Route Guards in Angular

Route guards in Angular **protect routes** by allowing or preventing navigation based on conditions like authentication, permissions, or data validity.

What are Route Guards?

Angular provides five types of route guards:

1. **CanActivate** – Checks if a route can be activated.
2. **CanActivateChild** – Checks if a child route can be activated.
3. **CanDeactivate** – Checks if a component can be exited.
4. **Resolve** – Pre-fetches data before loading the route.
5. **CanLoad** – Checks if a lazy-loaded module should be loaded.(deprecated)

CanActivate

The `canActivate` method is part of Angular's **Route Guards**, which help control access to certain routes in an Angular application. It determines whether a user is allowed to activate a route.

Syntax:

```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean
```

Parameters:

1. **`route: ActivatedRouteSnapshot`**
   - Provides access to the route parameters and data for the requested route.

2. **`state: RouterStateSnapshot`**
   - Provides the entire router state, including the URL of the attempted navigation.

Setting Up an Angular Project

Run the following commands to create a new project and generate components:

```
ng new route-guards-example -no-standalone
cd route-guards-example
ng g c home
ng g c dashboard
ng g c profile
ng g c login
ng g s auth
ng g g auth
```

- **home** → Default landing page.
- **dashboard** → Protected page (Requires login).
- **Userprofile->** **Protected**
- **login** → Page for user authentication.
- **auth** → Service to manage authentication.

Create an Authentication Service (`auth.service.ts`)

This service **simulates authentication** and stores login state.

1 `auth.service.ts` (Authentication Service)

```
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';

@Injectable({ providedIn: 'root' })
export class AuthService {
  private isLoggedIn = false;

  login() {
    this.isLoggedIn = true;
  }

  logout() {
    this.isLoggedIn = false;
    this.router.navigate(['/login']);
  }

  isAuthenticated(): boolean {
    return this.isLoggedIn;
  }

  constructor(private router: Router) {}
}
```

`auth.guard.ts` (Auth Guard)

```
import { Injectable } from '@angular/core';
import { CanActivate, Router, ActivatedRouteSnapshot, RouterStateSnapshot }
from '@angular/router';
import { AuthService } from '../auth.service';

@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
boolean {
    if (this.authService.isAuthenticated()) {
      return true;
    } else {
      alert('Access Denied! You need to log in.');
      this.router.navigate(['/login']);
      return false;
    }
  }
}
```

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { DashboardComponent } from './dashboard/dashboard.component';
import { ProfileComponent } from './profile/profile.component';
import { LoginComponent } from './login/login.component';
import { AuthGuard } from './guards/auth.guard';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'dashboard', component: DashboardComponent, canActivate:
[AuthGuard] },
  { path: 'profile', component: ProfileComponent, canActivate: [AuthGuard]
},
  { path: 'login', component: LoginComponent },
  { path: '**', redirectTo: '' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

- **Protects both `/dashboard` and `/profile` using `AuthGuard`.**

`login.component.ts` (Login Button & Message)

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { AuthService } from '../auth.service';

@Component({
```

```
  selector: 'app-login',
  template: `<h2>Login Page</h2>
             <button (click)="onLogin()">Login</button>`
})
export class LoginComponent {
  constructor(private authService: AuthService, private router: Router) {}

  onLogin() {
    this.authService.login();
    alert('You are now logged in!');
    this.router.navigate(['/dashboard']);
  }
}
```

`dashboard.component.ts` (Logout & Navigation)

```
import { Component } from '@angular/core';
import { AuthService } from '../auth.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-dashboard',
  template: `<h2>Dashboard</h2>
             <p>Welcome to your dashboard!</p>
             <button (click)="onLogout()">Logout</button>`
})
export class DashboardComponent {
  constructor(private authService: AuthService, private router: Router) {}

  onLogout() {
    this.authService.logout();
    alert('You have logged out!');
  }
}
```

`profile.component.ts` (Another Protected Page)

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-profile',
  template: `<h2>User Profile</h2>
             <p>This page is also protected by the Auth Guard.</p>`
})
export class ProfileComponent {}
```

- **Another protected page to demonstrate guard usage**.

