Authorization

Authorization refers to the process that determines what a user is able to do. For example, an administrative user is allowed to create, edit, and delete posts. A non-administrative user is only authorized to read the posts.

Authorization is orthogonal and independent from authentication. However, authorization requires an authentication mechanism.

There are many different approaches and strategies to handle authorization. The approach taken for any project depends on its particular application requirements. This chapter presents a few approaches to authorization that can be adapted to a variety of different requirements.

Basic RBAC implementation

Role-based access control (**RBAC**) is a policy-neutral access-control mechanism defined around roles and privileges. In this section, we'll demonstrate how to implement a very basic RBAC mechanism using Nest guards.

First, let's create a `Role` enum representing roles in the system:

```
@@filename(role.enum)
export enum Role {
  User = 'user',
  Admin = 'admin',
}
```

> info **Hint** In more sophisticated systems, you may store roles within a database, or pull them from the external authentication provider.

With this in place, we can create a `@Roles()` decorator. This decorator allows specifying what roles are required to access specific resources.

```
@@filename(roles.decorator)
import { SetMetadata } from '@nestjs/common';
import { Role } from '../enums/role.enum';

export const ROLES_KEY = 'roles';
export const Roles = (...roles: Role[]) => SetMetadata(ROLES_KEY, roles);
@@switch
import { SetMetadata } from '@nestjs/common';

export const ROLES_KEY = 'roles';
export const Roles = (...roles) => SetMetadata(ROLES_KEY, roles);
```

Now that we have a custom `@Roles()` decorator, we can use it to decorate any route handler.

```
@@filename(cats.controller)
@Post()
@Roles(Role.Admin)
create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@switch
@Post()
@Roles(Role.Admin)
@Bind(Body())
create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

Finally, we create a `RolesGuard` class which will compare the roles assigned to the current user to the actual roles required by the current route being processed. In order to access the route's role(s) (custom metadata), we'll use the `Reflector` helper class, which is provided out of the box by the framework and exposed from the `@nestjs/core` package.

```
@@filename(roles.guard)
import { Injectable, CanActivate, ExecutionContext } from
'@nestjs/common';
import { Reflector } from '@nestjs/core';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<Role[]>
(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) {
      return true;
    }
    const { user } = context.switchToHttp().getRequest();
    return requiredRoles.some((role) => user.roles?.includes(role));
  }
}
@@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { Reflector } from '@nestjs/core';

@Injectable()
@Dependencies(Reflector)
export class RolesGuard {
  constructor(reflector) {
    this.reflector = reflector;
  }

  canActivate(context) {
    const requiredRoles = this.reflector.getAllAndOverride(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) {
      return true;
    }
    const { user } = context.switchToHttp().getRequest();
    return requiredRoles.some((role) => user.roles.includes(role));
  }
}
```

> info **Hint** Refer to the Reflection and metadata section of the Execution context chapter for more details on utilizing `Reflector` in a context-sensitive way.

> warning **Notice** This example is named "**basic**" as we only check for the presence of roles on the route handler level. In real-world applications, you may have endpoints/handlers that involve several operations, in which each of them requires a specific set of permissions. In this case, you'll have to provide a mechanism to check roles somewhere within your business-logic, making it somewhat harder to maintain as there will be no centralized place that associates permissions with specific actions.

In this example, we assumed that `request.user` contains the user instance and allowed roles (under the `roles` property). In your app, you will probably make that association in your custom **authentication guard** - see authentication chapter for more details.

To make sure this example works, your `User` class must look as follows:

```
class User {
  // ...other properties
  roles: Role[];
}
```

Lastly, make sure to register the `RolesGuard`, for example, at the controller level, or globally:

```
providers: [
  {
    provide: APP_GUARD,
    useClass: RolesGuard,
  },
],
```

When a user with insufficient privileges requests an endpoint, Nest automatically returns the following response:

```
{
  "statusCode": 403,
  "message": "Forbidden resource",
  "error": "Forbidden"
}
```

> info **Hint** If you want to return a different error response, you should throw your own specific exception instead of returning a boolean value.

Claims-based authorization

When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is a name-value pair that represents what the subject can do, not what the subject is.

To implement a Claims-based authorization in Nest, you can follow the same steps we have shown above in the RBAC section with one significant difference: instead of checking for specific roles, you should compare **permissions**. Every user would have a set of permissions assigned. Likewise, each resource/endpoint would define what permissions are required (for example, through a dedicated `@RequirePermissions()` decorator) to access them.

```
@@filename(cats.controller)
@Post()
@RequirePermissions(Permission.CREATE_CAT)
create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@switch
@Post()
@RequirePermissions(Permission.CREATE_CAT)
@Bind(Body())
create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

> info **Hint** In the example above, `Permission` (similar to `Role` we have shown in RBAC section) is a TypeScript enum that contains all the permissions available in your system.

