Pontifícia Universidade Católica de Minas Gerais

Curso: Arquitetura de Software Distribuído
Disciplina: Plataformas Node.js
Professor: Samuel Martins
Valor: 20pts

Exercício 4

Introdução

Neste exercício iremos converter a nossa aplicação para uma arquitetura baseada em microserviços, utilizando o padrão Pub/Sub do Redis. Iremos extrair o módulo de Tasks da nossa base de código e colocá-lo em um pacote separado, onde este irá rodar separadamente da aplicação.

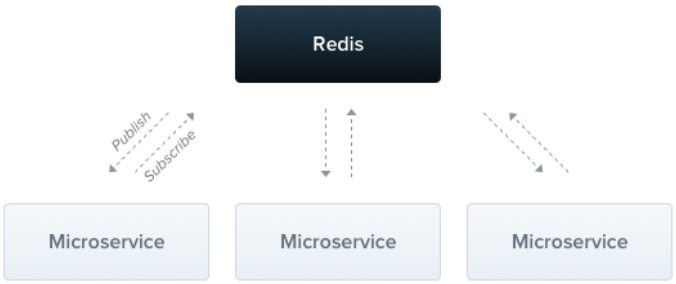
O modelo utilizado será o de monorepos, uma vez que conseguimos aproveitar algumas vantagens desse modelo em comparação com o de multi-repositórios.

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



Iremos utilizar o Redis para implementarmos a comunicação entre os microserviços, uma vez que o Tasks será independente do restante da aplicação. Para isso, instale o Docker Desktop (siga instruções aqui) e na raiz do projeto, crie um arquivo docker-compose.yml com o seguinte conteúdo:

```
services:
  redis: # Name of container
    image: redis
    ports:
      - 6379:6379
    volumes:
      - redis:/data
volumes:
  redis:
    driver: local
```

Iremos agora converter o nosso projeto para o sistema de monorepos. Para isso, execute o seguinte comando:

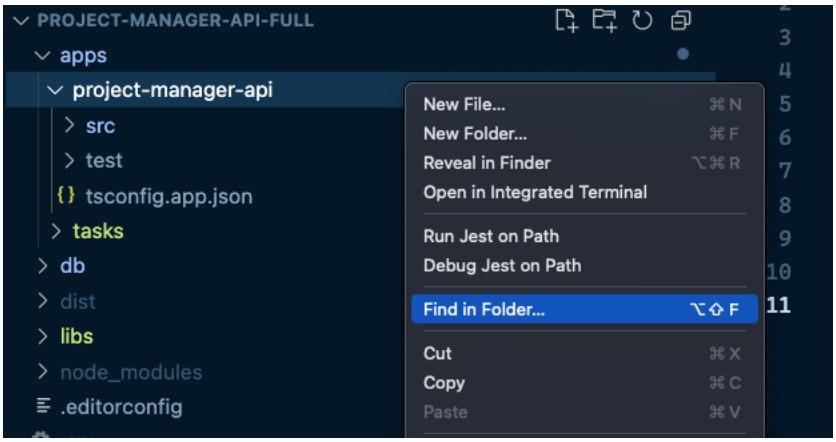
- **nest generate app tasks**

Isso irá transformar o projeto em um sistema de monorepos, com dois pacotes: project-manager-api e agora o tasks. Isso irá fazer com que os caminhos dos imports fiquem errados, pois os arquivos e pastas foram trocados de lugar. Para consertar esse problema, vamos buscar por todas as referências aos imports partindo da pasta src e trocá-los pelo alias do pacote do projeto principal.

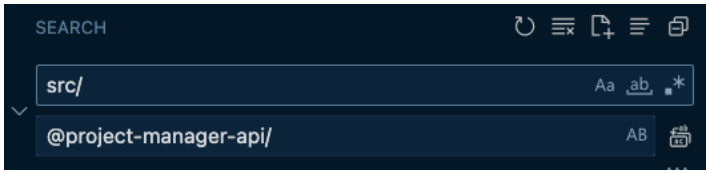
Primeiramente, no tsconfig.json, na propriedade **paths**, vamos adicionar um alias para o project-manager-api para facilitar os imports do projeto:

```
    "@project-manager-api/*": [
      "apps/project-manager-api/src/*"
    ],
```

Depois, siga o passo-a-passo do screenshot abaixo para consertar os imports:



