Les gardes canActivate, canActivateChildren

Temps de lecture : 5 minutes

Que sont les gardes ?

Les `guards` (gardes) sont des fonctions ou des services qui déterminent si une route peut être activée ou désactivée, offrant ainsi un contrôle précis sur la navigation au sein de l'application.

`Angular` propose plusieurs types de `guards` pour gérer l'accès aux routes :

- `CanActivate` : Vérifie si l'utilisateur peut accéder à une route spécifique.
- `CanActivateChild` : Vérifie si l'utilisateur peut accéder aux routes enfants d'une route spécifique.
- `CanDeactivate` : Vérifie si l'utilisateur peut quitter une route, utile pour prévenir la perte de données non sauvegardées.
- `CanLoad` : Détermine si un module peut être chargé, souvent utilisé pour le chargement paresseux (lazy loading).
- `CanMatch` : Introduit dans les versions récentes d'Angular, il remplace `CanLoad` pour déterminer si une route peut être activée en fonction de critères spécifiques.

Garde	Description	Retour attendu	Utilisation typique
`canActivate`	Empêche ou autorise la navigation vers une route.	`true`, `false`, `UrlTree`, `Observable`, ou `Promise`	Vérifier si un utilisateur a les autorisations nécessaires (par exemple, authentification, rôle requis) pour accéder à une page ou à une ressource.
`canActivateChild`	Empêche ou autorise la navigation vers une route enfant d'une route principale.	`true`, `false`, `UrlTree`, `Observable`, ou `Promise`	Appliquer des restrictions sur l'accès aux routes enfants, par exemple pour s'assurer que les utilisateurs autorisés pour une route principale le sont aussi pour ses enfants.
`canDeactivate`	Empêche ou autorise la sortie d'une route.	`true`, `false`, `Observable`, ou `Promise`	Empêcher de quitter une page sans sauvegarder des modifications ou sans confirmer une action (exemple : formulaire non sauvegardé).
`resolve`	Précharge les données avant de naviguer vers une route.	Données (tout type)	Charger des données nécessaires au composant avant qu'il ne s'affiche, comme les détails d'un utilisateur ou une liste d'articles. Ces données sont injectées via `ActivatedRoute.data`.
`canMatch`	Empêche ou autorise la navigation vers une route en fonction d'une logique personnalisée.	`true`, `false`, `Observable`, ou `Promise`	Utilisé pour configurer une logique personnalisée de correspondance de routes, par exemple en fonction d'une condition dynamique comme l'état du système ou des paramètres. Ce garde est introduit pour remplacer les fonctions de correspondance dans les définitions de routes.
`canLoad`	Empêche ou autorise le chargement d'un module via `loadChildren`.	`true`, `false`, `Observable`, ou `Promise`	Empêcher le chargement d'un module entier (lazy-loaded) si certaines conditions ne sont pas remplies, par exemple si l'utilisateur n'est pas authentifié. Cette vérification se fait avant

Garde	Description	Retour attendu	Utilisation typique
			le chargement effectif du code du module pour économiser les ressources.

Le garde `CanActivate`

Fonctionnement

Lorsqu'un utilisateur tente de naviguer vers une route protégée par `canActivate`, `Angular` exécute cette fonction avant de charger le composant associé.

Si `true` est retourné, `Angular` autorise l'accès à la route.

Si `false` ou un `UrlTree` est retourné, `Angular` empêche la navigation et peut rediriger l'utilisateur vers une autre `URL` si nécessaire.

La navigation est suspendue jusqu'à ce que toutes les gardes associées à la route soient évaluées.

Syntaxe et signature

`canActivate` est une fonction ou une méthode utilisée pour déterminer si une route peut être activée.

Elle est souvent utilisée pour vérifier si un utilisateur est authentifié ou possède les autorisations nécessaires avant d'accéder à une page.

```
export type CanActivateFn = (
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot
) => Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree;
```

- `route: ActivatedRouteSnapshot`
  - Contient des informations sur la route activée, comme les paramètres de route (`route.params`), les données associées (`route.data`), et d'autres métadonnées utiles.
  - Permet d'accéder aux détails de la configuration de la route et des informations contextuelles.
- `state: RouterStateSnapshot`
  - Représente l'état actuel du routeur au moment où la garde est évaluée.
  - Contient des informations comme l'URL actuelle et les segments de route.

Retour attendu :

`canActivate` peut retourner quatre types de valeurs :

- `boolean` : retourne `true` si l'accès est autorisé, ou `false` pour interdire l'accès.
- `UrlTree` : permet de rediriger vers une autre URL si l'accès est refusé. Vous pouvez utiliser `Router.createUrlTree()` ou `Router.parseUrl()` pour créer un `UrlTree`.
- `Observable<boolean | UrlTree>` : si la logique pour déterminer l'accès est basée sur des flux de données (comme une API), vous pouvez retourner un Observable qui émet `true`, `false`, ou un `UrlTree`.
- `Promise<boolean | UrlTree>` : si la vérification est asynchrone (comme un appel à une API avec `fetch`), vous pouvez retourner une `Promise`.