Integrating CASL

CASL is an isomorphic authorization library which restricts what resources a given client is allowed to access. It's designed to be incrementally adoptable and can easily scale between a simple claim based and fully featured subject and attribute based authorization.

To start, first install the `@casl/ability` package:

```
$ npm i @casl/ability
```

> info **Hint** In this example, we chose CASL, but you can use any other library like `accesscontrol` or `acl`, depending on your preferences and project needs.

Once the installation is complete, for the sake of illustrating the mechanics of CASL, we'll define two entity classes: `User` and `Article`.

```
class User {
  id: number;
  isAdmin: boolean;
}
```

`User` class consists of two properties, `id`, which is a unique user identifier, and `isAdmin`, indicating whether a user has administrator privileges.

```
class Article {
  id: number;
  isPublished: boolean;
  authorId: number;
}
```

`Article` class has three properties, respectively `id`, `isPublished`, and `authorId`. `id` is a unique article identifier, `isPublished` indicates whether an article was already published or not, and `authorId`, which is an ID of a user who wrote the article.

Now let's review and refine our requirements for this example:

- Admins can manage (create/read/update/delete) all entities
- Users have read-only access to everything
- Users can update their articles (`article.authorId === userId`)
- Articles that are published already cannot be removed (`article.isPublished === true`)

With this in mind, we can start off by creating an `Action` enum representing all possible actions that the users can perform with entities:

```
export enum Action {
  Manage = 'manage',
  Create = 'create',
  Read = 'read',
  Update = 'update',
  Delete = 'delete',
}
```

> warning **Notice** `manage` is a special keyword in CASL which represents "any" action.

To encapsulate CASL library, let's generate the `CaslModule` and `CaslAbilityFactory` now.

```
$ nest g module casl
$ nest g class casl/casl-ability.factory
```

With this in place, we can define the `createForUser()` method on the `CaslAbilityFactory`. This method will create the `Ability` object for a given user:

```
type Subjects = InferSubjects<typeof Article | typeof User> | 'all';

export type AppAbility = Ability<[Action, Subjects]>;

@Injectable()
export class CaslAbilityFactory {
  createForUser(user: User) {
    const { can, cannot, build } = new AbilityBuilder<
      Ability<[Action, Subjects]>
    >(Ability as AbilityClass<AppAbility>);

    if (user.isAdmin) {
      can(Action.Manage, 'all'); // read-write access to everything
    } else {
      can(Action.Read, 'all'); // read-only access to everything
    }

    can(Action.Update, Article, { authorId: user.id });
    cannot(Action.Delete, Article, { isPublished: true });

    return build({
      // Read https://casl.js.org/v5/en/guide/subject-type-detection#use-
classes-as-subject-types for details
      detectSubjectType: (item) =>
        item.constructor as ExtractSubjectType<Subjects>,
    });
  }
}
```

> warning **Notice** `all` is a special keyword in CASL that represents "any subject".

> info **Hint** `Ability`, `AbilityBuilder`, `AbilityClass`, and `ExtractSubjectType` classes are exported from the `@casl/ability` package.

> info **Hint** `detectSubjectType` option let CASL understand how to get subject type out of an object. For more information read CASL documentation for details.

In the example above, we created the `Ability` instance using the `AbilityBuilder` class. As you probably guessed, `can` and `cannot` accept the same arguments but have different meanings, `can` allows to do an action on the specified subject and `cannot` forbids. Both may accept up to 4 arguments. To learn more about these functions, visit the official CASL documentation.

Lastly, make sure to add the `CaslAbilityFactory` to the `providers` and `exports` arrays in the `CaslModule` module definition:

```
import { Module } from '@nestjs/common';
import { CaslAbilityFactory } from './casl-ability.factory';

@Module({
  providers: [CaslAbilityFactory],
  exports: [CaslAbilityFactory],
})
export class CaslModule {}
```

With this in place, we can inject the `CaslAbilityFactory` to any class using standard constructor injection as long as the `CaslModule` is imported in the host context:

```
constructor(private caslAbilityFactory: CaslAbilityFactory) {}
```

Then use it in a class as follows.

```
const ability = this.caslAbilityFactory.createForUser(user);
if (ability.can(Action.Read, 'all')) {
  // "user" has read access to everything
}
```

> info **Hint** Learn more about the `Ability` class in the official CASL documentation.

For example, let's say we have a user who is not an admin. In this case, the user should be able to read articles, but creating new ones or removing the existing articles should be prohibited.

```
const user = new User();
user.isAdmin = false;

const ability = this.caslAbilityFactory.createForUser(user);
ability.can(Action.Read, Article); // true
ability.can(Action.Delete, Article); // false
ability.can(Action.Create, Article); // false
```

> info **Hint** Although both `Ability` and `AbilityBuilder` classes provide `can` and `cannot` methods, they have different purposes and accept slightly different arguments.

