Securing a REST API with JWT (JSON Web Tokens) in NestJS involves using Passport, Guards, and the **@nestjs/jwt** and **@nestjs/passport** modules. This tutorial will guide you through setting up authentication in a NestJS application using JWTs.

Setup Your NestJS Project

Create a new NestJS project:

```
nest new project-name
```

Navigate into your project directory:

```
cd project-name
```

Install Required Packages

You'll need to install **@nestjs/jwt** for JWT handling, **@nestjs/passport** for integration with Passport, and **passport** & **passport-jwt** for the JWT strategy:

```
npm install @nestjs/jwt @nestjs/passport passport passport-jwt
npm install @types/passport-jwt --save-dev
```

Create the Auth Module

Generate an **auth** module, service, and controller:

```
nest g module auth
nest g service auth
nest g controller auth
```

Setting Up JWT Module

Configure the **JwtModule** in your **auth.module.ts**: In NestJS, **JwtModule.register({})** is used to configure and register the JWT (JSON Web Tokens) module from **@nestjs/jwt** package within your application. This method allows you to set up essential options for JWT token handling, including token signing, verification, and management. When you call **JwtModule.register({})** and pass in an options object, you're configuring how the JWT service within your application will behave. In this example the signing part will be done in the **auth.service.ts.**

By configuring the **JwtModule** with **register({})**, you ensure that the JWT functionality in your NestJS application aligns with your authentication strategy's requirements, allowing for the secure generation, signing, and verification of JWTs for client authentication.

```
import { Module } from '@nestjs/common';
import { AuthController } from './auth.controller';
import { AuthService } from './auth.service';
import { ConfigModule } from "@nestjs/config";
import { JwtModule } from '@nestjs/jwt';
import { JwtStrategy } from './strategy';

@Module({
  imports: [JwtModule.register({}), ConfigModule],
    controllers: [AuthController],
    providers: [AuthService, JwtStrategy]
})
export class AuthModule {}
```

Implement the JWT Strategy

Create a **jwt.strategy.ts** file in your **auth** directory:

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy, 'jwt'){
    constructor(private config: ConfigService, private prisma: PrismaService){
        super({
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
            secretOrKey: config.get('JWT_SECRET'),
        })
    }

  async validate(payload: {sub: number, email: string}) {
        const user = await this.prisma.user.findUnique({
            where:{
                id: payload.sub,
            }
        })
        delete user.password;

        return user;
    }
}
```

This strategy extracts the JWT token from the authorization header and validates it using the secret key. The **validate** method checks the user in the Prisma database and then returns the user's data based on the token's payload, which can be injected into request handlers.

Implementing Login in AuthService

In your **auth.service.ts**, implement a method to validate users and to login and another one to sign the token with the secret key and setting the expiring time:

```
async login(dto: AuthDto){
    // find user by email
    const user = await this.prisma.user.findFirst({
        where: {
            email: dto.email,
        }
    })

    //throw exception if user does not exist
    if (!user) throw new ForbiddenException('Credentials incorrect: email')

    //compare passwords
    const pwMatches = user.password === dto.password

    //throw exception if password not correct
    if (!pwMatches) throw new ForbiddenException('Credentials incorrect: password')

    return {
        user_name: user.name,
        access_token: await this.signToken(user.id, user.email)
    }
}
```

```
async signToken(userId: number, email: string): Promise<string>{
    const payload = {
        sub: userId,
        email: email
    }

    const secret = this.config.get('JWT_SECRET')

    const token = await this.jwt.signAsync(payload, {
        expiresIn: '30m',
        secret: secret
    })

    return token;
}
```

Add Login Route in AuthController

Modify your **auth.controller.ts** to include a login route:

```
import { Body, Controller, HttpCode, HttpStatus, Post } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthDto } from './dto';

@Controller('auth')
export class AuthController {
    constructor(private authService: AuthService){};

    @HttpCode(HttpStatus.OK)
    @Post('login')
    login(@Body() dto: AuthDto){
        return this.authService.login(dto)
    }
}
```

Protect Routes with AuthGuard

To protect routes with JWT authentication, use **@UseGuards(AuthGuard('jwt'))** on your controller routes. In NestJS, **AuthGuard** is part of the **@nestjs/passport** module, integrating

NestJS with Passport, a popular middleware for Node.js authentication. **AuthGuard** is used to protect routes by ensuring that requests to those routes are authenticated. When you apply an **AuthGuard** to a route (using **@UseGuards()**), NestJS invokes the corresponding Passport strategy to authenticate the request. If authentication succeeds, the request is processed; otherwise, an error is returned.

```
@UseGuards(JwtGuard)
@Controller('cars')
export class CarController {
```

```
export class JwtGuard extends AuthGuard('jwt'){
    constructor(){
        super();
    }
}
```

This @UseGuards(JwtGuard) will guard every API from the CarController