Routing en Angular: Guía completa: Parte 6

20 enero, 2021 por Ro

¡Seguimos con esta *Routing Saga!* Si acabas de aterrizar aquí, puedes consultar la parte anterior para ubicarte.

Índice

Cómo externalizar las rutas en un archivo a parte

Si nos fijamos en nuestro *AppModule*, vemos que la parte del *routing* ocupa mogollón de espacio. Como regla general, si tenemos más de un par de rutas, merece la pena **moverlas a un archivo independiente** y vincularlo al *AppModule*. Vamos a ello.

1. A la altura del *AppModule*, creamos un archivo (un módulo) para las rutas, normalmente llamado ***app-routing.module.ts***. Aprendí todo sobre los módulos de Angular en este completísimo curso de Max.

2. Al módulo le añadimos el decorador *ngModule*.

```
import { NgModule } from "@angular/core";

@NgModule({})
export class AppRoutingModule { }
```

3. En el *app.module.ts*, borramos el *RouterModule*, cortamos la variable *appRoutes* y la pegamos después de los *imports* del *app-routing.module.ts*.

4. Importamos todos los componentes y paquetes necesitarios, tal y como estaban en el *app.module.ts*.

No debemos añadir un *array* de *declarations*, porque esos componentes **ya están declarados** en el *app.module.ts*.

5. Añadimos un *array* de *imports* en el *ngModule*, usamos el *RouterModule* con el método *forRoot* y le pasamos nuestras rutas (*appRoutes)*.

Para poder usar las rutas de nuevo, debemos **avisar al *AppModule* de que el archivo *AppRoutingModule* existe**. Para eso tenemos el *array* de *exports*. Esta es la forma que Angular nos da para que podamos exponer este módulo (o las partes que queramos de él) a otros módulos, para así poder usarlo donde lo necesitemos.

En este caso, queremos exponer el *RouterModule* así que se lo indicamos en el *array* de *exports*.

```
import { NgModule } from "@angular/core";
import { RouterModule, Routes } from "@angular/route
import { HomeComponent } from "./home/home.component
import { PageNotFoundComponent } from "./page-not-fo
import { EditServerComponent } from "./servers/edit-
import { ServerComponent } from "./servers/server/se
import { ServersComponent } from "./servers/servers.
import { UserComponent } from "./users/user/user.com
import { UsersComponent } from "./users/users.compon

const appRoutes: Routes = [
  { path: 'users', component: UsersComponent, childr
    { path: ':id/:name', component: UserComponent },
  ]
  },
  { path: '', component: HomeComponent },
  { path: 'servers', component: ServersComponent, ch
    { path: ':id/edit', component: EditServerCompone
    { path: ':id', component: ServerComponent }
```

```
  ]
  },
  { path: 'not-found', component: PageNotFoundCompor
  { path: '**', redirectTo: '/not-found'}
];

@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes)
  ],
  exports: [RouterModule]
```

Con estos cambios, en el *AppModule* ya podemos **importar nuestro archivo de rutas**. Lo hacemos en el *array* de los *imports*.

```
imports: [
  BrowserModule,
  FormsModule,
  AppRoutingModule
],
```

De esta manera, nuestra app funciona igual que antes, pero ahora **el código está un poco más organizado** 🤓.

Introducción a los *Route Guards*

Con este nombre tan molón digno de un reino de El señor de los anillos, los guardianes de rutas juegan un papel fundamental en el ecosistema del *routing*.

Los guardianes de rutas (en inglés, *route guards)* son configuraciones de código que podemos hacer **antes de acceder a una ruta** o una vez abandonamos una ruta.

Hagamos un ejemplo para entenderlo bien. Supongamos que sólo queremos que un usuario tenga acceso al componente *Server* y al componente *EditServer* **si está "registrado"** en nuestra app. Lo pongo entre comillas porque vamos a simular ese proceso de registro (en inglés, *login)*.

Para eso, necesitaremos hacer esta comprobación antes de acceder a cualquier ruta relacionada con el *ServerComponent* o el *EditServerComponent*. Una **opción poco eficiente** de hacer esto sería de manera manual, en el *ngOnInit* de esos dos componentes. Además, tampoco sería escalable.

Debemos tomar otro camino, como es el usar una herramienta que viene incluída en el *Router* de Angular, que se encarga de ejecutar un código antes de que el componente se cargue: **el *canActive Guard***.