Also, as we have specified in our requirements, the user should be able to update its articles:

```
const user = new User();
user.id = 1;

const article = new Article();
article.authorId = user.id;

const ability = this.caslAbilityFactory.createForUser(user);
ability.can(Action.Update, article); // true

article.authorId = 2;
ability.can(Action.Update, article); // false
```

As you can see, `Ability` instance allows us to check permissions in pretty readable way. Likewise, `AbilityBuilder` allows us to define permissions (and specify various conditions) in a similar fashion. To find more examples, visit the official documentation.

Advanced: Implementing a `PoliciesGuard`

In this section, we'll demonstrate how to build a somewhat more sophisticated guard, which checks if a user meets specific **authorization policies** that can be configured on the method-level (you can extend it to respect policies configured on the class-level too). In this example, we are going to use the CASL package just for illustration purposes, but using this library is not required. Also, we will use the `CaslAbilityFactory` provider that we've created in the previous section.

First, let's flesh out the requirements. The goal is to provide a mechanism that allows specifying policy checks per route handler. We will support both objects and functions (for simpler checks and for those who prefer more functional-style code).

Let's start off by defining interfaces for policy handlers:

```
import { AppAbility } from '../casl/casl-ability.factory';

interface IPolicyHandler {
  handle(ability: AppAbility): boolean;
}

type PolicyHandlerCallback = (ability: AppAbility) => boolean;

export type PolicyHandler = IPolicyHandler | PolicyHandlerCallback;
```

As mentioned above, we provided two possible ways of defining a policy handler, an object (instance of a class that implements the `IPolicyHandler` interface) and a function (which meets the `PolicyHandlerCallback` type).

With this in place, we can create a `@CheckPolicies()` decorator. This decorator allows specifying what policies have to be met to access specific resources.

```
export const CHECK_POLICIES_KEY = 'check_policy';
export const CheckPolicies = (...handlers: PolicyHandler[]) =>
  SetMetadata(CHECK_POLICIES_KEY, handlers);
```

Now let's create a `PoliciesGuard` that will extract and execute all the policy handlers bound to a route handler.

```
@Injectable()
export class PoliciesGuard implements CanActivate {
  constructor(
    private reflector: Reflector,
    private caslAbilityFactory: CaslAbilityFactory,
  ) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const policyHandlers =
      this.reflector.get<PolicyHandler[]>(
        CHECK_POLICIES_KEY,
        context.getHandler(),
      ) || [];

    const { user } = context.switchToHttp().getRequest();
    const ability = this.caslAbilityFactory.createForUser(user);

    return policyHandlers.every((handler) =>
      this.execPolicyHandler(handler, ability),
    );
  }

  private execPolicyHandler(handler: PolicyHandler, ability: AppAbility) {
    if (typeof handler === 'function') {
      return handler(ability);
    }
    return handler.handle(ability);
  }
}
```

> info **Hint** In this example, we assumed that `request.user` contains the user instance. In your app, you will probably make that association in your custom **authentication guard** - see authentication chapter for more details.

Let's break this example down. The `policyHandlers` is an array of handlers assigned to the method through the `@CheckPolicies()` decorator. Next, we use the `CaslAbilityFactory#create` method which constructs the `Ability` object, allowing us to verify whether a user has sufficient permissions to perform specific actions. We are passing this object to the policy handler which is either a function or an instance of a class that implements the `IPolicyHandler`, exposing the `handle()` method that returns a boolean. Lastly, we use the `Array#every` method to make sure that every handler returned `true` value.

Finally, to test this guard, bind it to any route handler, and register an inline policy handler (functional approach), as follows:

```
@Get()
@UseGuards(PoliciesGuard)
@CheckPolicies((ability: AppAbility) => ability.can(Action.Read, Article))
findAll() {
  return this.articlesService.findAll();
}
```

Alternatively, we can define a class which implements the `IPolicyHandler` interface:

```
export class ReadArticlePolicyHandler implements IPolicyHandler {
  handle(ability: AppAbility) {
    return ability.can(Action.Read, Article);
  }
}
```

And use it as follows:

```
@Get()
@UseGuards(PoliciesGuard)
@CheckPolicies(new ReadArticlePolicyHandler())
findAll() {
  return this.articlesService.findAll();
}
```

> warning **Notice** Since we must instantiate the policy handler in-place using the `new` keyword, `ReadArticlePolicyHandler` class cannot use the Dependency Injection. This can be addressed with the `ModuleRef#get` method (read more here). Basically, instead of registering functions and instances through the `@CheckPolicies()` decorator, you must allow passing a `Type<IPolicyHandler>`. Then, inside your guard, you could retrieve an instance using a type reference: `moduleRef.get(YOUR_HANDLER_TYPE)` or even dynamically instantiate it using the `ModuleRef#create` method.