Pregunta:

How can I add role-based authorization using the permissions in the JWT payload?

Respuesta:

Of course! That's an excellent next step for your application. Implementing role-based authorization is crucial for securing endpoints, and NestJS provides a very elegant way to do this using Guards and Custom Decorators.

Based on your existing setup, your JWT payload already contains the user's permissions (permisos), which is perfect. Here’s how you can leverage that to protect your routes.

### The Strategy

1. Create a Custom Decorator (@HasPermissions): This decorator will let you specify which permissions are required to access a specific route (e.g., @HasPermissions('create:user')).
2. Create a Permissions Guard (PermissionsGuard): This guard will intercept incoming requests, check for the permissions specified by the decorator, and compare them against the permissions present in the user's JWT payload.
3. Apply the Guard and Decorator: You'll apply both the JwtAuthGuard and your new PermissionsGuard to the routes you want to protect.

Let's go through the implementation step-by-step.

### 1. Improve Type Safety for the Request User (Recommended)

First, let's create a specific type for the user object that your JwtStrategy attaches to the request. This improves type safety and makes your code easier to understand.

I'll add a UserPayload interface to your [auth.model.ts](code-assist-path:/home/ramon/code/cuadrantes2/cuadrantes2-0-api/src/auth/auth.model.ts) file.

auth.model.ts

-0+6

username: string;

permisos: Permiso[];

}

+

+export interface UserPayload {

+ userId: number;

+ username: string;

+ permisos: string[];

+}

Now, let's update your JwtStrategy to use this new type. This makes it clear what shape the request.user object will have.

jwt.strategy.ts

-7+3

import { Injectable } from '@nestjs/common';

import { ConfigService } from '@nestjs/config';

-import { AuthModel } from '../auth.model';

+import { AuthModel, UserPayload } from '../auth.model';

@Injectable()

export class JwtStrategy extends PassportStrategy(Strategy) {

Unchanged lines

}

*// Passport verifica el token y si es válido, llama a este método con el payload decodificado*

- validate(payload: AuthModel) {

+ validate(payload: AuthModel): UserPayload {

*// Lo que retornemos aquí se adjuntará a request.user*

- return {

- userId: payload.sub,

- username: payload.username,

- permisos: payload.permisos,

- };

+ return { userId: payload.sub, username: payload.username, permisos: payload.permisos };

}

}

### 2. Create the Permissions Decorator

Create a new folder src/auth/decorators and add the following file. This decorator will attach the required permissions as metadata to the route handler.

New file: permissions.decorator.ts

+4

+import { SetMetadata } from '@nestjs/common';

+

+export const PERMISSIONS\_KEY = 'permissions';

+export const HasPermissions = (...permissions: string[]) => SetMetadata(PERMISSIONS\_KEY, permissions);

### 3. Create the Permissions Guard

Next, create a src/auth/guards folder and add your new PermissionsGuard. This guard will contain the core authorization logic.

New file: permissions.guard.ts

+35

+import {

+ Injectable,

+ CanActivate,

+ ExecutionContext,

+ ForbiddenException,

+} from '@nestjs/common';

+import { Reflector } from '@nestjs/core';

+import { UserPayload } from '../auth.model';

+import { PERMISSIONS\_KEY } from '../decorators/permissions.decorator';

+

+@Injectable()

+export class PermissionsGuard implements CanActivate {

+ constructor(private reflector: Reflector) {}

+

+ canActivate(context: ExecutionContext): boolean {

const requiredPermissions = this.reflector.getAllAndOverride<string[]>(

+ PERMISSIONS\_KEY,

+ [context.getHandler(), context.getClass()],

+ );

+

+ if (!requiredPermissions) {

+ return true;

+ }

+

+ const { user }: { user: UserPayload } = context.switchToHttp().getRequest();

+

+ const hasAllPermissions = requiredPermissions.every((p) => user.permisos?.includes(p));

+

+ if (user?.permisos && hasAllPermissions) {

+ return true;

+ }

+

+ throw new ForbiddenException('No tienes los permisos necesarios para acceder a este recurso.');

+ }

+}

How it works:

* It uses the Reflector to extract the permissions metadata you set with the @HasPermissions() decorator.
* If no permissions are required for a route, it allows access.
* It gets the user object from the request (which was attached by JwtStrategy).
* It checks if the user.permisos array includes all of the requiredPermissions.
* If the user has the necessary permissions, it returns true. Otherwise, it throws a ForbiddenException.

### 4. Apply the Guard to Your Routes

Finally, you can use your new decorator and guard in your controllers. The order in @UseGuards is important: AuthGuard('jwt') must come first to ensure request.user is populated before the PermissionsGuard runs.

Here is an example of how to update your AuthController:

auth.controller.ts

-2+12

import { Controller, Post, UseGuards, Request, Get } from '@nestjs/common';

import { AuthGuard } from '@nestjs/passport';

import { AuthService } from './auth.service';

-import { LoginModel } from './auth.model';

+import { LoginModel, UserPayload } from './auth.model';

import { PermissionsGuard } from './guards/permissions.guard';

import { HasPermissions } from './decorators/permissions.decorator';

@Controller('auth')

export class AuthController {

Unchanged lines

*// Ejemplo de ruta protegida*

@UseGuards(AuthGuard('jwt'))

@Get('profile')

- getProfile(@Request() req: { user: LoginModel }) {

+ getProfile(@Request() req: { user: UserPayload }) {

*// req.user es poblado por JwtStrategy con el payload del token*

return req.user;

}

+

+ *// Ejemplo de ruta protegida con permisos*

+ @UseGuards(AuthGuard('jwt'), PermissionsGuard)

+ @HasPermissions('admin')

+ @Get('admin-area')

+ getAdminArea(@Request() req: { user: UserPayload }) {

+ return { message: `Bienvenido al área de administración, ${req.user.username}!` };

+ }

+}