

Route Guards

Em algumas situações, algumas rotas das nossas aplicações não podem estar totalmente visíveis para qualquer usuário, por exemplo, um usuário não pode acessar a página de dashboard se ele não estiver logado em sua conta. Isso seria um exemplo de controle para evitar que qualquer usuário acesse qualquer rota que deseje. Para evitar esse tipo de comportamento, nós podemos utilizar os **Route Guards**.

Os Route Guards são Services que são específicos para controlar o acesso à rotas na nossa aplicação. Nós temos vários tipos de Route Guards, mas vamos utilizar os principais.

Antes de vermos os principais Route Guards, para simularmos, crie uma rota principal na sua aplicação, que não precisará de nenhum teste para que possamos acessá-la e outra que terá o Route Guard que criaremos e faça uma navegação entre elas. A principal pode ser uma página de **login** e a que necessita de autenticação pode se chamar **home**. Ao criar as rotas, faça uma navegação entre as páginas. Feito isso, estamos prontos para testarmos os Route Guards

CanActivate

Um Route Guard do tipo **CanActivate** é um guard que controla se podemos em qualquer rota da nossa página web. Vamos criar um Guard que vai ver se existe um token de acesso salvo no localStorage da nossa aplicação. Se existir, nós poderemos entrar na rota, se não, voltaremos para a rota principal (login).

Para criarmos um Router Guard CanActivate, nós podemos utilizar o comando

```
ng generate guard token-test
```

ou

```
ng g g token-test
```

Quando você executar esse comando no terminal, aparecerá essa tela para você

```
➔ ng g g guards/token-test
? Which interfaces would you like to implement? (Press <space> to select, <a> to toggle all, <i> to invert select
ion, and <enter> to proceed)
) ● CanActivate
  ○ CanActivateChild
  ○ CanDeactivate
  ○ CanLoad
```

O Angular deseja saber qual tipo de Route Guard queremos criar. Nesse caso, criaremos do tipo CanActivate. Para selecionar um tipo, utilizando as teclas **↑** ou **↓** e utilizando a tecla Enter, selecione o tipo que deseja. Selecione o CanActivate, nesse caso.

Quando o Guard for criado, você deve possuir uma estrutura como essa

```
1 import { Injectable } from '@angular/core';
2 import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot, UrlTree } from '@angular/router';
3 import { Observable } from 'rxjs';
4
5 @Injectable({
6   providedIn: 'root'
7 })
8 export class TokenTestGuard implements CanActivate {
9   canActivate(
10     route: ActivatedRouteSnapshot,
11     state: RouterStateSnapshot
12   ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
13     return true;
14   }
15 }
16
17
```

Nosso Service implementa um método chamado **canActivate**. Esse método possui vários retornos, como é possível notar. Mas, basicamente, sempre será retornado ou um valor booleano ou uma **UrlTree**. Essa **UrlTree** faz a parte de redirecionamento do usuário para outra página, caso você deseje isso.


Vamos fazer o teste que devemos fazer. Vamos ver se existe dentro do **localStorage** um item chamado **access-token**, onde ficaria um possível token dentro do nosso elemento. Se não houver, redirecionaremos o usuário para a página de login novamente. Se existir, ele poderá entrar na página de login

```
1 export class TokenTestGuard implements CanActivate {
2
3   constructor(
4     private router: Router
5   ) {
6
7   }
8
9   canActivate(
10     route: ActivatedRouteSnapshot,
11     state: RouterStateSnapshot
12   ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
13
14     if (localStorage.getItem('access-token') !== null) {
15       return true
16     }
17
18     return this.router.createUrlTree(['/'])
19   }
20 }
```

Quando um Guard retorna o valor booleano **true**, significa que pode entrar na rota. Caso ele retorne **false**, significa que não pode entrar na rota. Mas apenas retornar o valor **false**, não faria nenhum redirecionamento, continuaria apenas na mesma página. Mas, se for retornado uma `UrlTree`, o usuário será redirecionado para a rota que você informou.

Para criarmos a `UrlTree`, nós utilizamos o método chamado de Router chamado **`createUrlTree`**, informando um array com a rota que deseja acessar. Nesse caso, ele será redirecionado para a rota principal da aplicação.

