significa que sólo se va a cargar el documento index.html en el navegador, pero no significa que nuestra aplicación no pueda estar dividida en varias vistas o páginas. La navegación entre las diferentes secciones o páginas de nuestra aplicación la gestiona Angular Router.

En nuestra aplicación de ejemplo, vamos a añadir 2 páginas más, además del listado de productos. Estas vistas serán una página de bienvenida (welcome) y otra para ver el detalle de un producto (product-detail).



Lo primero de todo será crear los componentes que gestionarán cada una de las nuevas páginas de la aplicación:

```
ng g component welcome

ng g component product-detail
```

Antes de establecer las rutas de nuestra aplicación para poder navegar entre las diferentes secciones o páginas, importamos el módulo RouterModule en el módulo de la aplicación:

```
import { RouterModule } from '@angular/router';

@NgModule({
  ---
  imports: [
    ---
    RouterModule
  ],
  ---
})
export class AppModule { }
```

Esto no es suficiente. Las diferentes rutas o páginas de la aplicación, se establecen llamando al método forRoot() del módulo que acabamos de importar. Vamos a crear un array de rutas antes del módulo (@NgModule). Más adelante veremos como reestructurar nuestro proyecto en módulos y crearemos un módulo específico para las rutas.

```
import { RouterModule, Route } from '@angular/router';
---
const APP_ROUTES: Route[] = [
  { path: 'welcome', component: WelcomeComponent },
  { path: 'products', component: ProductListComponent },
  // :id es un parámetro (id del producto)
  { path: 'products/:id', component: ProductDetailComponent },
  // 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({ ... })
```

Las rutas definidas son:

- /welcome → Carga el componente WelcomeComponent.
- /products → Carga el componente ProductListComponent.
- /products/:id → Carga el componente ProductDetailComponent. Esta ruta recibirá un parámetro (la id del producto a mostrar).
- La ruta por defecto (ruta vacía ''), y cuando la ruta no coincida con ninguna anterior (ruta comodín '**') redirigirán a /welcome.

Así cargamos las rutas en la aplicación (app.module.ts):

```
---
  imports: [
    ---
    RouterModule.forRoot(APP_ROUTES)
  ],
---
```

Ahora falta activar el router en la plantilla HTML. Para indicar donde se debe cargar el contenido de cada ruta (el componente asociado a la misma), creamos un elemento llamado <router-outlet> en la plantilla principal (AppComponent). Dentro, se cargará un contenido u otro en función de la ruta actual, permaneciendo estático el resto de elementos en esa plantilla.

Los enlaces para navegar a las diferentes rutas, en lugar de utilizar href, utilizarán la directiva de Angular routerLink. El valor será un array con la ruta a la que apuntan como primer parámetro.

```
# app.component.html

<nav class="navbar navbar-expand navbar-dark bg-dark mb-4">
  <a class="navbar-brand" href="#">{{title}}</a>
```

```
    <ul class="navbar-nav  mr-auto">
      <li class="nav-item">
        <a class="nav-link" [routerLink]="['/welcome']"
                [routerLinkActive]="['active']">Welcome</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" [routerLink]="['/products']"
                [routerLinkActive]="['active']">Products</a>
      </li>
    </ul>
  </nav>
  <div class="container-fluid">
    <router-outlet></router-outlet>
  </div>
```

La directiva routerLinkActive se utiliza para asignar una o varias clases CSS al enlace cuando la ruta está activa. En este caso la clase 'active'.

Cambiando el título de la página

El título de la página está definido en index.html (sección<head>). Angular sólo tiene control sobre lo que pasa dentro de la etiqueta <app-root> (el componente principal de la aplicación). Por ello no podemos usar interpolación para establecer el título de la sección actual <title>{{title}}</title>.

Para solucionar el problema, Angular tiene un servicio llamado Title, cuya función es simplemente cambiar el título de la página. Vamos a ver como establecer un título cuando estamos en la página de bienvenida (welcome).

El servicio debe ser declarado previamente en el módulo de la aplicación (array providers) para inyectarlo en el contenedor de dependencias de la aplicación, ya que a diferencia del servicio que hemos creado nosotros, no se "autoinyecta", con el atributo → providedIn: 'root'.

```
# app.module.ts
import { BrowserModule, Title } from '@angular/platform-browser';
---
@NgModule({
  ---
  providers: [Title],
  ---
})
export class AppModule { }

# welcome.component.ts
import { Title } from '@angular/platform-browser';
---
export class WelcomeComponent implements OnInit {

  constructor(private titleService: Title) { }

  ngOnInit() {
    this.titleService.setTitle('Bienvenido a Angular Products!');
  }

}
```

Conceptos más avanzados

Vamos a ver conceptos más avanzados del sistema de rutas de Angular como: activar una ruta por código, permitir o no el acceso a una ruta (por ejemplo, sólo si hemos iniciado sesión), o detectar cuando vamos a abandonar la página actual y actuar en consecuencia (avisando al usuario, guardando cambios, etc.).