Clique em “Find in Folder” (ou Encontrar na Pasta), em seguida clique na seta no canto esquerdo e coloque os seguintes dados:



O próximo passo é instalar os pacotes do redis e dos microserviços do nestj:

- **npm i --save @nestjs/microservices@10 ioredis redis@4**;

Passo 2 – Estrutura de microserviços

Substitua o arquivo main do pacote tasks para que possamos adicionar o transport do Redis e fazer com que a comunicação aconteça futuramente:

apps/tasks/src/main.ts

```
import { NestFactory } from '@nestjs/core';
import { Transport } from '@nestjs/microservices';
import { TasksModule } from './tasks.module';

async function bootstrap() {
  const app = await NestFactory.create(TasksModule);

  app.connectMicroservice(
    {
      transport: Transport.REDIS,
      options: {
        host: 'localhost',
        port: 6379,
      },
    },
    {
      inheritAppConfig: true,
    },
  );

  await app.startAllMicroservices();
}
bootstrap();
```

No controllers.module.ts iremos injetar um cliente da conexão da aplicação com o Redis para utilizarmos dentro do controller. Adicione o código abaixo nos ***imports*** do controllers.module.ts:

apps/project-manager-api/src/gateways/controllers/controllers.module.ts

```
    ClientsModule.register([
      { name: 'PROJECTS_MANAGER_API', transport: Transport.REDIS },
    ]),
```

Vamos criar uma biblioteca compartilhada entre os pacotes para compartilhar alguns arquivos e interfaces. Para isso, na raiz do projeto, execute o seguinte comando:

- **nest generate lib common**

Com a estrutura criada, iremos replicar uma forma simplificada da arquitetura CLEAN dentro do pacote Tasks. Para facilitar a estruturação das pastas, vamos executar um único comando que irá criar todas as pastas necessárias:

- **mkdir apps/tasks/src/gateways && mkdir apps/tasks/src/gateways/controllers && mkdir apps/tasks/src/infrastructure && mkdir apps/tasks/src/infrastructure/entities && mkdir apps/tasks/src/infrastructure/database && mkdir apps/tasks/src/domain && mkdir apps/tasks/src/domain/entities && mkdir apps/tasks/src/domain/interfaces && mkdir apps/tasks/src/domain/repositories && mkdir apps/tasks/src/domain/use-cases**

O comando é um pouco grande, porém a única coisa que ele faz é criar pastas dentro do pacote Tasks.

Vamos agora mover todos os arquivos relacionados ao Tasks de dentro do projects-manager-api para dentro da estrutura criada no pacote tasks. Para facilitar a manipulação dos arquivos, vamos seguir pela lista:

1. project-manager-api/domain/entities/task.ts -> tasks/src/domain/entities/task.ts;
2. project-manager-api/domain/interfaces/task.interface.ts -> tasks/src/domain/interfaces/task.interface.ts;
3. project-manager-api/domain/repositories/tasks-repository.interface.ts -> tasks/src/domain/repositories/tasks-repository.interface.ts;
4. todos os **arquivos** dentro de project-manager-api/domain/use-cases/tasks -> tasks/src/domain/use-cases;
5. project-manager-api/infrastructure/database/entities/task.entity.ts -> apps/tasks/src/infrastructure/entities/task.entity.ts;
6. apps/project-manager-api/src/infrastructure/database/repositories/tasks.repository.service.ts -> apps/tasks/src/infrastructure/repositories/tasks.repository.service.ts;
7. **Vamos apenas copiar o conteúdo do arquivo** apps/project-manager-api/src/gateways/controllers/tasks/tasks.controller.ts e colocar em apps/tasks/src/gateways/controllers/tasks.controller.ts;
8. apps/project-manager-api/src/gateways/controllers/tasks/dtos -> apps/tasks/src/gateways/controllers/dtos

Após mover os arquivos, iremos ver uma série de problemas de import. O mais fácil aqui é continuar os passos do exercício e, ao final, deixar que o próprio console nos mostre os erros. Como movemos muitos arquivos de uma vez, é normal que os caminhos fiquem incosistentes e que a aplicação não inicie.

