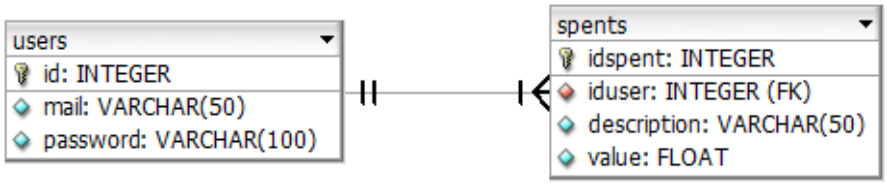
Objetivos:

I. Conexão com o SGBD PostgreSQL;

II. Conexão com o BD do SQLITE.

Nos exemplos considere as tabelas users e spents (gastos) representadas no modelo:



i. Conexão com o SGBD PostgreSQL

O projeto será organizado nas pastas:

- controllers: códigos para fazer a conexão com o SGBD e enviar os comandos SQL para o SGBD (Sistema de Gerenciamento de Banco de Dados);
- routes: códigos para processar as rotas direcionando para as funções objetivo. As "funções objetivos" das rotas serão os métodos create, list, delete e update das classes SpentController e UserController.

No arquivo db.ts foi criado um pool de conexão. Um pool de conexão é uma técnica utilizada em sistemas de banco de dados para gerenciar e reutilizar conexões com o SGBD. O objetivo principal é melhorar o desempenho e a eficiência na interação com o BD, especialmente em ambientes onde várias solicitações de conexão são feitas concorrentemente. Principais conceitos associados a um pool de conexão:

- Conexões com o BD: cada vez que um aplicativo precisa interagir com um BD, ele estabelece uma conexão. Essa conexão é um recurso valioso, e a criação e destruição frequentes de conexões prejudicam o desempenho;
- Pooling de conexões: em vez de criar uma conexão sempre que é necessário interagir com o BD, o pool mantém um conjunto de conexões pré-criadas e prontas para serem usadas;
- Reutilização de conexões: quando uma aplicação precisa realizar uma operação no BD, ela solicita uma conexão do pool. Após a conclusão da operação, a conexão é liberada de volta para o pool em vez de ser fechada. Isso permite que a conexão seja reutilizada por outras partes do código que necessitem de acesso ao BD;
- Benefícios:
  - Melhor desempenho: a reutilização de conexões reduz o tempo necessário para estabelecer novas conexões;
  - Redução de overhead: evita o custo associado à criação e destruição frequentes de conexões;
  - Controle de recursos: o pool limita o número total de conexões ativas, evitando sobrecarga no SGBD.

O uso de pools de conexão é uma prática comum em sistemas que exigem interações frequentes com BD, ajudando a otimizar recursos e melhorar o desempenho global do aplicativo.

Para fazer os testes, instale as seguintes dependências:

```
npm i pg
npm i @types/pg -D
```

No código a seguir, o construtor da classe `Pool` recebe como parâmetro um JSON com as propriedades de conexão com o SGBD, altere os valores das propriedades `database` e `password` para fazer a conexão com o seu BD.

A função `query` executa as consultas no SGBD. Como o resultado da chamada do método `pool.query` é um objeto com várias propriedades, então optou-se por extrair apenas as propriedades relevantes para cada comando SQL (insert, select, delete e update).

Arquivo: src/controllers/db.ts

```
import { Pool } from "pg";

const pool = new Pool({
  user: "postgres",
  host: "localhost",
  database: "bdaula",
  password: "123",
  port: 5432,
});

async function query(sql: string, params?: any[]) {
  try{
    const res = await pool.query(sql, params);
    if( res.command == 'INSERT' ){
      return res.rows[0];
    }
    else if( res.command == 'SELECT' ){
      return res.rows;
    }
    else if( res.command == 'DELETE' || res.command == 'UPDATE'){
      return {rowcount:res.rowCount};
    }
    else{
      return {sql};
    }
  }
  catch(e:any){
    return {message:e.message};
```

```
  }
};

export default query;
```

Os comandos para criar as tabelas foram colocados em um arquivo a parte pelo fato deles serem executados somente uma vez. Observe que os comandos SQL para criar as tabelas no SGBD são passados para a função `query` (do código anterior).

Adicione a propriedade `init` na propriedade `scripts` do arquivo package.json. Esse comando será uado para executar o código o código do arquivo src/controllers/init.ts e criar as tabelas `spents` e `users` no BD.

```
"scripts": {
  "init": "ts-node ./src/controllers/init",
  "dev": "ts-node-dev --respawn --transpile-only src/index.ts",
  "build": "tsc",
  "start": "node dist/index.js"
},
```

Utilize o comando `npm run init` para rodar o código do arquivo src/controllers/init.ts. Após executar o comando, abra o pgAdmin, ou algum outro cliente de PostgreSQL para checar se as tabelas `spents` e `users` foram criadas no BD.