Pasando parámetros a una ruta

En algunas páginas, como puede ser el detalle de un producto, tenemos que indicar de alguna manera qué producto queremos cargar. La ruta definida en este caso es 'products/:id'. Los elementos de la URL precedidos de ':' son parámetros que recibe la ruta. Para cargar el producto con id = 4 → products/4.

Vamos a ver como pasar la id del producto en la ruta. Para ello, vamos a establecer como enlace la descripción del producto:

```
# product-item.component.html
...
<div class="col-4">
  <a [routerLink]="['/products', product.id]">
    {{ product.description }}
  </a>
</div>
...
```

Como puedes observar, después del nombre de la ruta (donde se omiten los parámetros), se establecen en orden los parámetros que dicha ruta recibe. Ahora vamos a ver como recoger dicho parámetro en el componente que gestiona la ruta → ProductDetailComponent. Para ello inyectamos el servicio ActivatedRoute.

```
import { Component, OnInit } from '@angular/core';
import { IProduct } from '../interfaces/i-product';
import { ActivatedRoute } from '@angular/router';
import { ProductsService } from '../services/products.service';

...
export class ProductDetailComponent implements OnInit {
  product: IProduct;

  constructor(private route: ActivatedRoute,
              private productsService: ProductsService) { }

  ngOnInit() {
    const id = +this.route.snapshot.params.id; // Recibimos parámetro
    this.productService.getProduct(id)
      .subscribe(
        p => this.product = p,
        error => console.error(error)
      );
  }
}
```

Mostrando datos asíncronos

Por ejemplo, cuando navegamos a la página de detalle de un producto, primero debemos obtener sus datos del servidor. Esto es una operación asíncrona, por lo que hasta que el servicio no nos devuelva los datos, el valor del producto será undefined. Si en este estado, Angular intenta acceder a una propiedad para mostrarla (ej: product.description), saltará un error.

Al igual que podemos inicializar un array vacío hasta obtener los datos, se podría crear un objeto con las propiedades vacías de forma temporal, y cuando se asignara el de verdad, aparecerían los datos en la vista.

Otra opción (y muchas veces la mejor) es utilizar la directiva *ngIf, y no mostrar la estructura hasta que el objeto tenga un valor válido.

```
<div class="card"  *ngIf="product">
    ...
</div>
```

Si necesitamos mostrar por alguna razón la estructura sin los valores rellenos, se puede usar el operador opcional '?' después del nombre del objeto (ejemplo: product?.description). Esto indica a Angular que sólo acceda a la propiedad cuando product tenga un objeto válido. Aún así no podremos evitar muchas veces usar *ngIf en componentes que necesiten recibir el objeto o alguna propiedad del objeto (star-rating por ejemplo) para no crearlos hasta entonces.

product-detail.component.html

```
<div class="card">
  <div class="card-header bg-primary text-white">
    {{product?.description}}
  </div>
  <div class="card-block p-3 text-center">
    <img [src]="product.imageUrl" alt="">
    <div>Price: {{ product?.price | currency:'EUR':'symbol' }}</div>
    <div>Available since: {{ product?.available | date:'dd/MM/y' }}</div>
    <div>
      <star-rating *ngIf="product" [rating]="product.rating"
                   (ratingChanged)="changeRating($event)"></star-rating>
    </div>
    <div>
      <button type="button" class="btn btn-default" (click)="goBack()">
        Volver
      </button>
    </div>
  </div>
</div>
```

Activar una ruta mediante código

Hemos añadido un botón de "Volver" para regresar al listado en la página de detalle. En lugar de crear un enlace y usar [routerLink], vamos a navegar a la ruta por código. Para ello, primero debemos inyectar el servicio Router en el componente asociado:

```
...
```

```
import { ActivatedRoute, Router } from '@angular/router';
---
export class ProductDetailComponent implements OnInit {
  ---
  constructor(private router: Router,...) { }
  ---
  goBack() {
    this.router.navigate(['/products']);
  }

}
```

Como podemos observar, activar una ruta con código es similar a usar un enlace. Debemos pasar al método navigate un array cuyo primer elemento es la ruta, y si necesitara parámetros extra, los incluiremos en dicho array.

