**NestJs Authentification and authorization**

1. **Introduction**

Authentication and authorization are crucial aspects of building secure applications. In NestJS, authentication refers to verifying the identity of users, while authorization involves granting or denying access to resources or functionalities based on the authenticated user's permissions.

Here's an overview of authentication and authorization in NestJS:

**Authentication:**

**Une image contenant texte, capture d’écran, Police, logo

Description générée automatiquement**

1. **Authentication Strategies:**
   * NestJS supports various authentication strategies, such as JWT (JSON Web Tokens), OAuth, Passport, and more.
   * These strategies can be implemented using middleware, guards, or interceptors to authenticate incoming requests.
2. **Passport Module:**
   * NestJS integrates well with the popular **passport** module for implementing authentication strategies.
   * Passport strategies can be used to authenticate requests against various providers like JWT, OAuth, local username/password, etc.
3. **AuthGuard:**
   * NestJS provides the **AuthGuard** class that can be used to protect routes based on authentication status.
   * It can be extended to implement custom authentication logic.

### Authorization:

1. **Role-Based Access Control (RBAC):**
   * Authorization in NestJS often involves role-based access control, where users are assigned roles with specific permissions.
   * Guards and interceptors can be used to implement RBAC by checking a user's role or permissions before granting access to resources.
2. **Authorization Guards:**
   * NestJS provides **Guards** to implement authorization logic. These guards can be used to check permissions or roles before allowing access to routes or resources.
   * Custom guards can be created to implement specific authorization requirements.
3. **Use of Decorators:**
   * Decorators like **@Roles()** or **@Permissions()** can be created to define roles or permissions on specific endpoints or controllers.
4. This example uses a JWT authentication strategy with a guard (**JwtAuthGuard**) to protect the **profile** endpoint. The **JwtStrategy** class handles JWT validation, and the **JwtAuthGuard** ensures that only authenticated users can access the **profile** route.
5. Overall, NestJS provides a robust framework for implementing authentication and authorization using a variety of strategies, guards, interceptors, and middleware, allowing developers to build secure applications with ease.
6. This example uses a JWT authentication strategy with a guard (**JwtAuthGuard**) to protect the **profile** endpoint. The **JwtStrategy** class handles JWT validation, and the **JwtAuthGuard** ensures that only authenticated users can access the **profile** route.
7. Overall, NestJS provides a robust framework for implementing authentication and authorization using a variety of strategies, guards, interceptors, and middleware, allowing developers to build secure applications with ease.

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

1. **Creating a users resource**

**nest g resource users**

**Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement**

**Une image contenant texte, Police, logiciel, Logiciel multimédia

Description générée automatiquement**

1. **Hashing password**

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

#### Hashing[#](https://docs.nestjs.com/security/encryption-and-hashing#hashing)

For hashing, we recommend using either the **[bcrypt](https://www.npmjs.com/package/bcrypt" \t "_blank)** or [**argon2**](https://www.npmjs.com/package/argon2) packages. Nest itself does not provide any additional wrappers on top of these modules to avoid introducing unnecessary abstractions (making the learning curve short).

As an example, let's use bcrypt to hash a random password.

First install required packages:

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

1. **Implement Sign-in and Sign-up routes.**

**Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Authentication.service.ts**

import {  
 ConflictException,  
 Injectable,  
 UnauthorizedException,  
} from '@nestjs/common';  
import { InjectRepository } from '@nestjs/typeorm';  
import { User } from '../../users/entities/user.entity';  
import { Repository } from 'typeorm';  
import { HashingService } from '../hashing/hashing.service';  
import { SignUpDto } from './dto/sign-up.dto/sign-up.dto';  
import { SignInDto } from './dto/sign-in.dto/sign-in.dto';  
  
@Injectable()  
export class AuthentificationService {  
 constructor(  
 @InjectRepository(User) private readonly usersRepository: Repository<User>,  
 private readonly hashingService: HashingService,  
 ) {}  
  
 async signUp(signUpDto: SignUpDto) {  
 try {  
 const user = new User();  
 user.email = signUpDto.email;  
 user.password = await this.hashingService.hash(signUpDto.password);  
  
 await this.usersRepository.save(user);  
 } catch (err) {  
 const pgUniqueViolationErrorCode = '23505';  
 if (err.code === pgUniqueViolationErrorCode) {  
 throw new ConflictException();  
 }  
 throw err;  
 }  
 }  
  
 async signIn(signInDto: SignInDto) {  
 const user = await this.usersRepository.findOneBy({  
 email: signInDto.email,  
 });  
 if (!user) {  
 throw new UnauthorizedException('User does not exists');  
 }  
 const isEqual = await this.hashingService.compare(  
 signInDto.password,  
 user.password,  
 );  
 if (!isEqual) {  
 throw new UnauthorizedException('Password does not match');  
 }  
  
 // *TODO :We'll fill this gap in the next lesson* }  
}

**Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, Logiciel multimédia, logiciel, Police

Description générée automatiquement**

**Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, Logiciel multimédia, logiciel, Logiciel de graphisme

Description générée automatiquement**

1. **What’s JWT**

**Une image contenant capture d’écran, texte, Police, Graphique

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

JWT stands for JSON Web Token. It is a compact and self-contained way to transmit information between parties as a JSON object. This information can be securely passed between systems as a URL parameter, in an HTTP header, or within a request body. JWTs are commonly used for authentication and information exchange in web applications.

