**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**

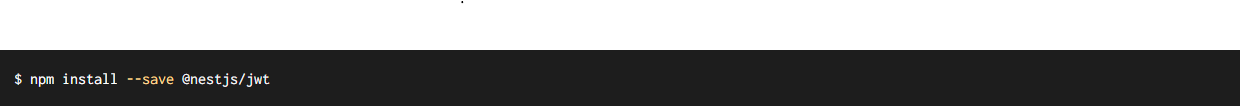
JWT stands for JSON Web Token. It is a compact and self-contained way to transmit information between parties as a JSON object. This information can be securely passed between systems as a URL parameter, in an HTTP header, or within a request body. JWTs are commonly used for authentication and information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

1. **Implementing Refresh Tokens**

Refresh tokens and access tokens are both part of an authentication and authorization mechanism used in security protocols like OAuth 2.0. They serve different purposes within the context of maintaining a user's session securely.

**Access Token:**

* An access token is a credential used to access protected resources or perform actions on behalf of a user. It is usually short-lived and has a limited lifespan.
* Typically, an access token is obtained after a user successfully authenticates (login) and authorizes their access to a specific resource or service.
* Access tokens carry specific permissions and are sent with each request to the server to access protected resources. They grant temporary access to the user's data or functionalities.
* Access tokens are more susceptible to interception or unauthorized use due to their short lifespan.

**Refresh Token:**

* A refresh token is a long-lived credential that is used to obtain a new access token after the current access token expires.
* Refresh tokens are more secure and are typically not sent with every request. Instead, they are exchanged for a new access token when the current access token becomes invalid or expires.
* Refresh tokens are usually used to maintain a user's session without requiring frequent logins. They help in reducing the frequency of the user's need to re-authenticate.
* Refresh tokens should be stored securely, and their usage should be controlled to prevent misuse or unauthorized access.

**Differences:**

1. **Lifespan:** Access tokens are short-lived and expire after a relatively short period, while refresh tokens have a longer lifespan and can be used to obtain new access tokens multiple times.
2. **Usage:** Access tokens are used to access protected resources and are sent with requests to the server. Refresh tokens are exchanged for new access tokens when needed but are not sent with each request.
3. **Security:** Refresh tokens are generally more secure as they are long-lived and should be stored securely. Access tokens are more susceptible to interception but have a shorter lifetime.
4. **Frequency of Use:** Access tokens are used frequently with each request to access resources. Refresh tokens are used less often, primarily to obtain new access tokens.

In summary, access tokens grant temporary access to resources and have a short lifespan, while refresh tokens are used to obtain new access tokens when needed and have a longer lifespan to maintain the user's session securely without frequent re-authentication. Both tokens play crucial roles in ensuring secure user sessions and access control in authentication systems.

**Example:**

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

Description générée automatiquement**

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

Description générée automatiquement**

**Authentication.service.ts**

import {  
 ConflictException,  
 Inject,  
 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';  
import { JwtService } from '@nestjs/jwt';  
import *jwtConfig* from '../config/jwt.config';  
import { ConfigType } from '@nestjs/config';  
import { ActiveUserDataInterface } from '../interfaces/active-user-data.interface';  
import { RefreshTokenDto } from './dto/refresh-token.dto';  
  
@Injectable()  
export class AuthentificationService {  
 constructor(  
 @InjectRepository(User) private readonly usersRepository: Repository<User>,  
 private readonly hashingService: HashingService,  
 private readonly jwtService: JwtService,  
 @Inject(*jwtConfig*.KEY)  
 private readonly jwtConfiguration: ConfigType<typeof *jwtConfig*>,  
 ) {}  
  
 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');  
 }  
 return await this.generateTokens(user);  
 }  
  
 async generateTokens(user: User) {  
 const [accessToken, refreshToken] = await *Promise*.all([  
 this.signToken<Partial<ActiveUserDataInterface>>(  
 user.id,  
 this.jwtConfiguration.accessTokenTtl,  
 { email: user.email },  
 ),  
 this.signToken(user.id, this.jwtConfiguration.accessTokenTtl),  
 ]);  
  
 return {  
 accessToken,  
 refreshToken,  
 };  
 }  
  
 async refreshTokens(refreshTokenDto: RefreshTokenDto) {  
 try {  
 const { sub } = await this.jwtService.verifyAsync<  
 Pick<ActiveUserDataInterface, 'sub'>  
 >(refreshTokenDto.refreshToken, {  
 secret: this.jwtConfiguration.secret,  
 audience: this.jwtConfiguration.audience,  
 issuer: this.jwtConfiguration.issuer,  
 });  
  
 const user = await this.usersRepository.findOneBy({  
 id: sub,  
 });  
 if (!user) {  
 throw new UnauthorizedException('User does not exists');  
 }  
 return this.generateTokens(user);  
 } catch (err) {  
 throw new UnauthorizedException();  
 }  
 }  
  
 private async signToken<T>(userId: number, expiresIn: number, payload?: T) {  
 return await this.jwtService.signAsync(  
 {  
 sub: userId,  
 ...payload,  
 },  
 {  
 audience: [this.jwtConfiguration.audience],  
 issuer: this.jwtConfiguration.issuer,  
 secret: this.jwtConfiguration.secret,  
 expiresIn,  
 },  
 );  
 }  
}