Cómo proteger rutas con *canActivate*

Vamos a crear nuestro guardián de rutas, un archivo que tomará forma de *service*. Una **práctica común es llamarlo *auth-guard*** (*auth* viene de *authentication*, "autenticación" en español).

1. Creamos el archivo a la altura del *AppComponent*.

2. Exportamos la clase *AuthGuard*. Podríamos llamarla *AuthGuardService*, para ser más específicos, pero no es necesario.

3. Implementamos la ***interface* de *CanActivate***, incluída en el paquete de *angular/router*. Esta *interface* nos obliga a tener un método llamado *canActivate*, que recibe dos argumentos:

- la **ruta**, que será de tipo *ActivatedRouteSnapshot*
- el **estado del *router***, que será de tipo *RouterStateSnapshot*

No olvides importarlos.

Estos parámetros nos los proporcionará Angular, pero **nuestro trabajo aquí consiste en gestionarlos**, porque recuerda que este código se ejecutará antes de ir a una ruta o cuando salgamos de alguna.

El método *canActivate* también nos puede devolver varias cosas (**son alternativas**, por eso debemos usar el *or operator*):

- un *observable*, que resultará en un *boolean*.
- una promesa, que resultará en un *boolean*.
- sencillamente un *boolean*, tal cual.

Esto significa que *canActivate* puede ejecutarse de manera:

asíncrona (devolviendo un *observable* o una promesa).
Ej: si la ruta tarda unos segundos en cargar porque debe conectarse a un servidor.

o **síncrona** (devolviendo un *boolean).*
Ej: si toda la actividad sucede en el lado del cliente.

No olvidemos añadir el **decorador *@Injectable*** para configurar correctamente un servicio.

```
1  import { Injectable } from "@angular/core";
2  import { ActivatedRouteSnapshot, CanActivate, Route
3  import { Observable } from "rxjs";
4
5  @Injectable()
6  export class AuthGuard implements CanActivate {
7    canActivate(route: ActivatedRouteSnapshot, state:
8      return
9    }
10
11 }
```

Dentro del método *canActivate* es donde crearemos el código para permitir el proceso de *login* y *logout*. Para eso vamos a necesitar la ayuda de otro

servicio, donde **simularemos un proceso de autenticación**.

4. Creamos ese *service* a la altura del *AppComponent* y lo llamamos ***auth.service.ts***. En una app de verdad, este servicio podría interactuar con un servidor para llevar a cabo el proceso de *login/logout*, pero para el caso que nos ocupa, lo vamos a simular.

5. Creamos una propiedad llamada *loggedIn* y le damos el valor de ***false***, porque es lo común cuando un usuario entra a una web, que no esté autenticado a priori.

6. Creamos un método llamado *login* que cambiará el valor de la propiedad *loggedIn* por *true*.

7. Creamos un método llamado *logout*, encargado de darle al *loggedIn* su valor inicial (*false)*.

8. Creamos un método para comprobar el estado de nuestra app, llamado *isAuthenticated*, donde simularemos que el código de su interior **tarda unos segundos en ejecutarse**, como si se conectara con un servidor. Para eso, creamos una promesa y usamos un *setTimeout*.

Si la promesa se resuelve, devolverá el valor de la propiedad *loggedIn*.

No olvidemos que una promesa siempre devuelve algo, tal y como Max enseña en su curso.

```
import { Injectable } from "@angular/core";

@Injectable()
export class AuthService {

  loggedIn = false;

  isAuthenticated() {
    const promise = new Promise(
      (resolve, reject) => {
        setTimeout(() => {
          resolve(this.loggedIn);
        }, 800);
      }
    );
    return promise;
  }

```

```
    login() {
      this.loggedIn = true;
    }

    logout() {
      this.loggedIn = false;
    }
  }
```

Con el *service* configurado, vamos a **usarlo** en el *auth-guard.service.ts*.

9. Creamos un constructor, inyectamos el *authService* y lo convertimos en una propiedad de TS.

10. En el guardián *canActivate*, comprobamos si un usuario **está autenticado o no**. Para eso, conectamos con nuestro *authService*, concretamente con el método *isAuthenticated*, que devuelve una promesa. Desde ahí, configuramos lo que sucederá si la promesa se resuelve (con el método *then)*.

