Pontifícia Universidade Católica de Minas Gerais

Curso: Arquitetura de Software Distribuído
Disciplina: Plataformas Node.js
Professor: Samuel Martins
Valor: 15pts

Exercício 3

Introdução

Neste exercício iremos implementar a autenticação no módulo de usuários, juntamente com o um sistema de cache.

Informações sobre a entrega

A entrega deste exercício deve ser composta por:

- Código completo, seguindo o passo a passo do enunciado;
- Prints conforme orientado no último passo;

Passo 1 – Estrutura da aplicação

Neste primeiro passo iremos instalar alguns pacotes adicionais. Para começar, abra o terminal e rode o seguinte comando para instalarmos algumas dependências que serão utilizadas posteriormente:

npm install --save bcrypt @types/bcrypt @nestjs/jwt @nestjs/cache-manager@2 cache-manager@5 cache-manager-redis-store@2 redis

Estrutura do código - infrastructure

Vamos criar alguns módulos da aplicação e iniciar a construção da nossa estrutura.

Execute os seguintes comandos, um por um, na seguinte ordem:

1. nest g module infrastructure/redis
2. nest g module infrastructure/auth

Aqui estamos criando os módulos que irão receber algumas das nossas implementações necessárias para o funcionamento da aplicação. Crie um serviço dentro da pasta auth através do seguinte comando:

- nest g service infrastructure/auth/auth –flat

Crie um arquivo com o nome constants.ts que irá conter um valor que será usado posteriormente para a implementação da autenticação com JWT:

src/infrastructure/auth/constants.ts

```
export const jwtConstants = {
  secret: 'S3CR3T!',
};
```

Vamos implementar um caso de uso para usuários para que possamos recuperar um usuário por email. Implemente o seguinte método dentro do arquivo **src/infrastructure/database/repositories/users.repository.service.ts**

```
  findByEmail(email: string): Promise<IUser> {
    return this.findOneBy({ email });
  }
```

Caso de uso – recuperar usuários por email

Agora, vamos criar um caso de uso para esse cenário. Execute o comando **nest g service domain/use-cases/users/get-user-by-email –flat** e coloque o seguinte conteúdo dentro do arquivo criado

src/domain/use-cases/users/get-user-by-email.service.ts

```
import { Injectable } from '@nestjs/common';
import { IUser } from 'src/domain/interfaces/user.interface';
import { UsersRepositoryService } from
'src/infrastructure/database/repositories/users.repository.service';

@Injectable()
export class GetUserByEmailService {
  constructor(private readonly usersRepository: UsersRepositoryService) {}

  async execute(email: string): Promise<IUser> {
    const user = await this.usersRepository.findByEmail(email);

    if (!user) {
      throw new Error('Usuário não encontrado');
    }

    return user;
  }
}
```

Altere o users.module para exportar o novo caso de uso:

src/domain/use-cases/users/users.module.ts

```
import { Module } from '@nestjs/common';
import { CreateUserService } from './create-user.service';
import { GetUserByIdService } from './get-user-by-id.service';
import { DatabaseModule } from
'src/infrastructure/database/database.module';
import { GetUserByEmailService } from './get-user-by-email.service';

@Module({
  imports: [DatabaseModule],
  providers: [CreateUserService, GetUserByIdService, GetUserByEmailService],
  exports: [CreateUserService, GetUserByIdService, GetUserByEmailService],
})
export class UsersModule {}
```

Serviço de autenticação

Continuando com a implementação do processo de autenticação, vamos voltar para o arquivo **auth.service.ts** e implementar a lógica para recuperar um usuário da base de dados e comparar a senha recebida via parâmetro:

```
import { Injectable, UnauthorizedException } from '@nestjs/common';

import { compare } from 'bcrypt';
import { JwtService } from '@nestjs/jwt';
import { GetUserByEmailService } from 'src/domain/use-cases/users/get-user-
by-email.service';

@Injectable()
export class AuthService {
  constructor(
    private readonly getUserByEmailUserCase: GetUserByEmailService,
    private readonly jwtService: JwtService,
  ) {}

  async login(email: string, password: string) {
    const user = await this.getUserByEmailUserCase.execute(email);
    const isAValidUser = await compare(password, user.password);

    if (!isAValidUser) {
      throw new UnauthorizedException();
    }
    const payload = { sub: user.id, username: user.email };

    return {
      access_token: await this.jwtService.signAsync(payload),
    };
  }
}
```