`home.component.ts` (Navigation Bar)

```
import { Component } from '@angular/core';

@Component({
```

```
  selector: 'app-home',
  templateUrl: `.app/userprofile.html’


})
export class HomeComponent {}
```

Provides navigation links to test different routes.

app.component.html

```
<h1>Angular Auth Guard Demo</h1>
<nav>
                <a routerLink="/">Home</a> |
                <a routerLink="/dashboard">Dashboard</a> |
                <a routerLink="/profile">Profile</a> |
                <a routerLink="/login">Login</a>
              </nav>`
<router-outlet></router-outlet>
```

4.2 CanActivateChild Guard (For Child Routes)

`CanActivateChild` Guard in Angular

The **`CanActivateChild`** guard is used to protect child routes from unauthorized access. It ensures that a user must meet certain conditions before accessing child routes.

Location: `src/app/guards/child.guard.ts`

```
import { Injectable } from '@angular/core';
import { CanActivateChild, Router } from '@angular/router';
import { AuthService } from '../auth.service';

@Injectable({
  providedIn: 'root'
})
export class ChildGuard implements CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivateChild(): boolean {
    if (this.authService.isUserAdmin()) {
      return true;
    } else {
```

```
      this.router.navigate(['/dashboard']);
      return false;
    }
  }
}
```

- **Protects child routes under `dashboard`** – Only admins can access them.

4.3 CanDeactivate Guard (For Unsaved Changes)

The **`CanDeactivate`** guard is used to prevent the user from navigating away from the current route if certain conditions are met. This is typically useful when you want to warn the user about unsaved changes or confirm that they indeed want to leave the page (e.g., on a form).

For example, in a form, if the user is editing data and tries to navigate away without saving, the guard can prompt them with a confirmation message asking if they are sure about leaving the page without saving.

How **`CanDeactivate`** Works

- **`CanDeactivate`** can be added to a route when you want to prevent the user from accidentally leaving the page (or navigating away from a form).
- The guard needs to check whether there are unsaved changes or other conditions before allowing or blocking the navigation.

Step 1: Create the **`CanDeactivate`** Guard

First, let's create the **`CanDeactivate`** guard.

ng generate guard guards/can-deactivate --implements CanDeactivate

1. **Define the `CanDeactivate` guard** in `can-deactivate.guard.ts`:

```
import { CanDeactivateFn } from '@angular/router';
import { inject } from '@angular/core';
import { Observable } from 'rxjs';

export interface CanComponentDeactivate {
  canDeactivate: () =>  boolean;
}

export const canDeactivateGuard: CanDeactivateFn<CanComponentDeactivate> =
(component) => {
  return component.canDeactivate ? component.canDeactivate() : true;
};
```

Step 2: Implement `canDeactivate()` in the Component

Now, you need to implement the `canDeactivate()` method in the component you want to protect. The method should return a boolean or an observable/promise that resolves to `true` (allow navigation) or `false` (block navigation).

Example: **`edit.component.ts`** (where the user is editing a form):

```
import { Component } from '@angular/core';

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


@Component({
  selector: 'app-edit',
  template: `
    <h2>Edit Form</h2>
    <form>
      <label>Name:</label>
      <input type="text" [(ngModel)]="formData" />
      <button type="submit">Save</button>
    </form>
    <button (click)="resetForm()">Reset</button>
  `,
})
export class EditComponent implements CanComponentDeactivate {
  formData: string = '';
  isDirty: boolean = false;


  constructor() {
    // Simulating change detection
    setTimeout(() => (this.isDirty = true), 2000);
  }
```

```
  canDeactivate(): boolean {

    if (this.isDirty) {

      return confirm('You have unsaved changes. Do you really want to
leave?');

    }

    return true;

  }



  resetForm() {

    this.isDirty = false; // Reset form status after saving

  }

}
```