Sabemos que, de resolverse, **obtendremos un *boolean*** (si el usuario está o no autenticado, o sea, *true* o *false)*. Ahí dentro comprobamos si el valor es *true*, en cuyo caso devolveremos *true*. En caso contrario, navegaremos fuera de la ruta.

⛵ Para navegar fuera de la ruta, inyectamos el *router* en el constructor y lo usamos junto con el método *navigate* para llevar al usuario a la ruta raíz, por ejemplo (la *home page*, es decir, /).

```
canActivate(route: ActivatedRouteSnapshot,
            state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
   return this.authService.isAuthenticated()
     .then(
       (authenticated: boolean) => {
         if (authenticated) {
           return true;
         } else {
           this.router.navigate(['/']);
         }
       }
     );
 }
```

Con estos cambios, el guardián aún no está listo para usar . Queda un toque final. Para ello nos vamos al *app-routing.module.ts*, desde donde **especificaremos qué rutas queremos proteger**. Esto se hace en la ruta, por ejemplo, en *servers*, añadiendo la propiedad *canActivate*.

La propiedad *canActivate* acepta un *array* donde se le especifican todos los guardianes que queremos aplicar a esta ruta, **aplicándose automáticamente a todas sus *child routes*.** Así que le indicamos nuestro guardián, el *AuthGuard*.

```
1 | { path: 'servers', canActivate:[AuthGuard], componen
```

11. Nos vamos al *app.module.ts* y añadimos en los *providers* los dos servicios que hemos creado para que Angular pueda inyectarlos.

```
1 | providers: [ServersService, AuthGuard, AuthService],
```

Tal y como está nuestro código, no deberíamos tener acceso **a ninguna ruta de los *servers***, ya que la propiedad *loggedIn* tiene el valor inicial de *false*. Puedes comprobarlo en tu navegador.

¡Genial! Aunque un poco radical, porque no podemos acceder a absolutamente nada de los *servers*, y al hacer clic somos redirigidos (después de 0.8 segundos) a la *Home page*. Quedaría mejor si pudiésemos **ver la lista de servidores**, protegiendo solo las *child routes*. Sigue leyendo para averiguar cómo hacerlo 👇.

Cómo proteger las *child routes* usando *canActivateChild*

Una opción para conseguir esto sería cortar la propiedad *canActivate* del *path "servers"* y aplicarla a las *nested routes:*

```
1 { path: 'servers', component: ServersComponent, chil
2   { path: ':id/edit', canActivate:[AuthGuard], compor
3   { path: ':id', canActivate:[AuthGuard], component:
```

Pero eso, aunque funciona, no sería muy cómodo, ya que si añadiésemos otra *child route*, tendríamos que añadir el *canActivate* también ahí **manualmente**.

Existe otro guardián de rutas que podemos usar, muy parecido al *canActivate:* ***CanActivateChild***.

1. En el *auth-guard.service.ts*, implementamos la *interface "CanActivateChild"*, que nos obliga a añadir el método *canActiveChild* dentro de la clase. Este método toma la misma forma que su hermano mayor, necesitando los mismos parámetros y siendo del mismo tipo.

2. Hacemos un *return* del método *canActivate*, pasándole la ruta y el estado como argumentos.

```
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot,
              state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return this.authService.isAuthenticated()
      .then(
        (authenticated: boolean) => {
          if(authenticated) {
            return true;
          } else {
            this.router.navigate(['/']);
          }
        }
      );
  }

  canActivateChild(route: ActivatedRouteSnapshot,
                   state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return this.canActivate(route, state);
  }
```

Con estos cambios, ahora podemos añadir **un *hook* nuevo** a nuestras rutas del *app-routing.module.ts*.

3. Comentamos la propiedad *canActivate* y añadimos la propiedad *canActivateChild*, que también espera un *array* de servicios que actúen de guardianes de rutas. Le pasamos nuestro guardián *AuthGuard*.

```
1 | path: 'servers',
```

```
2 | // canActivate: [AuthGuard],
3 | canActivateChild: [AuthGuard],
4 | component: ServersComponent,
```

¡Y listo! 🎉 Ahora ya **vemos la lista de servidores**, pero no podemos acceder a ninguno y somos redireccionados a la *Home page* al hacer clic en alguno, al igual que si en la URL intentamos acceder a alguno individual.

THE END!