No arquivo **auth.module.ts**, vamos configurar o JWT globalmente na aplicação:

```
import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { jwtConstants } from './constants';
import { UsersModule } from 'src/domain/use-cases/users/users.module';
import { AuthService } from './auth.service';

@Module({
  imports: [
    UsersModule,
```

```
    JwtModule.register({
      global: true,
      secret: jwtConstants.secret,
      signOptions: { expiresIn: '60m' },
    }),
  ],
  providers: [AuthService],
  exports: [AuthService],
})
export class AuthModule {}
```

Vamos agora criar um Guard. Esse conceito é responsável por processar as requisições e tomar a decisão de entrar no controller ou interromper o processo da requisição, baseada na regra que quisermos. Neste caso, iremos verificar se o token recebido no cabeçalho da requisição é um token válido e em seguida, deixar o usuário entrar no controller caso esteja autenticado ou simplesmente interromper a requisição e retornar um Unauthorized Exception. Execute o seguinte comando:

- **nest g service gateways/guards/auth-guard –flat**

Dentro do arquivo criado, coloque o seguinte conteúdo:

src/gateways/guards/auth-guard.service.ts

```
import {
  CanActivate,
  ExecutionContext,
  Injectable,
  UnauthorizedException,
} from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { JwtService } from '@nestjs/jwt';
import { SetMetadata } from '@nestjs/common';
import { Request } from 'express';
import { jwtConstants } from 'src/infrastructure/auth/constants';

export const IS_PUBLIC_KEY = 'isPublic';
export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);

@Injectable()
export class AuthGuardService implements CanActivate {
  constructor(
    private jwtService: JwtService,
    private reflector: Reflector,
  ) {}
```

```
  async canActivate(context: ExecutionContext): Promise<boolean> {
    const isPublic =
this.reflector.getAllAndOverride<boolean>(IS_PUBLIC_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);

    if (isPublic) {
      return true;
    }

    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);

    if (!token) {
      throw new UnauthorizedException();
    }

    try {
      const payload = await this.jwtService.verifyAsync(token, {
        secret: jwtConstants.secret,
      });
      request['user'] = 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;
  }
}
```

Exporte o AuthGuardService no módulo GatewaysModule:

src/gateways/gateways.module.ts

```
import { Module } from '@nestjs/common';
import { ControllersModule } from './controllers/controllers.module';
import { AuthGuardService } from './guards/auth-guard.service';

@Module({
  imports: [ControllersModule],
  providers: [AuthGuardService],
  exports: [AuthGuardService],
})
export class GatewaysModule {}
```

Com o guard criado e com o serviço de autenticação implementado, precisamos registrar o Guard globalmente na aplicação. Para isso, altere o arquivo app.module.ts para o seguinte conteúdo:

src/app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { InfrastructureModule } from
'./infrastructure/infrastructure.module';
import { DomainModule } from './domain/domain.module';
import { GatewaysModule } from './gateways/gateways.module';
import { APP_GUARD } from '@nestjs/core';
import { AuthGuardService } from './gateways/guards/auth-guard.service';

@Module({
  imports: [InfrastructureModule, DomainModule, GatewaysModule],
  controllers: [AppController],
  providers: [
    AppService,
    {
      provide: APP_GUARD,
      useClass: AuthGuardService,
    },
  ],
})
export class AppModule {}
```

Iremos agora criar um controller para login. Execute o seguinte comando:

- **nest g controller gateways/controllers/auth/auth –flat**

Antes de implementar o controller, crie uma pasta dto dentro da pasta auth do controller, e implemente o seguinte arquivo:

src/gateways/controllers/auth/dtos/login.dto.ts

```
import { IsNotEmpty, IsString } from 'class-validator';

export class LoginDto {
  @IsNotEmpty()
  @IsString()
  email: string;

  @IsString()
  @IsNotEmpty()
  password: string;
}
```

Vamos implementar o controller responsável por fazer o login da aplicação:

src/gateways/controllers/auth/auth.controller.ts

```
import { Body, Controller, HttpCode, HttpStatus, Post } from
'@nestjs/common';
import { AuthService } from 'src/infrastructure/auth/auth.service';
import { Public } from 'src/gateways/guards/auth-guard.service';
import { LoginDto } from './dtos/login.dto';