Step 3: Apply the `CanDeactivate` Guard to Routes

Now that the guard is set up, apply it to the route you want to protect in your **`app-routing.module.ts`**.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { EditComponent } from './edit/edit.component';
import { canDeactivateGuard } from './guards/can-deactivate.guard';  //
Import the guard

const routes: Routes = [
  {
    path: 'edit',
    component: EditComponent,
    canDeactivate: [canDeactivateGuard]  // Apply the CanDeactivate guard
here
  },

  { path: '', redirectTo: '/edit', pathMatch: 'full' },
  { path: '**', redirectTo: '/edit' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Location: `src/app/guards/unsaved-changes.guard.ts`

```
import { CanDeactivateFn } from '@angular/router';
import { inject } from '@angular/core';
import { Observable } from 'rxjs';

export interface CanComponentDeactivate {
  canDeactivate: () =>  boolean;
}

export const canDeactivateGuard: CanDeactivateFn<CanComponentDeactivate> =
(component) => {
  return component.canDeactivate ? component.canDeactivate() : true;
};
```

- **Prevents navigation if there are unsaved changes.**

4.4 Resolve Guard (Pre-Fetching Data)

A **Resolve Guard** in Angular is used to fetch data **before** navigating to a route. This ensures that the route is loaded only after the required data is available.

ng generate guard guards/user-resolver --implements Resolve

```
import { Injectable } from '@angular/core';
import { Resolve } from '@angular/router';
import { Observable, of } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class UserResolver implements Resolve<any> {
resolve(): Observable<any> //Observable: A stream of data that you can observe over time.
{
    // Simulating an API call (Replace with real API call)
    const userData = { id: 1, name: 'John Doe', email: 'john@example.com' };
    return of(userData);
  }
}
```

App.routing-module.ts

```
{
   /* path: 'edit',
    component: EditComponent,
    canDeactivate: [canDeactivateGuard]  // Apply the CanDeactivate guard here
    */
    path: 'edit',
    component: EditComponent,
    resolve: { user: UserResolver }
  },
```

editComponent.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-edit',
  template: `<h2>Edit User</h2> <p>{{ userData | json }}</p>`,
  standalone:false
})
export class EditComponent implements OnInit {
  userData: any;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
// Subscribe: A method to listen to and handle values emitted by the observable.
    this.route.data.subscribe(data => {
      this.userData = data['user']; //  Access resolved data
      console.log('Resolved User Data:', this.userData);
    });
  }
}
```

- **Fetches data before loading the route.**

4.5 CanLoad Guard (Lazy Loading Protection)

CanLoad Guard in Angular

The **`CanLoad`** guard in Angular is used to prevent a route from being loaded until certain conditions are met. Unlike **`CanActivate`** and **`CanActivateChild`** which are executed when a route is about to be activated, the **`CanLoad`** guard is used **before the route is loaded** for lazy-loaded modules.

It is particularly useful for scenarios where you want to restrict access to lazy-loaded modules based on specific conditions (e.g., authentication, permission checks) before the module is loaded. This guard is executed even before the module is fetched, so it can help reduce unnecessary loading if a user doesn't have the necessary permissions.

How Does `CanLoad` Work?

- **Use Case**: Prevent loading a module if the user does not meet certain conditions (e.g., not logged in).
- **When is it triggered?**: It's triggered before the route is loaded, ensuring that a lazy-loaded module doesn't load unless the guard conditions are satisfied.

Example Usage of `CanLoad` Guard

1. **Create the CanLoad Guard**: You need to create a guard that implements the `CanLoad` interface. The `CanLoad` guard can return either `true` (allowing the route to be loaded) or a `UrlTree` (to redirect the user to another route).

