CRUD BackEnd

CRUD é um conjunto de operações básicas que podem ser realizadas em qualquer banco de dados ou sistema que lida com dados. As operações são as seguintes:

- **Create (Criação)**: Permite adicionar novos registros à sua base de dados.
- **Read (Leitura)**: Permite recuperar informações de registros existentes na base de dados.
- **Update (Atualização)**: Permite modificar informações em registros já existentes.
- **Delete (Exclusão)**: Permite remover registros da base de dados.

CRUD para Cadastro de Vendedores:

Create:

➔ Criar a entidade modelo para representar um vendedor. Definir os campos como CPF, E-mail, Nome, Telefone, Endereço, Data de Nascimento.

➔ Criar o controlador **VendedorController** e definir a rota **POST** para criar um novo vendedor. O controlador, receberá os dados do vendedor em uma solicitação e os salvará no banco de dados.

Read:

➔ Criar a rota **GET** no **VendedorController** para recuperar todos os vendedores ou uma rota **GET** com um parâmetro para buscar vendedores específicos por ID ou nome.

Update:

➔ Criar a rota **PUT** no **VendedorController** que permita a atualização dos detalhes de um vendedor específico com base em seu ID. Os dados atualizados serão recebidos na solicitação e aplicados ao registro existente no banco de dados.

Delete:

➔ Criar a rota **DELETE** no **VendedorController** que permita a exclusão de um vendedor com base em seu ID. Fazer a implementação de confirmação ou autenticação para evitar exclusões acidentais.

CRUD para Cadastro de Clientes:

Create:

➔ Criar a entidade modelo para representar um cliente. Definir os campos como Primeira compra, CPF, E-mail, Nome, Telefone, Endereço, Data de Nascimento, Sexo.

➔ Criar o controlador **ClienteController** e definir uma rota **POST** para criar um novo cliente. Receber os dados do cliente em uma solicitação e os armazene no banco de dados.

Read:

➔ Criar a rota **GET** no **ClienteController** para recuperar todos os clientes ou uma rota **GET** com parâmetros de pesquisa para encontrar clientes específicos.

Update:

➔ Criar a rota **PUT** no **ClienteController** que permita a atualização dos detalhes de um cliente específico com base em seu ID. Os dados atualizados serão recebidos na solicitação e aplicados ao registro existente no banco de dados.

Delete:

➔ Criar a rota **DELETE** no **ClienteController** para permitir a exclusão de um cliente com base em seu ID. Fazer a implementação de confirmação ou autenticação para evitar exclusões acidentais.

CRUD para Cadastro de Notas Fiscais:

Create:

➔ Criar a entidade modelo para representar uma nota fiscal. Defina os campos, como Código do Produto, Nome do Produto, Unidade de Medida, Grupo do Produto, Valor do Produto, Registro Fiscal, além dos campos relacionados ao cliente e vendedor associados à nota fiscal.

➔ Criar o controlador **NotaFiscalController** e definir uma rota **POST** para criar uma nova nota fiscal. Receber os dados da nota fiscal em uma solicitação e os associar ao cliente e vendedor apropriados no banco de dados.

Read (Leitura):

➔ Crie uma rota **GET** no **NotaFiscalController** para visualizar todas as notas fiscais associadas a um cliente ou vendedor específico. Usar parâmetros na rota para filtrar as notas fiscais por cliente ou vendedor.

Update (Atualização):

➔ Crie uma rota **PUT** no **NotaFiscalController** que permita a atualização das informações de uma nota fiscal específica com base em seu ID. Os dados atualizados serão recebidos na solicitação e aplicados ao registro existente no banco de dados.

Delete (Exclusão):

➔ Criar a rota **DELETE** no **NotaFiscalController** para permitir a exclusão de uma nota fiscal com base em seu ID. Fazer a implementação de confirmação ou autenticação para evitar exclusões acidentais.

