Using Guards in Angular Routing

Introduction

Route guards are important features in Angular that help control navigation from the current URL to a different page with some restrictions or checks.

Consider the following scenarios:

* The user is not authorized to navigate to the target component.
* The user must login (authenticate) first.
* You need to fetch some data before you display the target component.
* You want to save pending changes before leaving a component.
* You need to ask the user if it's OK to discard pending changes rather than save them.

We add guards to the route configuration to handle these scenarios.

A guard's return value controls the router's behavior according to the following logic:

* If it returns true, the navigation process continues.
* If it returns false, the navigation process stops and the user stays put.
* If it returns a UrlTree, the current navigation cancels and a new navigation is initiated to the UrlTree returned.

This guide will explain how to use two route guards: CanActivate and CanDeactivate.

CanActivate

Applications often restrict access to a feature area based on who the user is. You can permit access only to authenticated users or to users with a specific role. You might block or limit access until the user's account is activated.

The CanActivate guard is the tool to manage these navigation business rules.

Below is the command to generate a guard in Angular.

1

ng generate guard auth/auth

cmd

Consider an example in which you are creating admin route in a company application. At first, you add an Admin link to the AppComponent shell so that users can get to this feature.

File name: app.component.ts

1

2

3

4

5

<h1 class="title">Angular Router</h1>

<nav>

<a routerLink="/employees" routerLinkActive="active">Employees</a>

<a routerLink="/admin" routerLinkActive="active">Admin</a>

</nav>

html

Every route within the Employees section is open to everyone. The new admin feature should be accessible only to authenticated users.

Now, write a canActivate() guard method to redirect anonymous users to the login page when they try to enter the admin area. Next, open admin-routing.module.ts, import the AuthGuard class, and update the admin route with a canActivate guard property that references it.

File name: admin-routing.module.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

import { AuthGuard } from '../auth/auth.guard';

const adminRoutes: Routes = [

{

path: 'admin',

component: AdminComponent,

canActivate: [AuthGuard],

children: [

{

path: '',

children: [

{ path: 'employees', component: EmployeesComponent },

{ path: '', component: AdminDashboardComponent }

],

}

]

}

];

@NgModule({

imports: [

RouterModule.forChild(adminRoutes)

],

exports: [

RouterModule

]

})

export class AdminRoutingModule {}

typescript

The admin feature is now protected by the guard, but the guard requires more customization to work fully.

Authenticating with AuthGuard

Make the AuthGuard mimic authentication. The AuthGuard should call an application service that can login a user and retain information about the current user. Generate a new AuthService in the auth folder:

File name: auth.service.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

import { Injectable } from '@angular/core';

import { Observable, of } from 'rxjs';

import { tap, delay } from 'rxjs/operators';

@Injectable({

providedIn: 'root',

})

export class AuthService {

isLoggedIn = false;

// store the URL so we can redirect after logging in

redirectUrl: string;

login(): Observable<boolean> {

return of(true).pipe(

delay(1000),

tap(val => this.isLoggedIn = true)

);

}

logout(): void {

this.isLoggedIn = false;

}

}

typescript

Although it doesn't actually log in, it has an isLoggedIn flag to tell us whether the user is authenticated. Its login() method simulates an API call to an external service by returning an observable that resolves successfully after a short pause. The redirectUrl property stores the URL that the user wanted to access so they can navigate to it after authentication.

File name: auth.guard.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

import { Injectable } from '@angular/core';

import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from '@angular/router';

import { AuthService } from './auth.service';

@Injectable({

providedIn: 'root',

})

export class AuthGuard implements CanActivate {

constructor(private authService: AuthService, private router: Router) {}

canActivate(

next: ActivatedRouteSnapshot,

state: RouterStateSnapshot): boolean {

let url: string = state.url;

return this.checkLogin(url);

}

checkLogin(url: string): boolean {

if (this.authService.isLoggedIn) { return true; }

// Store the attempted URL for redirecting

this.authService.redirectUrl = url;

// Navigate to the login page with extras

this.router.navigate(['/login']);

return false;

}

}

typescript

Now inject the AuthService and the Router in the constructor. You haven't provided the AuthService yet, but it's good to know that you can inject helpful services into routing guards.

This guard returns a synchronous Boolean result. If the user is logged in, it returns true and the navigation continues. We need to register a /login route in the auth/auth-routing.module.ts. In app.module.ts, import and add the AuthModule to the AppModule imports.