```
import { Injectable } from '@angular/core';
import { CanLoad, Route, UrlSegment, UrlTree, Router } from
'@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanLoad {

  constructor(private router: Router) {}

  canLoad(
    route: Route,
    segments: UrlSegment[]): Observable<boolean | UrlTree> |
Promise<boolean | UrlTree> | boolean | UrlTree {

    // Check if the user is authenticated
    const isAuthenticated = localStorage.getItem('isLoggedIn') === 'true';

    if (isAuthenticated) {
      return true;  // Allow loading of the module
    } else {
      // Redirect to the login page if not authenticated
      this.router.navigate(['/login']);
      return false;  // Prevent module loading
    }
  }
}
```

2. **Apply `CanLoad` to Routes**:

You can apply the `CanLoad` guard to your lazy-loaded modules in the route configuration.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AuthGuard } from './auth.guard'; // Import your guard

const routes: Routes = [
  {
    path: 'dashboard',
    loadChildren: () => import('./dashboard/dashboard.module').then(m =>
m.DashboardModule),
    canLoad: [AuthGuard]  // Apply the CanLoad guard here
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

3. **Result**:
    - When a user tries to access the **`dashboard`** route, the `CanLoad` guard will check if the user is authenticated.
    - If not authenticated, the guard will prevent the module from loading and redirect the user to the login page.

Key Points about `CanLoad`:

- **Lazy-Loaded Modules**: The `CanLoad` guard is specifically useful for **lazy-loaded modules**, helping to control access before the module is even fetched.
- **Prevention of Unnecessary Loading**: Unlike **`CanActivate`** or **`CanActivateChild`**, which check conditions after the route is loaded, **`CanLoad`** prevents the route from even being loaded, making it more efficient.
- **Returns**: The `canLoad()` method can return:
    - `true`: Allow loading the route.
    - `false`: Prevent loading the route.
    - `UrlTree`: Redirect the user to another route.

When to Use `CanLoad`?

- When you want to **restrict access to a lazy-loaded module** until certain conditions (e.g., authentication, permissions) are met, preventing unnecessary HTTP requests to load the module.
- Typically used for **authentication checks** or **feature access restrictions**.

Comparison with `CanActivate`:

- **`CanLoad`** is triggered **before** the route is loaded (for lazy-loaded modules).
- **`CanActivate`** is triggered **when** the route is activated, but after the module has been loaded.

Summary of Guards

Guard	Purpose
`CanActivate`	Prevents unauthorized users from accessing routes.
`CanActivateChild`	Prevents unauthorized users from accessing child routes.
`CanDeactivate`	Asks users for confirmation before leaving a page with unsaved changes.
`Resolve`	Pre-fetches data before loading a route.
`CanLoad`	Prevents lazy-loaded modules from being loaded if the user lacks access.

Assignment: Role-Based Navigation in an E-Commerce Admin Panel

Problem Statement:

You are developing an **Admin Panel** for an **E-Commerce platform**. The platform has three types of users:

1. **Admin** - Full access to all sections.
2. **Manager** - Access to product management and order management but not user management.
3. **Customer Support** - Access only to order management.

Your task is to **implement navigation restrictions** using **Angular Guards (`CanActivate` and `CanActivateChild`)** to ensure that users can only access the sections they are authorized for.

Requirements:

- Use **`CanActivate` Guard** to protect the main admin routes based on user roles.
- Use **`CanActivateChild` Guard** to restrict access to specific child routes under the admin panel.
- Store user roles in a **mock authentication service** (`AuthService`).

- Redirect unauthorized users to a **403 Forbidden page** if they try to access restricted pages.

Assignment : Online Learning Platform

Scenario:

You are building a **simple online learning platform** where users can navigate between different sections like **Home, Courses, and Profile** using **Angular Modules and Routing**.

Requirements:

1. **Create an Angular app** with separate modules for `Home`, `Courses`, and `Profile`.
2. **Implement routing** so users can navigate between pages.
3. **Use RouterLink** to navigate without refreshing the page.
4. **Pass a dynamic Course ID** as a route parameter.
5. **Protect the Profile page** with a simple authentication guard.

Expected Features

1. Users can **navigate** between Home, Courses, and Profile.
2. Clicking a **course** shows detailed information.
3. **Profile page is protected**, requiring a login.
4. **Lazy Loading** improves performance.
5. **404 Page** displays for unknown routes.