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

  @HttpCode(HttpStatus.OK)
  @Post('login')
  @Public()
  login(@Body() loginDto: LoginDto) {
    return this.authService.login(loginDto.email, loginDto.password);
  }
}
```

Criptografia de senha

Para garantir uma camada extra de segurança na nossa aplicação, vamos implementar o uso do bcrypt para criptografar a senha que será armazenada no banco. Altere o caso de uso do create-user para o seguinte código:

src/domain/use-cases/users/create-user.service.ts

```
import { Injectable } from '@nestjs/common';
import { BaseUseCase } from '../base-use-case';
import { UsersRepositoryService } from
'src/infrastructure/database/repositories/users.repository.service';
import { CreateUserDto } from 'src/gateways/controllers/users/dtos/create-
user.dto';
import { IUser } from 'src/domain/interfaces/user.interface';
import { hash } from 'bcrypt';

@Injectable()
export class CreateUserService implements BaseUseCase {
  private readonly DEFAULT_SALT_ROUNDS = 10;

  constructor(private readonly usersRepository: UsersRepositoryService) {}

  async execute(user: CreateUserDto): Promise<IUser> {
    const hashedPassword = await hash(user.password,
this.DEFAULT_SALT_ROUNDS);
    const createdUser = await this.usersRepository.add({
      ...user,
      password: hashedPassword,
    });

    if (!createdUser) {
      throw new Error('Usuário não pôde ser criado');
    }

    return createdUser;
  }
}
```

Autenticação nos controladores

O endpoint de criação de usuário precisa ser público, pois precisamos conseguir criar usuários a qualquer momento. Adicione o decorator @Public no método create() do users.controller.ts

src/gateways/controllers/users/users.controller.ts

```
  @Post()
  @Public()
  async create(@Body() createUserDto: CreateUserDto) {
```

Agora que temos a possibilidade de autenticar usuários, vamos modificar os nossos controllers para trabalharem com o usuário logado. Altere o módulo de controllers para importar o módulo de autenticação:

```
import { Module } from '@nestjs/common';
import { ProjectsController } from './projects/projects.controller';
import { TasksController } from './tasks/tasks.controller';
import { UsersController } from './users/users.controller';
import { UseCasesModule } from 'src/domain/use-cases/use-cases.module';
import { AuthController } from './auth/auth.controller';
import { AuthModule } from 'src/infrastructure/auth/auth.module';

@Module({
  imports: [UseCasesModule, AuthModule],
  controllers: [
    ProjectsController,
    TasksController,
    UsersController,
    AuthController,
  ],
})
export class ControllersModule {}
```

Altere agora os controllers de projects e tasks, respectivamente, para os seguintes conteúdos:

src/gateways/controllers/projects/projects.controller.ts

```
import {
  Body,
  Controller,
  Get,
  Inject,
  NotFoundException,
  Param,
  Post,
  Req,
  UnprocessableEntityException,
} from '@nestjs/common';
import { CreateProjectService } from 'src/domain/use-cases/projects/create-
project.service';
```

```typescript
import { GetAllProjectsService } from 'src/domain/use-cases/projects/get-all-projects.service';
import { GetProjectByIdService } from 'src/domain/use-cases/projects/get-project-by-id.service';
import { CreateProjectDto } from './dtos/create-project.dto';

@Controller('projects')
export class ProjectsController {
  constructor(
    private readonly getAllProjectsUseCase: GetAllProjectsService,
    private readonly getProjectByIdUseCase: GetProjectByIdService,
    private readonly createProjectUseCase: CreateProjectService,
  ) {}

  @Get()
  async findAll(@Req() request) {
    try {
      const loggedUser = request.user;
      return await this.getAllProjectsUseCase.execute(loggedUser.sub);
    } catch (error) {
      throw new NotFoundException(error.message);
    }
  }

  @Get(':id')
  async findOne(@Req() request, @Param('id') id: number) {
    try {
      const loggedUser = request.user;

      return await this.getProjectByIdUseCase.execute({
        userId: loggedUser.sub,
        projectId: id,
      });
    } catch (error) {
      throw new NotFoundException(error.message);
    }
  }