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

Description générée automatiquement**

1. **Invalidating Token**

Here's a general approach to invalidating tokens using refresh token rotation in a system:

1. **Token Revocation List (Blacklist)**: Maintain a list of revoked or invalidated refresh tokens. When a token is deemed compromised, expired, or should no longer be used (e.g., upon user logout, detected suspicious activity, or password change), add its identifier to a blacklist.
2. **Database or Cache Management**: Store token identifiers (or metadata) in a secure database or cache to keep track of tokens that should not be accepted for authentication or token refreshing.
3. **Token Verification Middleware/Interceptor**: Implement middleware or interceptors in your authentication logic that checks the incoming refresh token against the blacklist before allowing token refresh or access.
4. **Token Expiration**: While refresh tokens typically have a longer lifespan, establish policies to set expiration times, after which the token becomes invalid and requires re-authentication.
5. **Rotate Refresh Tokens**: As part of the rotation process, when issuing new refresh tokens, ensure that the old token identifiers are added to the blacklist to prevent their further usage.

**In our Example we will use Rotate refresh tokens and we will use Redis to store our token in a database (it IS NOT RECOMMENDED TO STORE OUR TOKENS IN A DATABASE FOR SECURITY ISSUE)**

**Our Example:**

****

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

Description générée automatiquement**

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

Description générée automatiquement**

import {  
 Injectable,  
 OnApplicationBootstrap,  
 OnApplicationShutdown,  
} from '@nestjs/common';  
import Redis from 'ioredis';  
  
export class InvalidatedRefreshTokenError extends Error {}  
  
@Injectable()  
export class RefreshTokenIdsStorage  
 implements OnApplicationBootstrap, OnApplicationShutdown  
{  
 private redisClient: Redis;  
  
 onApplicationBootstrap() {  
 // *TODO: ideally ,we should move this to the dedicated "RedisModule"* // Insted of initiating the connection here  
 this.redisClient = new Redis({  
 host: 'localhost',  
 port: 6379,  
 });  
 }  
  
 onApplicationShutdown() {  
 return this.redisClient.quit();  
 }  
  
 async insert(userId: number, tokenId: string): Promise<void> {  
 await this.redisClient.set(this.getKey(userId), tokenId);  
 }  
  
 async validate(userId: number, tokenId: string): Promise<boolean> {  
 const storedId = await this.redisClient.get(this.getKey(userId));  
 if (storedId !== tokenId) {  
 throw new InvalidatedRefreshTokenError();  
 }  
 return storedId === tokenId;  
 }  
  
 async invalidate(userId: number): Promise<void> {  
 await this.redisClient.del(this.getKey(userId));  
 }  
  
 private getKey(userId: number): string {  
 return `user-${userId}`;  
 }  
}

**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 capture d’écran, texte, logiciel, Logiciel multimédia

Description générée automatiquement**

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

Description générée automatiquement**

1. **Role-Based Acceess control**

[**https://docs.nestjs.com/security/authorization#authorization**](https://docs.nestjs.com/security/authorization#authorization)

<https://docs.nestjs.com/security/authorization#basic-rbac-implementation>

**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.

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

Description générée automatiquement**

In NestJS, role-based access control (RBAC) can be implemented to restrict access to certain endpoints or resources based on the roles assigned to users. Here's a step-by-step guide to implementing RBAC in NestJS:

1. **Define Roles**: Define roles that represent different levels of access in your application. For example, roles like 'admin', 'user', 'moderator', etc.
2. **Authentication**: Implement an authentication system (using JWT, sessions, OAuth, etc.) to authenticate users and assign roles upon successful authentication. You can use middleware or guards to handle authentication.
3. **Create a Roles Guard**: Create a custom guard that checks whether the user has the required role(s) to access a specific route. This guard will be responsible for checking the user's role against the roles required for the endpoint.
4. **Apply Guard to Endpoints**: Apply the custom roles guard to the relevant endpoints or controllers where you want to enforce role-based access control.

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

Description générée automatiquement**

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

Description générée automatiquement**

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

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

Description générée automatiquement**

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

Description générée automatiquement**

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

Description générée automatiquement**

1. **Claims based Authorization.**

**In claims-based authorization instead of using a set up of roles can be assigned to users we define multiple permission and then have the ability to grand those permissions to individual users**

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**](https://docs.nestjs.com/security/authorization#basic-rbac-implementation) 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.

Claims-based authorization is an approach to access control and authorization in software systems where permissions or access rights are granted based on specific claims or attributes associated with a user. In this context, a "claim" refers to a piece of information about the user, such as their roles, permissions, identity, or any other relevant attribute.

Instead of solely relying on a user's identity (like username or ID), claims-based authorization considers additional information stored in security tokens, often in the form of JSON Web Tokens (JWT) or Security Assertion Markup Language (SAML) assertions. These tokens contain claims that describe the user's identity, roles, group memberships, or any other relevant information.