Permitir navegar a una ruta con CanActivate

En una aplicación Angular, podemos controlar si el usuario puede acceder a una página, o abandonarla, usando *guards*. Son un tipo de servicios que se utilizan en combinación con el router de Angular. Por ejemplo, si queremos impedir que un usuario que ha iniciado sesión acceda a la parte privada de la aplicación, usaremos un tipo de *guard* llamado CanActivate.

Vamos a crear un *guard* de ejemplo que sólo nos dejará acceder al detalle de un producto si la id que recibimos es numérica. Ejecutamos lo siguiente:

```
ng g guard guards/product-detail
```

```
? Which interfaces would you like to implement?
❯◯ CanActivate
 ◯ CanActivateChild
 ◯ CanLoad
```

Cuando nos pregunte, seleccionaremos implementar CanActivate.

```
...

@Injectable({
  providedIn: 'root'
})
export class ProductDetailGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> |
                  Promise<boolean | UrlTree> | boolean | UrlTree {
    return true;
  }
}
```

Para asignarlo a una ruta (en este caso 'product-detail'), creamos la propiedad canActivate, que es un array (podríamos añadir varios *guards*, y todos se deberían validar en orden para acceder).

```
{
    path: 'products/:id',
    canActivate: [ProductDetailGuard],
    component: ProductDetailComponent
}
```

Ahora implementaremos el método CanActivate. Necesitamos comprobar si la id recibida es numérica, y devolver true en ese caso o false en el contrario (boolean). Aunque el método permite devolver valores asíncronos (por ejemplo el resultado de llamar a un servicio web) siempre que el valor sea de tipo Promise<boolean> o Observable<boolean>. Desde Angular 7 también se puede devolver un objeto URLTree para redirigir a otra página.

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree,
Router } from '@angular/router';

@Injectable({
  providedIn: 'root'
})
export class ProductDetailGuard implements CanActivate {
  constructor(private router: Router) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean | UrlTree {
      const id = +route.params.id;
      if (isNaN(id) || id < 1) {
        return this.router.createUrlTree(['/productos']);
      }
      return true;
  }
}
```

Prueba ahora a acceder a la página con una id no válida, como por ejemplo:

http://localhost:4200/products/hola

Permitir abandonar la ruta actual con CanDeactivate

El *guard* CanDeactivate evita que un usuario pueda abandonar la ruta actual si no se cumplen ciertas condiciones. Por ejemplo, el usuario está editando la información de un producto y no ha guardado los cambios. Podríamos utilizar este tipo de *guard* para mostrarle un aviso para que confirme si quiere abandonar la página sin guardar los cambios.

Vamos a crear una nueva ruta para editar un producto ('/product/edit/:id' → ProductEditComponent) y el componente asociado (product-edit) también. Este componente contendría un formulario para editar el producto seleccionado (no hace falta implementarlo). Cuando el usuario intente abandonar esta página le aparecerá un mensaje de confirmación.

```
{
      path: 'products/edit/:id',
      canActivate: [ProductDetailGuard],
      component: ProductEditComponent
```

```
}
```

En la página de detalle de un producto (product-detail.component.ts), añadiremos un botón (a continuación del botón de volver) que nos enviará a la página que acabamos de crear:

```
<button type="button" class="btn btn-success ml-4" (click)="edit()">
      Editar producto
</button>
```

Y en el componente añadimos:

```
  edit() {
    this.router.navigate(['/products/edit', this.product.id]);
  }
```

Vamos a crear primero el guard, aunque seleccionaremos CanActivate, ya que Angular CLI no tiene automatizada la creación de este tipo de guards. Cambiamos la interfaz a CanDeactivate (y borramos el método canActivate). Esta interfaz además, necesita definir sobre qué componente va a actuar (ProductEditComponent).

```
ng g guard guards/leave-page
```

```
? Which interfaces would you like to implement?
  ◯ CanActivate
  ◯ CanActivateChild
❯ ◉ CanDeactivate
  ◯ CanLoad
```

```
@Injectable({
  providedIn: 'root'
})
export class LeavePageGuard implements CanDeactivate<ProductEditComponent> {
  canDeactivate(
    component: ProductEditComponent,
    currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot,
    nextState?: RouterStateSnapshot
  ): boolean | UrlTree {
    // Recibimos el componente por parámetro → podemos acceder sus métodos
    return confirm('¿Quieres abandonar la página?. Los cambios no se
guardarán.');
  }
}
```

El método canDeactivate debe devolver un booleano (directamente o envuelto en una promesa u observable) o un objeto UrlTree para redirigir a otra página. En la configuración de la ruta con el componente establecido, debemos indicar que queremos usar este *guard* como *canDeactivate*.