  @Post()
  async create(@Req() request, @Body() createProjectDto: CreateProjectDto) {
    try {
      const loggedUser = request.user;

      return await this.createProjectUseCase.execute({
        userId: loggedUser.sub,
        project: createProjectDto,
```

```
      });
    } catch (error) {
      throw new UnprocessableEntityException(error.message);
    }
  }
}
```

Para o passo anterior, lembre-se de adicionar o userId no repositório de projetos, conforme orientado pelo debug do terminal e visto em sala de aula.

O userId também precisará de ser inserido no caso de uso do create-task após a alteração no projects.repository.

src/gateways/controllers/tasks/tasks.controller.ts

```
import {
  Body,
  Controller,
  Get,
  NotFoundException,
  Param,
  Post,
  Req,
  UnprocessableEntityException,
} from '@nestjs/common';
import { CreateTaskService } from 'src/domain/use-cases/tasks/create-
task.service';
import { GetAllTasksService } from 'src/domain/use-cases/tasks/get-all-
tasks.service';
import { GetTaskByIdService } from 'src/domain/use-cases/tasks/get-task-by-
id.service';
import { CreateTaskDto } from './dtos/create-task.dto';

@Controller('tasks')
export class TasksController {
  constructor(
    private readonly getAllTasksUseCase: GetAllTasksService,
    private readonly getTaskByIdUseCase: GetTaskByIdService,
    private readonly createTaskUseCase: CreateTaskService,
  ) {}

  @Get()
  async findAll(@Req() request) {
    try {
      const loggedUser = request.user;
```

```
      return await this.getAllTasksUseCase.execute({ userId: loggedUser.sub
});
    } catch (error) {
      throw new NotFoundException(error.message);
    }
  }

  @Get(':id')
  async findOne(@Req() request, @Param('id') id: number) {
    try {
      const loggedUser = request.user;

      return await this.getTaskByIdUseCase.execute({
        userId: loggedUser.sub,
        taskId: id,
      });
    } catch (error) {
      throw new NotFoundException(error.message);
    }
  }

  @Post()
  async create(@Req() request, @Body() createTaskDto: CreateTaskDto) {
    try {
      const loggedUser = request.user;

      return await this.createTaskUseCase.execute({
        userId: loggedUser.sub,
        task: createTaskDto,
      });
    } catch (error) {
      throw new UnprocessableEntityException(error.message);
    }
  }
}
```

Perceba que em cada um dos métodos agora estamos usando a variável loggedUser que recupera um valor do objeto request, que contém os dados do usuário logado. Esses dados passaram pelo Guard, que verificou a existência do token e retornou os dados necessários daquela sessão, utilizando a estrutura JWT.

Passo 2 – Estratégia de Cache

No arquivo redis.module.ts, adicione o seguinte conteúdo

src/infrastructure/redis/redis.module.ts

```
import { CacheModule } from '@nestjs/cache-manager';
import { Module } from '@nestjs/common';

@Module({
  imports: [
    CacheModule.register({
      isGlobal: true,
      ttl: 10000,
    }),
  ],
})
export class RedisModule {}
```

Essa implementação irá servir como base para a implementação do cache utilizando o Redis posteriormente.

Altere agora o método findAll do controller de projetos para o seguinte conteúdo:

src/gateways/controllers/projects/projects.controller.ts

```
  async findAll(@Req() request) {
    try {
      const loggedUser = request.user;

      const cachedData = await this.cacheService.get<{ name: string }>(
        `user-${loggedUser.sub}/all-projects`,
      );

      console.log(cachedData);

      if (cachedData) {
        console.log(`Getting data from cache!`);
        return cachedData;
      }

      const data = await this.getAllProjectsUseCase.execute(loggedUser.sub);

      await this.cacheService.set(`user-${loggedUser.sub}/all-projects`,
data);