Exemple

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';

export const authGuard: CanActivateFn = (route, state) => {
  const isAuthenticated = // logique pour vérifier l'authentification
  const router = inject(Router);

  if (isAuthenticated) {
    return true;
  } else {
    return router.parseUrl('/login');
  }
};
```

Configuration de la route avec `CanActivate` :

```
import { Routes } from '@angular/router';
import { DashboardComponent } from './dashboard.component';
import { authGuard } from './auth.guard';

export const routes: Routes = [
  {
    path: 'dashboard',
    component: DashboardComponent,
    canActivate: [authGuard],
  },
  // autres routes
];
```

Exemple asynchrone

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';

export const authGuard: CanActivateFn = async (route, state) => {
  const router = inject(Router);

  // Appel à une API pour vérifier si l'utilisateur est authentifié
  const response = await fetch('https://example.com/api/is-authenticated', {
    method: 'GET',
    credentials: 'include', // Envoie les cookies
  });

  const data = await response.json();

  if (data.isAuthenticated) {
```

```
    // Autoriser la navigation
    return true;
  } else {
    // Rediriger vers une autre route en cas d'échec
    return router.parseUrl('/forbidden');
  }
};
```

Appel à une API externe : le guard effectue un appel asynchrone à une API REST pour vérifier si l'utilisateur est authentifié.

Gestion de la réponse :

- Si l'utilisateur est authentifié (`data.isAuthenticated === true`), le `guard` retourne `true`, ce qui autorise la navigation.
- Si l'utilisateur n'est pas authentifié, le guard retourne un `UrlTree` pour rediriger vers la route `/forbidden`.

Injection de `Router` : le service `Router` est injecté à l'aide de `inject()` pour permettre la redirection en cas de navigation refusée.

Et pour les routes :

```
import { Routes } from '@angular/router';
import { UsersComponent } from './components/users.component';
import { ForbiddenComponent } from './components/forbidden.component';
import { authGuard } from './guards/auth.guard';

export const routes: Routes = [
  {
    path: 'users',
    component: UsersComponent,
    canActivate: [authGuard],
  },
  {
    path: 'forbidden',
    component: ForbiddenComponent,
  },
];
```

`CanActivateChild`

Le guard `CanActivateChild` fonctionne de manière similaire à `CanActivate`, mais s'applique aux routes enfants.

Il est utile pour protéger l'accès à des sections spécifiques d'une application, comme des sous-pages nécessitant des autorisations particulières.

```
import { inject } from '@angular/core';
import { CanActivateChildFn, Router } from '@angular/router';
```

```
export const adminGuard: CanActivateChildFn = (childRoute, state) => {
  const hasAdminRights = // logique pour vérifier les droits d'administration
  const router = inject(Router);

  if (hasAdminRights) {
    return true;
  } else {
    return router.parseUrl('/access-denied');
  }
};
```

Configuration de la route avec `CanActivateChild` :

```
import { Routes } from '@angular/router';
import { AdminComponent } from './admin.component';
import { adminGuard } from './admin.guard';

export const routes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivateChild: [adminGuard],
    children: [
      {
        path: 'settings',
        component: AdminSettingsComponent,
      },
      // autres routes enfants
    ],
  },
  // autres routes
];
```

Exemple de la vidéo

Générez le garde `auth` dans `shared/guards` :

```
ng g g auth
```

Sélectionnez `canActivate`.

Et générez dans components, le composant `forbidden` :

```
ng g c forbidden
```

`src/app/shared/guards/auth.guard.ts`

```
import { inject } from '@angular/core';
import { CanActivateFn, Router } from '@angular/router';
```

```
export const authGuard: CanActivateFn = async (route, state) => {
  const { answer } = await (await fetch('https://yesno.wtf/api')).json();
  const router = inject(Router);

  return answer === 'yes' ? true : router.parseUrl('/forbidden');
};
```

src/app/app.routes.ts

```
import { Routes } from '@angular/router';
import { HomepageComponent } from './components/homepage.component';
import { NotfoundComponent } from './components/notfound.component';
import { UserComponent } from './components/user.component';
import { UsersComponent } from './components/users.component';
import { ForbiddenComponent } from './components/forbidden.component';
import { authGuard } from './shared/guards/auth.guard';

export const routes: Routes = [
  {
    path: 'users',
    canActivate: [authGuard],
    component: UsersComponent,
  },
  {
    path: 'users/:id',
    component: UserComponent,
  },
  {
    path: '',
    component: HomepageComponent,
  },
  {
    path: 'forbidden',
    component: ForbiddenComponent,
  },
  {
    path: '**',
    component: NotfoundComponent,
  },
];
```