The authorization process involves evaluating these claims against predefined policies or rules to determine whether the user is allowed to perform a particular action or access specific resources within an application or system. The claims provide a finer-grained control over access rights, allowing for more flexible and dynamic authorization decisions based on various attributes associated with the user.

Claims-based authorization offers a flexible and scalable approach to access control by decoupling the authorization logic from the application code. It allows for easier management of user permissions and facilitates interoperability between different systems by providing a standardized way of conveying authorization information.

**Example:**

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

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, 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, Système d’exploitation

Description générée automatiquement**

**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, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

1. **Policy based Authorization.**

#### Advanced: Implementing a PoliciesGuard[#](https://docs.nestjs.com/security/authorization" \l "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.

In NestJS, policy-based authorization can be implemented using Guards. Guards are used to control access to different parts of your application based on certain conditions. Policy-based authorization involves creating and applying policies that define whether a user is allowed to perform a specific action or access a particular resource.

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

Description générée automatiquement**

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

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Système d’exploitation

Description générée automatiquement**

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

Description générée automatiquement**

****

import {  
 CanActivate,  
 ExecutionContext,  
 ForbiddenException,  
 Injectable,  
 Type,  
} from '@nestjs/common';  
import { Reflector } from '@nestjs/core';  
import { Policy } from '../policies/interfaces/policy.interface';  
import { *POLICIES\_KEY* } from '../decorators/policies.decorator';  
import { *REQUEST\_USER\_KEY* } from '../../iam.constants';  
import { PolicyHandlersStorage } from '../policies/policy-handlers.storage';  
import { ActiveUserDataInterface } from '../../interfaces/active-user-data.interface';  
  
@Injectable()  
export class PoliciesGuard implements CanActivate {  
 constructor(  
 private readonly reflector: Reflector,  
 private readonly policyHandlerStorage: PolicyHandlersStorage,  
 ) {}  
 async canActivate(context: ExecutionContext): Promise<boolean> {  
 const policies = this.reflector.getAllAndOverride<Policy[]>(*POLICIES\_KEY*, [  
 context.getHandler(),  
 context.getClass(),  
 ]);  
  
 if (policies) {  
 const user: ActiveUserDataInterface = context.switchToHttp().getRequest()[  
 *REQUEST\_USER\_KEY* ];  
 await *Promise*.all(  
 policies.map((policy) => {  
 const policyHandler = this.policyHandlerStorage.get(  
 policy.constructor as Type,  
 );  
 return policyHandler.handle(policy, user);  
 }),  
 ).catch((err) => {  
 throw new ForbiddenException(err.messages);  
 });  
 }  
 return true;  
 }  
}

**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, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

1. **Introduction to API keys**

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

Description générée automatiquement**

API keys are strings of characters, often alphanumeric, that are used to authenticate and control access to an API (Application Programming Interface). They serve as a form of security mechanism, allowing developers and services to access specific functionalities or resources provided by an API.

Here's an introduction to API keys covering their purpose, usage, and best practices:

### Purpose of API Keys:

1. **Access Control**: API keys act as a form of authentication, enabling the API provider to control who can access their APIs and what level of access they have.
2. **Security**: They provide a simple method for authenticating requests and help in preventing unauthorized access to APIs.

### Usage of API Keys:

1. **Authentication**: Developers include the API key in their requests (usually in the header or query parameter) to authenticate themselves with the API server.
2. **Authorization**: Sometimes, API keys are also used to authorize access to specific endpoints or features within the API.

### Best Practices for API Keys:

1. **Keep Them Secret**: Treat API keys like passwords. Never expose them in public repositories or share them in places accessible to unauthorized individuals.
2. **Use HTTPS**: Always transmit API keys over HTTPS to encrypt the communication between the client and server, preventing interception.
3. **Implement Rate Limiting**: Set limits on API key usage to prevent abuse or overuse, protecting the API server from being overwhelmed.
4. **Rotate Keys Regularly**: Periodically change API keys to mitigate the risk of unauthorized access due to key compromise.
5. **Restrict Access**: Apply access controls on API keys, limiting their scope to only what's necessary for the application.
6. **Monitor Usage**: Keep track of API key usage to detect any abnormal behavior or potential security threats.
7. **Consider Multiple Keys**: Provide different keys for different environments (development, staging, production) or for different functionalities to improve control and security.

### Examples of API Key Usage:

* **Google Maps API**: Developers need an API key to access the mapping services provided by Google Maps.
* **Stripe API**: When integrating Stripe's payment services into an application, an API key is used for authentication and authorization.

API keys are a fundamental aspect of securing and controlling access to APIs. However, they should be used in conjunction with other security measures (like OAuth tokens, JWTs, etc.) for stronger security and access control mechanisms based on the specific requirements of an application.

1. **Integrate API Keys feature.**

**A faire**

1. **PART 3: Google Authentication – NestJS portion.**

**A faire**

1. **Bonus: Two-factor authentification.**

**A faire**

1. **Sessions with passport**

**A faire**