Agora, com nosso Guard criado, precisamos informar quais rotas irão utilizar esse guard. Para isso, no nosso arquivo de definição das rotas, vamos utilizar a propriedade **`canActivate`** das rotas. Dentro dessa propriedade, nós informamos quais Guards do tipo `CanActivate` serão utilizados



```
1  {  
2    path: 'home',  
3    component: HomeComponent,  
4    canActivate: [  
5      TokenTestGuard  
6    ]  
7  }
```

Agora, para acessar essa rota, todos os Guards do tipo `CanActivate` devem ser satisfeitos para que seja possível entrar nessa rota.

CanActivateChild

Um Guard do tipo **`CanActivateChild`** possui a mesma estrutura de um Guard do tipo `CanActivate`. Mas, ele serve para controlar a entrada em uma rota filha (sub-rota) de alguma rota da sua aplicação. Quando você utilizar o `CanActivateChild`, para todas as rotas filhas da sua rota, ele irá atuar, ou seja, qualquer sub-rota da sua rota terá a ação dos seus

CanActivateChilds sobre elas. Para criarmos um Guard do tipo CanActivateChild, basta utilizar o mesmo comando visto anteriormente e selecionar o tipo CanActivateChild. Assim, ele criará um Guard desse tipo para você e você terá uma estrutura desse tipo

```
1 import { Injectable } from '@angular/core';
2 import { ActivatedRouteSnapshot, CanActivateChild, RouterStateSnapshot, UrlTree } from '@angular/router';
3 import { Observable } from 'rxjs';
4
5 @Injectable({
6   providedIn: 'root'
7 })
8 export class TokenTestGuard implements CanActivateChild {
9
10   canActivateChild(
11     route: ActivatedRouteSnapshot,
12     state: RouterStateSnapshot
13   ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
14     return true
15   }
16 }
17
```

É exatamente a mesma estrutura vista antes, mas, nesse caso, implementamos um método chamado **canActivateChild**. Vamos utilizar a mesma lógica que vimos anteriormente para vermos se temos um access-token no localStorage.

```
1 import { Injectable } from '@angular/core';
2 import { ActivatedRouteSnapshot, CanActivateChild, Router, RouterStateSnapshot, UrlTree } from '@angular/router';
3 import { Observable } from 'rxjs';
4
5 @Injectable({
6   providedIn: 'root'
7 })
8 export class TokenTestGuard implements CanActivateChild {
9
10   constructor(
11     private router: Router
12   ) {}
13
14   canActivateChild(
15     route: ActivatedRouteSnapshot,
16     state: RouterStateSnapshot
17   ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
18
19     if (localStorage.getItem('access-token') !== null) {
20       return true
21     }
22
23     return this.router.createUrlTree(['/home'])
24   }
25 }
26
```

O que alteramos é que ele nos redirecionar para a página home, ao invés para a página de login. Agora, vamos criar duas rotas filhas e seus respectivos componentes, **Sub1** e **Sub2**. Crie suas respectivas sub-rotas dentro de home e crie uma navegação para elas no componente Home. Ao fazer isso, vamos adicionar nosso CanActivateChild na nossa rota **home**. Para isso, basta utilizar a propriedade **canActivateChild**. Ela funciona como a

propriedade **canActivate**, mas ela recebe os guards do tipo **CanActivateChild**.



```
1  {
2    path: 'home',
3    component: HomeComponent,
4    children: [
5      {
6        path: 'sub1',
7        component: Sub1Component
8      },
9      {
10       path: 'sub2',
11       component: Sub2Component
12     }
13   ],
14   canActivateChild: [
15     TokenTestGuard
16   ]
17 }
```

Agora, para que possamos entrar em qualquer sub-rota da rota **home**, os **CanActivateChild** devem ser satisfeitos. A partir disso, poderemos entrar em qualquer sub-rota. Lembrando que os **CanActivateChilds** atuam em todas as sub-rotas de uma rota.

CanDeactivate

Um Guard do tipo **CanDeactivate** é um tipo especial de guard que impede que o usuário saia de uma rota. Imagino que você já viu em algumas páginas de formulários de alguns sites que, quando você tenta sair sem salvar o que foi digitado no formulário, ele te mostra uma mensagem de confirmação perguntando se você quer sair da página sem salvar os dados. Esse seria um ótimo exemplo do uso de um guard do tipo CanDeactivate.