Crie os módulos no pacote tasks conforme instruções abaixo:

apps/tasks/src/domain/domain.module.ts

```
import { Module } from '@nestjs/common';
import { TasksModule } from './use-cases/tasks.module';

@Module({
  imports: [TasksModule],
  exports: [TasksModule],
})
export class DomainModule {}
```

apps/tasks/src/gateways/gateways.module.ts

```
import { Module } from '@nestjs/common';
import { TasksController } from './controllers/tasks.controller';
import { DomainModule } from '../domain/domain.module';

@Module({
  imports: [DomainModule],
  controllers: [TasksController],
})
export class GatewaysModule {}
```

apps/tasks/src/infrastructure/infrastructure.module.ts

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { TaskEntity } from './entities/task.entity';
import { TasksRepositoryService } from
'./repositories/tasks.repository.service';
import { ProjectEntity } from '@project-manager-
api/infrastructure/database/entities/project.entity';
import { UserEntity } from '@project-manager-
api/infrastructure/database/entities/user.entity';

@Module({
  imports: [
    TypeOrmModule.forFeature([TaskEntity, ProjectEntity, UserEntity]),
    TypeOrmModule.forRoot({
      type: 'sqlite',
      database: 'db/sql.sqlite',
      entities: [__dirname + 'dist/**/*.entity{.ts,.js}'],
      synchronize: true,
      autoLoadEntities: true,
    }),
  ],
  providers: [TasksRepositoryService],
  exports: [TasksRepositoryService],
})
export class InfrastructureModule {}
```

apps/tasks/src/tasks.module.ts

```
import { Module } from '@nestjs/common';
import { InfrastructureModule } from
'./infrastructure/infrastructure.module';
import { GatewaysModule } from './gateways/gateways.module';

@Module({
  imports: [InfrastructureModule, GatewaysModule],
})
export class TasksModule {}
```

Passo 3 – Ajustes nas instâncias e dependências entre os projetos

Uma vez que removemos a entidade de Task do pacote principal, precisamos registrá-la novamente, agora com o caminho correto que aponta para o pacote tasks. Isso irá criar uma dependência entre os projetos (assim como aconteceu no passo anterior), mas o processo de refatoração de um monolito para microserviço é longo e gradativo. Vamos nos concentrar em fazer as aplicações serem instanciadas corretamente de forma independente e estabelecer uma ponte de comunicação entre elas.

apps/project-manager-api/src/infrastructure/database/database.module.ts

```
import { Module } from '@nestjs/common';
import { ProjectsRepositoryService } from
'./repositories/projects.repository.service';
import { UsersRepositoryService } from
'./repositories/users.repository.service';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UserEntity } from './entities/user.entity';
import { ProjectEntity } from './entities/project.entity';
import { TaskEntity } from
'apps/tasks/src/infrastructure/entities/task.entity';

@Module({
  imports: [
    TypeOrmModule.forFeature([UserEntity, ProjectEntity, TaskEntity]),
    TypeOrmModule.forRoot({
      type: 'sqlite',
      database: 'db/sql.sqlite',
      entities: [__dirname + 'dist/**/*.entity{.ts,.js}'],
      synchronize: true,
      autoLoadEntities: true,
    }),
  ],
```

```
  providers: [ProjectsRepositoryService, UsersRepositoryService],
  exports: [ProjectsRepositoryService, UsersRepositoryService],
})
export class DatabaseModule {}
```

Iremos remover também, por hora, as dependências do pacote Task com os repositórios do pacote principal. Para não copiarmos todos os códigos aqui dentro do exercício, consulte o código-fonte disponível no canvas e substitua os códigos de todos os arquivos dentro de apps/tasks/src/domain/use-cases pelos códigos disponíveis na plataforma.

Basicamente o que estamos fazendo é removendo a necessidade de uma consulta direta aos repositórios dos outros pacotes para que tais validações possam, futuramente, ser incorporadas em seus respectivos módulos.

Vamos ajustar também o código do tasks.controller do project-manager-api. Aqui estamos em um ponto chave: o controller que antes recebia requisições e repassava os dados para os use-cases, agora irá enviar uma mensagem para que um outro microserviço (tasks) trate a requisição:

apps/project-manager-api/src/gateways/controllers/tasks/tasks.controller.ts

```
import {
  Body,
  Controller,
  Get,
  Inject,
  NotFoundException,
  Param,
  Post,
  Req,
  UnprocessableEntityException,
} from '@nestjs/common';
import { CreateTaskDto } from './dtos/create-task.dto';
import { ClientProxy } from '@nestjs/microservices';