Passo a Passo CRUD

Para criar a pasta

- **nest new** NOME-DA-PASTA

Gerar o módulo

- **nest generate module** NOME-DO-MODULO
- **cd** nome-do-projeto

Instale o Prisma

npm install @prisma/cli @prisma/client

Inicialize o Prisma

npx prisma init

Defina o modelo (entidade) no schema.prisma

```
model User {
  id    Int     @id @default(autoincrement())
  name  String
  email String  @unique
}
```

Gere o Prisma Client

npx prisma generate

Crie um serviço com Prisma

```
import { Injectable } from '@nestjs/common';
import { PrismaService } from './prisma.service';
import { Prisma } from '@prisma/client';

@Injectable()
export class UserService {
  constructor(private prisma: PrismaService) {}

  async findAll(): Promise<Prisma.User[]> {
```

```
    return this.prisma.user.findMany();
  }

  async findOne(id: number): Promise<Prisma.User | null> {
    return this.prisma.user.findUnique({
      where: { id },
    });
  }

  async create(data: Prisma.UserCreateInput): Promise<Prisma.User> {
    return this.prisma.user.create({
      data,
    });
  }

  async update(id: number, data: Prisma.UserUpdateInput): Promise<Prisma.User | null> {
    return this.prisma.user.update({
      where: { id },
      data,
    });
  }

  async remove(id: number): Promise<Prisma.User | null> {
    return this.prisma.user.delete({
      where: { id },
    });
  }
}
```

Gerar o Controlador

- **nest generate controller** NOME-DO-CONTROLLER

Gerar o módulo

- **nest generate module** NOME-DO-MODOLO

Defina as Rotas

xxxxxxxxx.controller.ts

```
import { Controller, Get, Post, Put, Delete } from '@nestjs/common';


@Controller('xxxxxxxx')
```

```
export class xxxxxxxxController {
  @Get()
  findAll(): string {
  Lógica para listar todos os xxxxxxxx
  }

  @Post()
  create(): string {
  Lógica para criar um novo  xxxxxxxx
  }
  @Put(':id')
  update(@Param('id') id: string): string {
  Lógica para atualizar um  xxxxxxxx pelo ID
  }

  @Delete(':id')
  remove(@Param('id') id: string): string {
  Lógica para excluir um  xxxxxxxx pelo ID
  }
}
```

b.kleuvyn

Conecte o controlador no app.module.ts

```
import { Module } from '@nestjs/common';
import { UserController } from './user.controller';
import { UserService } from './user.service';
import { PrismaService } from './prisma.service';

@Module({
  controllers: [UserController],
  providers: [UserService, PrismaService],
})
export class AppModule {}
```

Instalar Pacotes

- **npm install mysql2**

Configurar o Banco de Dados

app.module.ts

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',  Endereço do banco de dados
      port: 3306, Porta do banco de dados MySQL
      username: 'xxxxxxxx',
      password: 'xxxxxxxx',
      database: 'nome_do_banco', // Nome do banco de dados
      entities: [], // Entidades serão adicionadas mais tarde
      synchronize: true, // Isso cria tabelas automaticamente (apenas para desenvolvimento)
    }),
  ],
})
export class AppModule {}
```

b.kleuvyn

Criar Entidades

xxxxxxxx.entity.ts

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class xxxxxxxx {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  nome: string;

  @Column()
  email: string;
}
```

Registrar Entidades

app.module.ts

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
```

```
import { User } from './user.entity'; // Importe a entidade

@Module({
  imports: [
    TypeOrmModule.forRoot({
      // ... outras configurações
      entities: [User], // Registre a entidade aqui
      synchronize: true, // Apenas para desenvolvimento
    }),
  ],
})
export class AppModule {}
```

b.kleuvyn

Usar Entidades nos Serviços e Controladores

user.service.ts

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './user.entity';

@Injectable()
export class UserService {
  constructor(
    @InjectRepository(User)
    private userRepository: Repository<User>,
  ) {}

  async findAll(): Promise<User[]> {
    return this.userRepository.find();
  }

  async findOne(id: number): Promise<User> {
    return this.userRepository.findOne(id);
  }

  async create(user: User): Promise<User> {
    return this.userRepository.save(user);
  }

  async update(id: number, user: User): Promise<User> {
    await this.userRepository.update(id, user);
    return this.userRepository.findOne(id);
  }
```

```
  async remove(id: number): Promise<void> {
    await this.userRepository.delete(id);
  }
}
```

Teste

npm run start:dev