```
{
   path: 'products/edit/:id',
   canActivate: [ProductDetailGuard],
   canDeactivate: [LeavePageGuard],
   component: ProductEditComponent
}
```

Como puedes observar, al tener que especificar la clase de componente para este tipo de *guard*, sólo podremos reutilizarlo en las rutas con dicho componente (normalmente una). Sin embargo, podríamos crear un *CanDeactivate* retutilizable.

Para ello vamos a crear una interfaz llamada ComponentDeactivate, para que los componentes que la implementen puedan utilizar este guard. Dichos componentes deberán implementar un método llamado canDeactivate que devolverá un booleano. En lugar de usar un componente en concreto, usaremos la interfaz como tipo en el *guard*.

```
export interface ComponentDeactivate {
  canDeactivate: () => boolean | UrlTree | Observable<boolean | UrlTree> |
Promise<boolean | UrlTree>;
}

@Injectable()
export class LeavePageGuard implements CanDeactivate<ComponentDeactivate> {
  canDeactivate(
    component: ComponentDeactivate,
    currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot,
    nextState?: RouterStateSnapshot
  ): boolean | UrlTree | Observable<boolean | UrlTree> | Promise<boolean |
UrlTree> {
    // Recibimos el componente por parámetro, por lo que podemos acceder sus
métodos
    return component.canDeactivate ? component.canDeactivate() : true;
  }
}
```

Ahora sólo necesitamos implementar la interfaz ComponentDeactivate en los componentes donde queramos utilizar este *guard*. Cada componente gestionará en su método canDeactivate las condiciones para salir de la ruta.

```
export class ProductEditComponent implements OnInit, ComponentDeactivate {
  ...
  canDeactivate() {
    return confirm('¿Quieres abandonar la página?. Los cambios no se
guardarán.');
  }

}
```

Obteniendo datos antes de activar una ruta con Resolve

A veces conviene obtener datos por adelantado (generalmente del servidor) antes de cargar una página. En lugar de obtener el producto a partir de la id cuando cargamos la página de ProductDetailComponent, vamos a obtenerlo antes de cargarla con un *guard* de tipo Resolve. De esta manera, sabremos que la página se cargará cuando tengamos la información del producto ya disponible (y comprobemos que se ha cargado de forma correcta).

```
ng g service resolvers/product-detail
```

Este tipo de *guard* puede devolver cualquier tipo de dato directamente o dentro de una Promesa u Observable (Angular se suscribirá automáticamente para obtener el

valor). También podemos prevenir errors (catchError) y actuar en consecuencia, como por ejemplo navegar a otra página. Vamos a modificar el nombre del servicio recién creado cambiándole el sufijo Service por Resolve (también el nombre del archivo) e implementando la interfaz Resolve<IProduct> (product-detail.resolve.ts):

```
...
@Injectable({
  providedIn: 'root'
})
export class ProductDetailResolve implements Resolve<Producto> {

  constructor(
    private productsService: ProductsService,
    private router: Router) { }

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
Observable<IProduct> {
    const id = +route.params.id;
    return this.productsService.getProduct(id).pipe(
      catchError(e => {
        this.router.navigate(['/products']);
        return of(null);
      })
    );
  }
}
```

Si ocurre algún error, nos redirigirá a la página de productos (esto también se podría controlar con un guard de tipo CanActivate ya que se ejecuta después del Resolve). En caso contrario, devolvemos el producto obtenido por el servidor dentro del observable. Así es como implementamos un servicio de tipo Resolve en la ruta:

```
 {
      path: 'products/:id',
      canActivate: [ProductDetailGuard],
      component: ProductDetailComponent,
      resolve: {
            product: ProductDetailResolve
      }
}
```

En la clase ProductDetailComponent, en lugar de llamar al servicio web para obtener el producto (ya no hay necesidad), lo podemos obtener directamente de los datos de la ruta (ya que el Resolve lo ha precargado).

```
...
export class ProductDetailComponent implements OnInit {
  ...
  ngOnInit() {
    this.product = this.route.snapshot.data.product;
  }
  ...
}
```

Ya podríamos quitar la directiva *ngIf de la plantilla de detalle de producto. Prueba a pasar una id de un producto que no exista en la url, verás como te redirige al listado de productos.