JWTs consist of three parts separated by dots: the header, the payload, and the signature. Each part is encoded using Base64Url encoding, and when concatenated together, they form a string in the format **header.payload.signature**.

* Header: Contains information about the type of token and the signing algorithm used.
* Payload: Often referred to as the claims or assertions. It contains the data being transmitted, such as user information or metadata. The payload includes predefined claims like issuer, subject, expiration time, etc.
* Signature: Created by encoding the header, payload, and a secret key using the specified algorithm. It is used to verify that the message hasn't been tampered with and comes from a trusted source.

JWTs are stateless, meaning the server doesn't need to store sessions for authenticated users. They can be easily verified since the signature is generated using a secret key known only to the server.

Overall, JWTs are widely used in authentication mechanisms for securely transmitting information between parties, and their flexibility and ease of use make them popular in various web-based applications and APIs.

****

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, logiciel, Logiciel multimédia, Logiciel de graphisme

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Logiciel multimédia, logiciel

Description générée automatiquement**

1. **Protecting our routes with a guard**

We can now address our final requirement: protecting endpoints by requiring a valid JWT be present on the request. We'll do this by creating an AuthGuard that we can use to protect our routes.

**In this example we will protect all our routes by default**

**Access-token.guard.ts**

import {  
 CanActivate,  
 ExecutionContext,  
 Inject,  
 Injectable,  
 UnauthorizedException,  
} from '@nestjs/common';  
import { JwtService } from '@nestjs/jwt';  
import *jwtConfig* from '../../../config/jwt.config';  
import { ConfigType } from '@nestjs/config';  
import { *REQUEST\_USER\_KEY* } from '../../../iam.constants';  
import { Request } from 'express';  
  
@Injectable()  
export class AccessTokenGuard implements CanActivate {  
 constructor(  
 private jwtService: JwtService,  
 @Inject(*jwtConfig*.KEY)  
 private readonly jwtConfiguration: ConfigType<typeof *jwtConfig*>,  
 ) {}  
  
 async canActivate(context: ExecutionContext): Promise<boolean> {  
 const request = context.switchToHttp().getRequest();  
 const token = this.extractTokenFromHeader(request);  
 if (!token) {  
 throw new UnauthorizedException();  
 }  
 try {  
 const payload = await this.jwtService.verifyAsync(  
 token,  
 this.jwtConfiguration,  
 );  
 // 💡 We're assigning the payload to the request object here  
 // so that we can access it in our route handlers  
 request[*REQUEST\_USER\_KEY*] = payload;  
 *console*.log('Payload', payload);  
 } catch {  
 throw new UnauthorizedException();  
 }  
 return true;  
 }  
  
 private extractTokenFromHeader(request: Request): string | undefined {  
 const [type, token] = request.headers.authorization?.split(' ') ?? [];  
 return type === 'Bearer' ? token : undefined;  
 }  
}

**Une image contenant texte, capture d’écran, Police, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

1. **Adding Public routes**

With this in place, Nest will automatically bind AuthGuard to all endpoints.

Now we must provide a mechanism for declaring routes as public. For this, we can create a custom decorator using the SetMetadata decorator factory function.

**Une image contenant capture d’écran, texte, Logiciel multimédia, logiciel

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

In the file above, we exported two constants. One being our metadata key named IS\_PUBLIC\_KEY, and the other being our new decorator itself that we’re going to call Public (you can alternatively name it SkipAuth or AllowAnon, whatever fits your project).

Now that we have a custom @Public() decorator, we can use it to decorate any method, as follows:

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, logiciel, Logiciel multimédia, Logiciel de graphisme

Description générée automatiquement**

**Une image contenant logiciel, Logiciel multimédia, capture d’écran, texte

Description générée automatiquement**

1. **Active User decorator**

**After**

@Get()  
findAll(@Req() request) {  
 *console*.log('request', request.user);  
 return this.coffeesService.findAll();  
}

**Une image contenant capture d’écran, Logiciel de graphisme, Logiciel multimédia

Description générée automatiquement**

**Before**

**We wiil create a decorator that helps us to fetch active users.**

**In the example provided earlier, it might be applied to a controller method where the @ActiveUser() decorator is used as a parameter to retrieve the active user object within that specific route or endpoint.**

**Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement**

**Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

Description générée automatiquement**