Route Guards

BUILDING WEB APPLICATIONS USING ANGULAR

Contents

Route Guards

- Route Guards provide a way for us to guard against certain routes being triggered in certain conditions
- You may wish to ensure a user is authorised to view that route, or that you do not navigate away from a route without saving changes the user has made
- Guards are set to return true or false, respectively allowing or cancelling the navigation
- Guards can re-route navigation as well
- Most of the time, Guards will not be able to return a synchronous result and so a Promise<boolean> or Observable<boolean> are acceptable and the router waits for the resolution

Route Guards

- Guards come in several flavours:
  - CanActivate – to allow/deny navigation to a route
  - CanActivateChild – to allow/deny navigation to a child route
  - CanDeactivate – to allow/deny navigation away from a route
  - Resolve – to perform data retrieval before activating the new route
  - CanLoad – to allow/deny navigation to a module loaded asynchronously

- Multiple guards at every level are permitted
  - CanDeactive and CanActivateChild are checked first from the deepest child route to the root route
  - CanActivate from the root route to the deepest child route are checked next
  - If the module is loaded asynchronously then CanLoad is checked before loading

CanActivate

- CanActivate route guards can be used to restrict access to particular parts of your application
- To create a guard we need:
  - A class that implements the CanActivate interface (and therefore has a canActivate method that returns a Boolean)

```
@Injectable()
   export class AuthGuardService implements CanActivate {

   constructor() { }

   canActivate() {
      console.log('canActivate called');
      return true;
   }
}
```

CanActivate

- To create a guard we need:
  - A class that implements the CanActivate interface (and therefore has a canActivate method that returns a Boolean)
  - A reference to the guard class in the Routes array, for the route that we wish to run the guard against

```
{ path: 'admin', canActivate: [AuthGuardService] }
```

CanActivateChild

- CanActivate does not check when you switch from one child route to another
- CanActivateChild route guards are used to guard changes between child routes
- We create them in the same way as CanActivate guards except with the CanActivateChild interface

```
@Injectable()
   export class AuthGuardService implements CanActivateChild {

   constructor() { }

   canActivateChild() {
      console.log('canActivateChild called');
      return true;
   }
}
```

CanActivateChild

- Then add them to your Routes object as you did with CanActivate guards

```
{
   path: 'admin',
   canActivate: [AuthGuardService],
   canActivateChild: [AuthGuardService]
   children: […]
}
```

CanDeactivate

- The CanDeactive guard allows us to stop navigation while we check that navigating away from this route is permissible
- Often we will want to check with the user whether they want to save their data first - and if so, we will then need to save that data before executing the navigation.
- Similar to other route guards we need a class to act as the guard, and then we add it to the route within the Routes array.
- CanDeactivate guards often have different implementation for different components and we do not necessarily want our guard to be aware of this implementation, so we set up an interface to abstract this

CanDeactivate

- First we create our interface for a component that has a CanDeactivate guard placed on it

```
export interface CanComponentDeactive {
   canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}
```

- Then we create the guard which calls upon this abstracted canDeactivate() method when required (the guard is passed the instance of the component in question!)

```
@Injectable()
export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactive> {

   constructor() { }

   canDeactivate(component: CanComponentDeactive) {
      if (!component) { return true }
         return component.canDeactivate ? component.canDeactivate() : true;
      }
   }
}
```

CanDeactivate

- Finally we implement the guard within the component class

```
export class ManageUsersComponent implements OnInit {

   constructor() { }

   canDeactivate() {
      return timer(2000)
   }
}
```

CanDeactivate

- Not forgetting to declare the guard on the route itself

```
{
   path: ‘users',
   canDeactivate: [CanDeactivateService],
   children: […]
}
```

Resolve

- Fetching data asynchronously is a common task, one that will occasionally fail.
- A Resolve Guard will allow you to attempt to fetch the required data ahead of activating the requested navigation, stopping the attempt if the data request fails
- Set up in a very similar way to other guards, we have a guard class which will handle the data request to the service instead of the component itself

Resolve

- The Resolve Guard class should implement Resolve with the anticipated type of data returned
- The guard will then make the request to the service for the data and then handle the success/failure of that request – perhaps redirecting the user to another url should the data not return as expected

```
export class UsersResolver implements Resolve<User[]> {
        …
        resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<User[]> {
                let id = route.params['id'];
                return this.userService.getUsers().pipe(
                        map(users=>{
                                if (users) {
                                   return users;
                                 } else {
                                   this.router.navigate(['/users']);
                                   return null
                                 }
                        })
                }
        }
}
```

Resolve

- The resolve guard is then added to the route as an object which declares the name of the property where the returned data will be stored

```
resolve: {users: UsersResolver}
```

- Which is then retrieved in the component being navigated to

```
this.activatedRoute.data.subscribe((data: {users: User[]}) => {
   this.users = data.users;
})
```

Lazy Loading

- When the user visits our application for the first time they are currently asked to download our entire application. But do they need it? Will they visit every feature? Unlikely.
- Lazy loading affords us the ability to earmark certain feature modules for loading only when requested by the user.
- To make a module ‘lazy load’ we take two steps:
  - remove all references to it from its parent module (likely to be the root module)
  - Add an appropriate path route with loadChildren property to the parent modules router

```
{path: 'users', loadChildren: 'app/users/users.module#UsersModule' }
```

CanLoad

- So we can use Route Guards to deny loading a particular route, but if that route is lazily loaded we need to be able to also stop the asynchronous loading of the requested module – this is the job of the CanLoad guard
- CanLoad works hand-in-hand with CanActivate given you want them both to allow/deny a navigation request in unison

```
{
   path: 'users',
   loadChildren: 'app/users/users.module#UsersModule‵,
   canLoad: [AuthGuardService]
}
```

- With appropriate canLoad() method

```
canLoad(): boolean {
   return true;
}
```

Exercise

USING ROUTE GUARDS IN YOUR APPLICATION