@Controller('tasks')
export class TasksController {
  constructor(
    @Inject('PROJECTS_MANAGER_API') private readonly redisClient:
ClientProxy,
  ) {}

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

```
      console.log('Disparando mensagem para Tasks');

      return this.redisClient.send(
        { cmd: 'get_tasks' },
        { 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 this.redisClient.send(
        { cmd: 'get_task_by_id' },
        {
          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 this.redisClient.send(
        { cmd: 'create_task' },
        {
          userId: loggedUser.sub,
          task: createTaskDto,
        },
      );
    } catch (error) {
      throw new UnprocessableEntityException(error.message);
    }
  }
}
```

No TasksController do pacote tasks, coloque o seguinte código:

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

```
import {
  Controller,
  NotFoundException,
  UnprocessableEntityException,
} from '@nestjs/common';
import { MessagePattern, Payload } from '@nestjs/microservices';
import { CreateTaskService } from '../../domain/use-cases/create-
task.service';
import { GetAllTasksService } from '../../domain/use-cases/get-all-
tasks.service';
import { GetTaskByIdService } from '../../domain/use-cases/get-task-by-
id.service';
import { CreateTaskDto } from './dtos/create-task.dto';

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

  @MessagePattern({ cmd: 'get_tasks' })
  async findAll(@Payload() data: { userId: number }) {
    try {
      console.log('recebendo mensagens em task');

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

  @MessagePattern({ cmd: 'get_task_by_id' })
  async findOne(@Payload() data: { userId: number; taskId: number }) {
    try {
      return await this.getTaskByIdUseCase.execute({
        userId: data.userId,
        taskId: data.taskId,
      });
    } catch (error) {
      throw new NotFoundException(error.message);
    }
```

```
  }

  @MessagePattern({ cmd: 'create_task' })
  async create(@Payload() data: { task: CreateTaskDto; userId: number }) {
    try {
      return await this.createTaskUseCase.execute({
        userId: data.userId,
        task: data.task,
      });
    } catch (error) {
      throw new UnprocessableEntityException(error.message);
    }
  }
}
```

O @MessagePattern é responsável por observar os eventos vindos de outros microserviço e executar ações sempre que esse evento for chamado.

O tasks.module.ts da pasta use-cases do micro serviço de Tasks também precisa de uma alteração, para que o módulo de infrastructure de Tasks seja uma dependência do seu módulo de casos de uso:

apps/tasks/src/domain/use-cases/tasks.module.ts

```
import { Module } from '@nestjs/common';
import { GetAllTasksService } from './get-all-tasks.service';
import { GetTaskByIdService } from './get-task-by-id.service';
import { CreateTaskService } from './create-task.service';
import { UpdateTaskService } from './update-task.service';
import { InfrastructureModule } from
'@tasks/infrastructure/infrastructure.module';

@Module({
  imports: [InfrastructureModule],
  providers: [
    GetAllTasksService,
    GetTaskByIdService,
    CreateTaskService,
    UpdateTaskService,
  ],
  exports: [
    GetAllTasksService,
    GetTaskByIdService,
    CreateTaskService,
    UpdateTaskService,
  ],
})
export class TasksModule {}
```

Passo 4 – Instância do Docker e ajustes nos caminhos

Inicie o docker na sua máquina, e em seguida abra o terminal na raiz da aplicação e digite o seguinte comando:

- **docker-compose up --build**

Caso encontre dificuldades para fazer com que o docker funcione, você também pode optar por basear o microserviço no protocolo TCP.

Com esses ajustes, podemos iniciar agora as duas aplicações separadamente. Lembre-sen que, nessa fase, possivelmente teremos alguns erros de caminhos ainda a serem corrigidos. Como

dito anteriormente, a forma mais fácil de resolvê-los é deixar o próprio terminal nos informar dos erros e usar o código-fonte disponibilizado no canvas para consulta.