¡Y hasta aquí la parte #6 de esta completa guía sobre el *Routing*! Espero que hayas aprendido algo nuevo . Si te queda alguna duda, ¡nos vemos en los comentarios! Y si quieres seguir aprendiendo, aquí tienes la parte #7 (disponible próximamente).

Sobre la autora de este post

Soy Rocío, una abogada reconvertida en programadora. Soy una apasionada de aprender cosas nuevas y ferviente defensora de que la única manera de ser feliz es alcanzando un equilibrio entre lo que te encanta hacer y lo que te saque de pobre. Mi historia completa, aquí.

Si quieres ayudar a hacer este blog sostenible, puedes invitarme a un café digital ツ

¡Gracias!

Más recursos de aprendizaje

En mi experiencia, la manera más eficaz para aprender Angular es combinando varias vías de aprendizaje. Uno de mis métodos favoritos son los **vídeo-cursos** y mi plataforma predilecta para eso es Udemy. He hecho varios cursos pero sólo recomiendo aquellos que verdaderamente me han sido útiles. Aquí van:

Max Schwarzmüller

Fernando Herrera

Si necesitas apoyo en forma de **libro**, puede que éstos te sirvan de ayuda:

La programación es un mundo que evoluciona a una velocidad de vértigo. Los autores de estos libros lo saben, por eso suelen encargarse de **actualizar** su contenido regularmente. Asegúrate de que así sea antes de adquirirlos .

Participo en el programa de afiliados de Udemy y Amazon, lo que significa que, si compras alguno de estos cursos y/o libros, yo me llevaré una pequeña comisión y a ti no costará nada extra. Vamos, lo que se dice un *win-win* .

Otros artículos que pueden interesarte

Cómo aprendí a programar cuando estaba «programada» para ser de letras

[tcb-script src="https://player.vimeo.com/api/player.js"][/tcb-script]A nadie le gusta su trabajo. Eso es lo que me decía a mí misma cuando

Guía de iniciación al data binding en Angular

¿Qué es el databinding? El databinding es la forma que tiene Angular para permitirnos mostrar contenido dinámico en lugar de estático (en inglés, hardcoded)

Claves para entender Angular – Qué es y cómo se utiliza

Angular es un framework creado por Google que nos permite construir Single Page Applications (SPA, por sus siglas en inglés).Frameworks¿

trabajo mientras estudias y sin que te cueste un riñón, he preparado un vídeo para ti.

Tu nombre

Tu email

¡MÁNDAMELO!

Si crees que este post puede serle útil a alguien, por favor, ¡compártelo!:

Frontend

Route Guards, routing

< Routing en Angular: Guía completa: Parte 5

> Routing en Angular: Guía completa: Parte 7

Deja un comentario

Conectado como JOAQUÍN DELHOM VIANA. Edita tu perfil. ¿Salir? Los campos obligatorios están marcados con *

PUBLICAR COMENTARIO

Este sitio usa Akismet para reducir el spam. Aprende cómo se procesan los datos de tus comentarios.

Entradas recientes

Recipes App – Parte 1 14 diciembre, 2022

Cómo crear contenido en Trello usando su API y Postman 24 agosto, 2022

Pilares del Software Testing 27 julio, 2022

Sesión de preguntas y respuestas #2 22 septiembre, 2021

Categorías

Backend

De Derecho a IT

Frontend

Otras pasiones

Reto Geek 365

Testing

Etiquetas

@Injectable() Angular Angular basics APIs async & await async JS Babel Bases de datos Cloud Firestore coding challenges componentes data binding dependency injection Directives DOM Firebase GitHub inversiones JavaScript JavaScript basics JavaScript intermedio JS clases local storage navegación programática ngClass ngFor ngStyle npm objetos Observables paths POO private promises proyectos Angular proyectos JS query params router-outlet routerLink routerLinkActive routing rxjs services subscribe() Webpack

Archivos

Elegir el mes

Comentarios recientes

Ro en Async JS: Guía completa sobre código asíncrono – Parte #1

Fernando Ernesto en Async JS: Guía completa sobre código asíncrono – Parte #1

Ro en Cómo aprendí a programar cuando estaba «programada» para ser de letras

Dario en Cómo aprendí a programar cuando estaba «programada» para ser de letras

Ro en Ejercicio con componentes, data binding y encapsulation