File name: app.module.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { FormsModule } from '@angular/forms';

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { AppComponent } from './app.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

import { ComposeMessageComponent } from './compose-message/compose-message.component';

import { AppRoutingModule } from './app-routing.module';

import { EmployeesModule } from './employees/employees.module';

import { AuthModule } from './auth/auth.module';

@NgModule({

imports: [

BrowserModule,

BrowserAnimationsModule,

FormsModule,

EmployeesModule,

AuthModule,

AppRoutingModule,

],

declarations: [

AppComponent,

ComposeMessageComponent,

PageNotFoundComponent

],

bootstrap: [ AppComponent ]

})

export class AppModule {

}

typescript

CanDeactivate

In some scenarios, you might have to accumulate the user's changes, validate across fields, validate on the server, or hold changes in a pending state until the user confirms them as a group or cancels and reverts all changes.

When the user navigates away, you can let the user decide what to do with unsaved changes. If the user cancels, you'll stay put and allow more changes. If the user approves, the app can save.

You should still delay navigation until the save succeeds. If you were to let the user move to the next screen immediately and saving were to fail, you would lose the context of the error.

You need to stop the navigation while you wait, asynchronously, for the server to return with its answer.

The CanDeactivate guard helps you decide what to do with unsaved changes and how to proceed.

Cancel and Save

Both buttons navigate back to the employees list after save or cancel.

File name: employee-detail.component.ts

1

2

3

4

5

6

7

8

cancel() {

this.gotoEmployees();

}

save() {

this.employee.name = this.editName;

this.gotoEmployees();

}

typescript

In this scenario, the user could click the employees link, cancel, push the browser back button, or navigate away without saving. This example app asks the user to be explicit with a confirmation dialog box that waits asynchronously for the user's response.

File name: dialog.service.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

import { Injectable } from '@angular/core';

import { Observable, of } from 'rxjs';

@Injectable({

providedIn: 'root',

})

export class DialogService {

confirm(message?: string): Observable<boolean> {

const confirmation = window.confirm(message || 'Yes');

return of(confirmation);

};

}

typescript

It returns an observable that resolves when the user eventually decides what to do, either to discard changes and navigate away.

Generate a guard that checks for the presence of a canDeactivate() method in a component.

1

ng generate guard can-deactivate

cmd

While the guard doesn't have to know which component has a deactivate method, it can detect that the EmployeesComponent component has the canDeactivate() method and call it. The guard, not knowing the details of any component's deactivation method, is reusable.

File name: can-deactivate.guard.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

import { Injectable } from '@angular/core';

import { CanDeactivate } from '@angular/router';

import { Observable } from 'rxjs';

export interface CanComponentDeactivate {

canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;

}

@Injectable({

providedIn: 'root',

})

export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {

canDeactivate(component: CanComponentDeactivate) {

return component.canDeactivate ? component.canDeactivate() : true;

}

}

typescript

Looking back at EmployeesComponent, it implements the confirmation workflow for unsaved changes.

File name: employee-detail.component.ts

1

2

3

4

5

6

canDeactivate(): Observable<boolean> | boolean {

if (!this.employee || this.employee.name === this.editName) {

return true;

}

return this.dialogService.confirm('Discard changes?');

}

typescript

Add the guard to the crisis detail route in employee-routing.module.ts using the canDeactivate array property.

File name: employees-routing.module.ts

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

import { NgModule } from '@angular/core';

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

import { EmployeesHomeComponent } from './employees-home/employees-home.component';

import { EmployeesListComponent } from './employees-list/employees-list.component';

import { EmployeesComponent } from './employees/employees.component';

import { EmployeesDetailComponent } from './employees-detail/employees-detail.component';

import { CanDeactivateGuard } from '../can-deactivate.guard';

const crisisCenterRoutes: Routes = [

{

path: 'employees',

component: EmployeesComponent,

children: [

{

path: '',

component: EmployeesListComponent,

children: [

{

path: ':id',

component: EmployeesDetailComponent,

canDeactivate: [CanDeactivateGuard]

},

{

path: '',

component: EmployeesHomeComponent

}

]

}

]

}

];

@NgModule({

imports: [

RouterModule.forChild(employeesRoutes)

],

exports: [

RouterModule

]

})

export class EmployeesRoutingModule { }

typescript

Conclusion

In this guide, we have explored how to use guards in Angular routes. We have also learned how to activate and deactivate routes using CanActivate and CanDeactivate while navigating from one URL to another.

You can learn more about Angular in my guide [Activating Routes with RouterLink in Angular](https://app.pluralsight.com/guides/activating-routes-with-routerllink-in-angular).