      return data;
```

```
    } catch (error) {
      throw new NotFoundException(error.message);
    }
  }
```

O construtor também deverá ser alterado para injetar o cache manager:

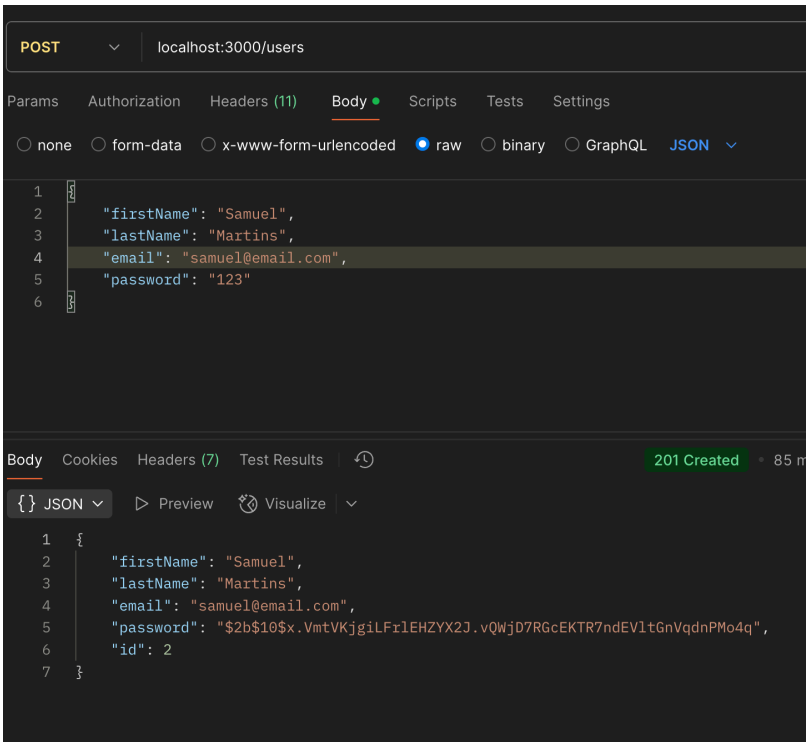
```
  constructor(
    private readonly getAllProjectsUseCase: GetAllProjectsService,
    private readonly getProjectByIdUseCase: GetProjectByIdService,
    private readonly createProjectUseCase: CreateProjectService,
    @Inject(CACHE_MANAGER) private cacheService: Cache,
  ) {}
```