Para vermos isso funcionando, vamos criar um formulário simples utilizando o Reactive Forms dentro do nosso componente Home, com os campos **username** e **password**.

Com o nosso formulário pronto, vamos criar nosso guard. Vamos criar um Guard do tipo CanDeactivate chamado **FormVerification**. Ao criá-lo, você deve possuir essa estrutura básica

```
1 import { Injectable } from '@angular/core';
2 import { ActivatedRouteSnapshot, CanDeactivate, RouterStateSnapshot, UrlTree } from '@angular/router';
3 import { Observable } from 'rxjs';
4
5
6 @Injectable({
7   providedIn: 'root'
8 })
9 export class FormVerificationGuard implements CanDeactivate<unknown> {
10   canDeactivate(
11     component: unknown,
12     currentRoute: ActivatedRouteSnapshot,
13     currentState: RouterStateSnapshot,
14     nextState?: RouterStateSnapshot
15   ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
16
17     return true
18   }
19 }
```

Repare que, diferentemente dos outros Guards, o CanDeactivate possui um generic. Esse generic é utilizado para que ele identifique para qual componente ele deve fazer referência para que ele possa acessar, dentro do guard, propriedades e métodos desse componente e validar se é possível sair ou não desse elemento. Nesse caso, vamos fazer o teste com o nosso componente Home. Então, vamos chamá-lo no generic.

```

1 import { Injectable } from '@angular/core';
2 import { ActivatedRouteSnapshot, CanDeactivate, RouterStateSnapshot, UrlTree } from '@angular/router';
3 import { HomeComponent } from '../pages/home/home.component';
4 import { Observable } from 'rxjs';
5
6 @Injectable({
7   providedIn: 'root'
8 })
9 export class FormVerificationGuard implements CanDeactivate<HomeComponent> {
10   canDeactivate(
11     component: HomeComponent,
12     currentRoute: ActivatedRouteSnapshot,
13     currentState: RouterStateSnapshot,
14     nextState?: RouterStateSnapshot
15   ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
16
17     return true
18   }

```

Tanto o generic quanto o parâmetro **component** do método devem possuir o mesmo tipo. Agora, dentro do método podemos fazer uma checagem para ver se os dados do formulário não foram salvos. Se não foram salvos, mostraremos um alerta perguntando se o usuário deseja sair da página. Se ele quiser, o método retornará **true**, mas, se não, retornará **false**.

```


1 import { Injectable } from '@angular/core';
2 import { ActivatedRouteSnapshot, CanDeactivate, RouterStateSnapshot, UrlTree } from '@angular/router';
3 import { HomeComponent } from '../pages/home/home.component';
4 import { Observable } from 'rxjs';
5
6 @Injectable({
7   providedIn: 'root'
8 })
9 export class FormVerificationGuard implements CanDeactivate<HomeComponent> {
10   canDeactivate(
11     component: HomeComponent,
12     currentRoute: ActivatedRouteSnapshot,
13     currentState: RouterStateSnapshot,
14     nextState?: RouterStateSnapshot
15   ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
16
17     if (component.form.dirty) {
18       return confirm('Se você sair da página, os dados serão perdidos. Quer realmente sair da página?')
19     }
20
21     return true
22   }
23 }

```

A propriedade **dirty** de um FormGroup informa se existe algum campo com algum valor inserido. Se tiver algo inserido em alguns dos campos, ele retornará **true** e ele perguntará se o usuário realmente quer sair da página.

A linha **21** possui um retorno padrão true, pois a condição acima pode não ser verdadeira, assim, podendo não ter retorno. A linha **21** impede isso.

Com o nosso guard pronto, basta informar que a rota **home** irá utilizá-lo. Para isso, basta que, na declaração da rota, utilizemos a propriedade **canDeactivate**, informando quais guards desse tipo devem ser utilizados dentro da rota.



```
1  {  
2    path: 'home',  
3    component: HomeComponent,  
4    canDeactivate: [  
5      FormVerificationGuard  
6    ]  
7  }
```

Testando a rota, repare que, caso você digite algo e tente mudar para outra rota da aplicação, ele irá te impedir e te mostrará a mensagem que foi informada.

OBS: O CanDeactivate não impede que o usuário feche a janela. Ele impede que o usuário altere a rota da sua página.