Na raiz do projeto, rode o comando

- **npm run start:dev**

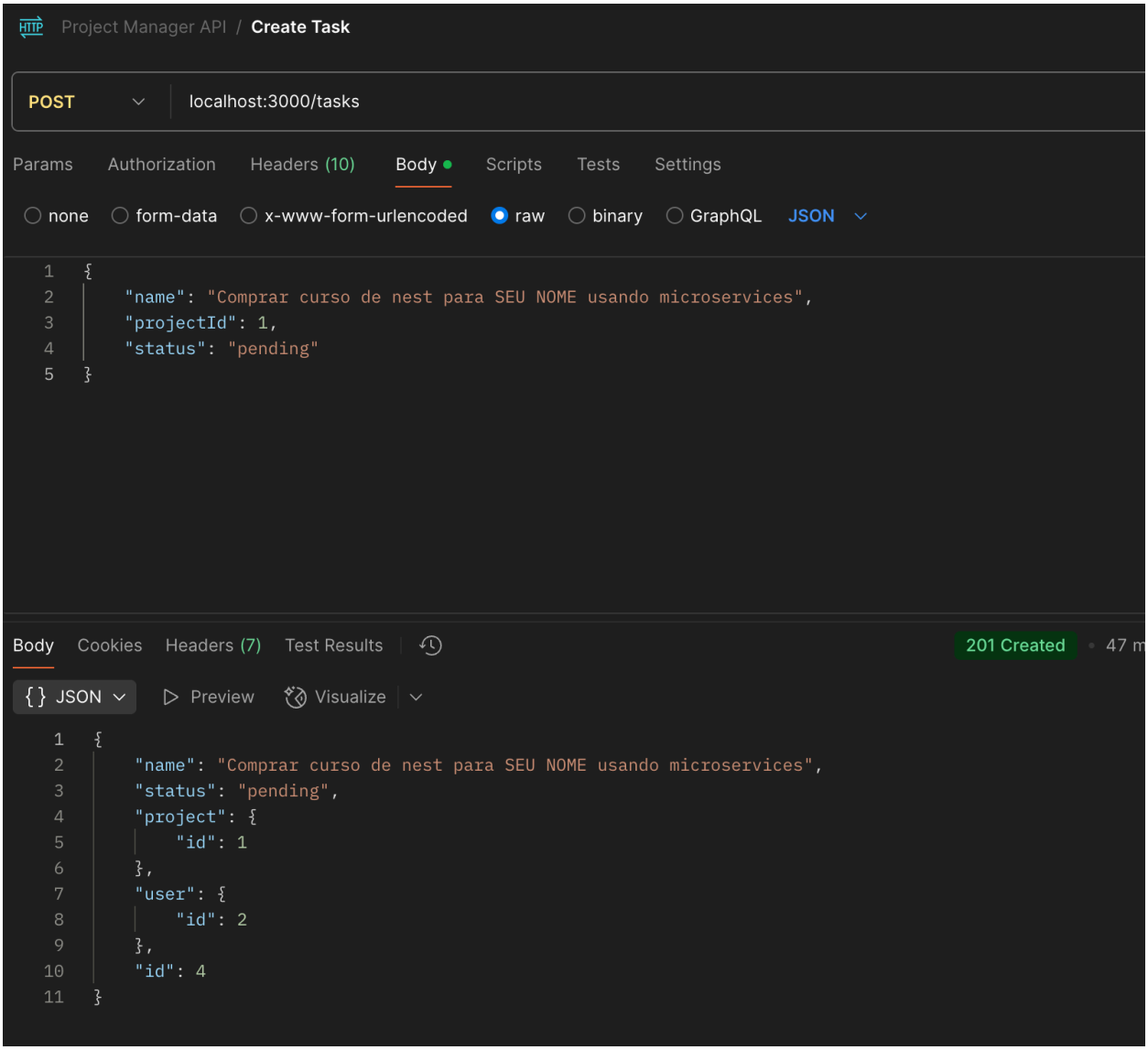
Abra uma outra aba (ou janela) do terminal e rode a aplicação tasks