O TTL (Time-to-live) está configurado para 10 segundos. Portanto, após duas requisições no endpoint /projects, devemos ver a seguinte mensagem no terminal:

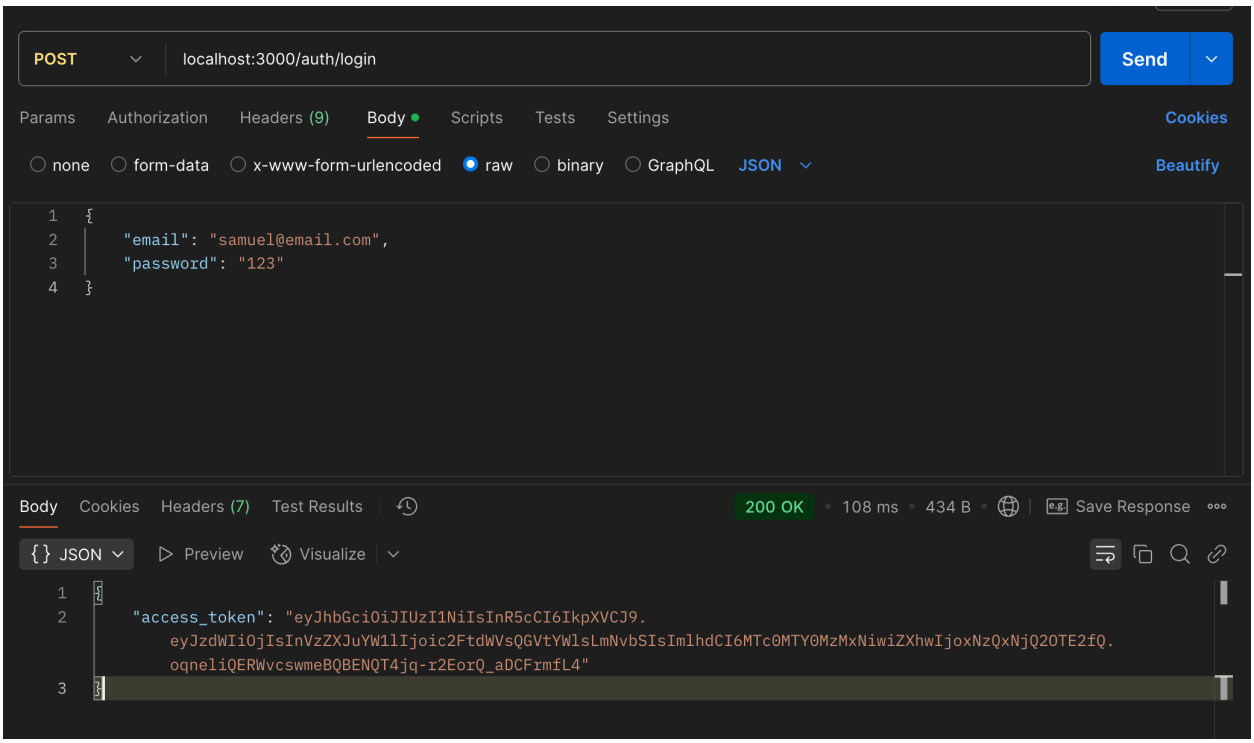
Passo 3 – Testes via postman

Para garantir que tudo funciona corretamente, tire os seguintes prints colocando os **SEUS** dados.

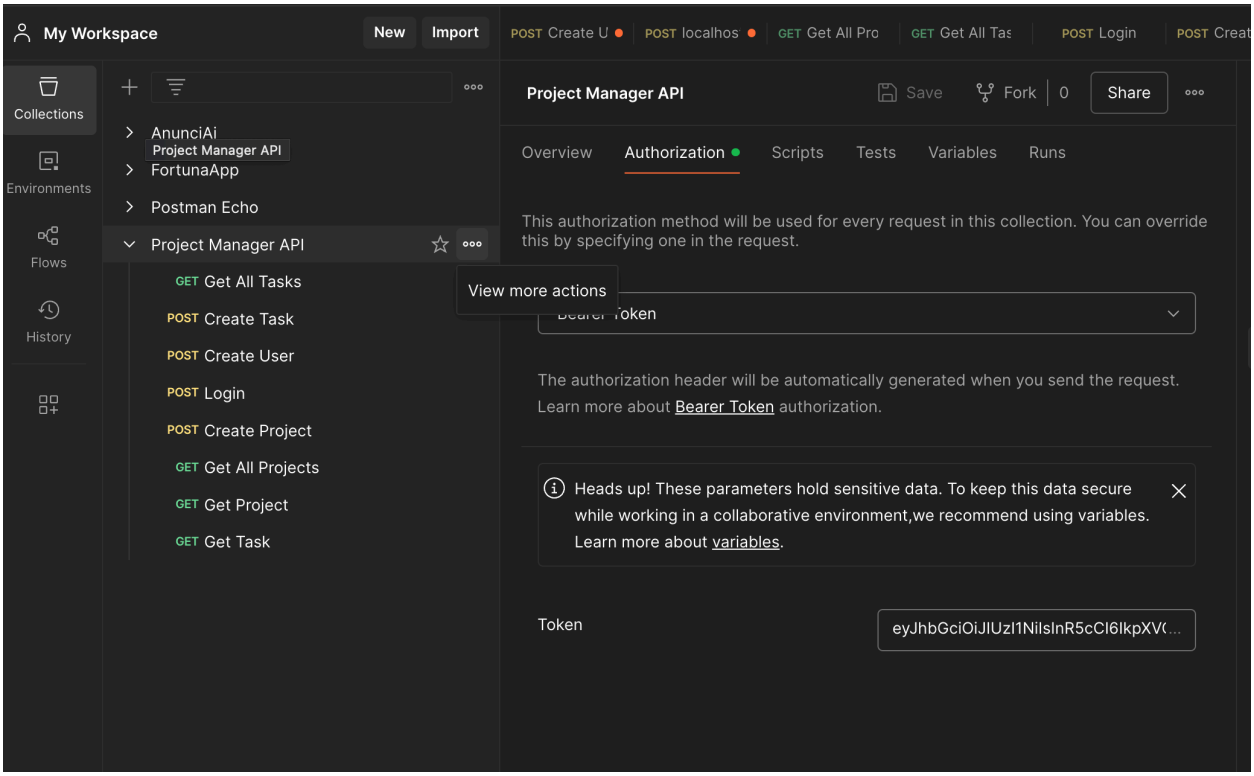
Criando um usuário com senha



Login usuário



Copie o conteúdo do token criado e na collection enviada via Canvas, clique em editar e cole o valor no campo Token:



Projeto criado com usuário autenticado