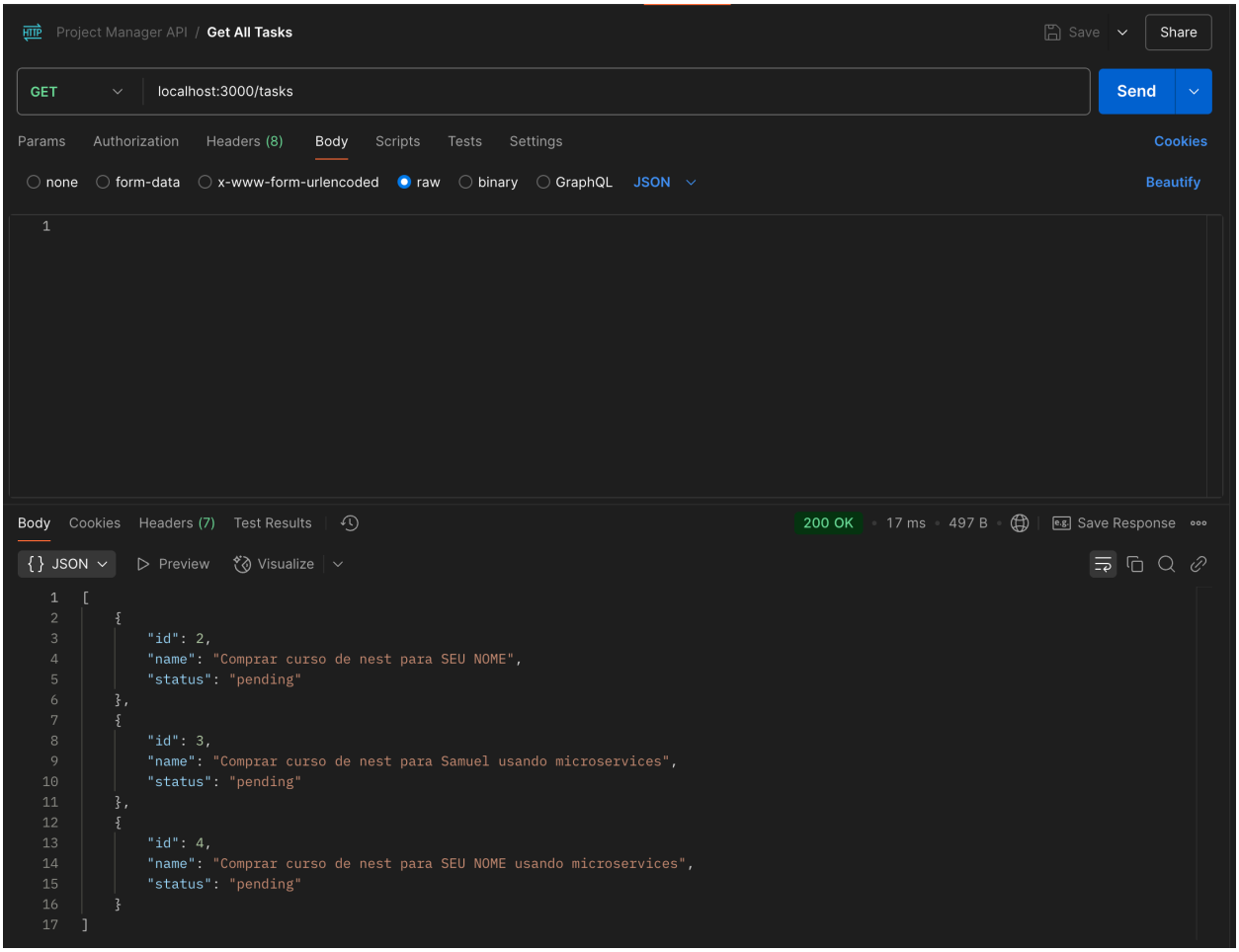
- **nest start tasks –watch**

Passo 5 – Testes e entrega

Abra o postman e faça o login com as suas credenciais. Em seguida, dispare uma requisição para tasks substituindo os dados “SEU NOME”.

Envie os prints juntamente com o código fonte






```
[Nest] 52907  - 03/17/2025, 6:53:02 PM     LOG [RoutesResolver] AuthController {/auth}: +0ms
[Nest] 52907  - 03/17/2025, 6:53:02 PM     LOG [RouterExplorer] Mapped {/auth/login, POST} route +0ms
[Nest] 52907  - 03/17/2025, 6:53:02 PM     LOG [NestApplication] Nest application successfully started +1ms
Disparando mensagem para Tasks - Samuel Martins
```

```
[Nest] 52854  - 03/17/2025, 6:52:54 PM     LOG [InstanceLoader] TasksModule dependencies initialized +0ms
[Nest] 52854  - 03/17/2025, 6:52:54 PM     LOG [InstanceLoader] GatewaysModule dependencies initialized +0ms
[Nest] 52854  - 03/17/2025, 6:52:54 PM     LOG [NestMicroservice] Nest microservice successfully started +30ms
recebendo mensagens em task - Samuel Martins
```