Arquivo: src/controllers/init.ts

```
import query from "./db";

async function init() {
    return await query(`
        START TRANSACTION;
        DROP TABLE IF EXISTS spents, users;

        CREATE TABLE IF NOT EXISTS users (
            id serial PRIMARY KEY,
            mail VARCHAR(50) NOT NULL,
            password VARCHAR(100) NOT NULL
        );

        CREATE TABLE IF NOT EXISTS spents (
            id serial PRIMARY KEY,
            iduser integer not null,
            description VARCHAR(50) NOT NULL,
            value decimal(10,2) NOT NULL,
            constraint fk_iduser
                foreign key (iduser)
                references users (id)
                on delete cascade
```

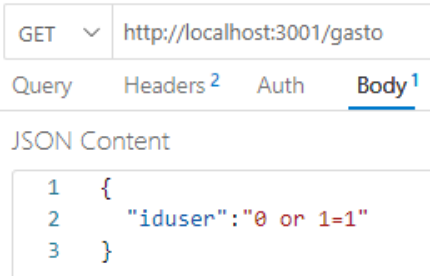
```
                on update cascade
        );
        COMMIT;
    `);
}

init()
.then((r) => console.log(r))
.catch((e) => console.log(e));
```

Nas classes `UserController` e `SpentController` estão os métodos para fazer o CRUD (Create, Read, Update e Delete) nas tabelas. Os comandos SQL são submetidos usando a função `query`, do arquivo db.ts. A função `query` pode receber como 2º parâmetro um array com os valores a serem usados no comando SQL – esse recurso evita injeção de SQL.

Para evitar ataques de injeção de SQL podemos utilizar consultas parametrizadas. No exemplo a seguir, os marcadores de posição de parâmetro `$1` e `$2` serão substituídos pelos valores passados no array `[mail,password]`, onde `$1` receberá o valor da 1ª posição do array e `$2` receberá o valor da 2ª posição do array.

```
const r:any = await query(
  "INSERT INTO users(mail,password) VALUES ($1,$2) RETURNING id",
  [mail,password]
);
```

A injeção de SQL é uma vulnerabilidade que ocorre quando dados não confiáveis são incorporados diretamente em instruções SQL sem a devida validação ou tratamento.



Considere como exemplo que a consulta do método `list`, da classe `SpentController`, receba o valor da variável `iduser` diretamente no comando SQL, então o usuário poderia fornecer `"0 or 1=1"` como valor do parâmetro `iduser` fazendo com que a consulta retornasse todos os registros da tabela users, pois seria executado o seguinte comando SQL:

```
SELECT id,description,value FROM spents WHERE iduser=0 or 1=1 ORDER BY id DESC
```

```
public async list(req: Request, res: Response): Promise<Response> {
  const { iduser } = req.body;
  const r:any = await query(
    `SELECT id,description,value FROM spents WHERE iduser=${iduser} ORDER BY id DESC`
  );
  return res.json(r);
}
```

O uso de consultas parametrizadas ajuda a prevenir ataques de injeção de SQL, pois os valores dos parâmetros são tratados separadamente da instrução SQL, reduzindo a possibilidade de manipulação maliciosa. Além de facilitar a reutilização da consulta com diferentes conjuntos de valores.

Arquivo: src/controllers/UserController.ts

```
import { Request, Response } from "express";
import query from "./db";

class UserController {
  public async create(req: Request, res: Response): Promise<void> {
    const { mail, password } = req.body;
    const r: any = await query(
      "INSERT INTO users(mail,password) VALUES ($1,$2) RETURNING id",
      [mail, password]
    );
    res.json(r);
  }

  public async list(_: Request, res: Response): Promise<void> {
    const r: any = await query("SELECT id,mail FROM users ORDER BY mail");
    res.json(r);
  }

  public async delete(req: Request, res: Response): Promise<void> {
    const { id } = req.body; // id do registro a ser excluído
    const r: any = await query("DELETE FROM users WHERE id = $1", [id]);
    res.json(r);
  }

  public async update(req: Request, res: Response): Promise<void> {
    const { id, mail, password } = req.body;
    const r: any = await query(
      "UPDATE users SET mail=$2, password=$3 WHERE id=$1",
      [id, mail, password]
    );
    res.json(r);
  }
}

export default new UserController();
```

Arquivo: src/controllers/SpentController.ts

```
import { Request, Response } from "express";
import query from "./db";
```

```
class SpentController {
  public async create(req: Request, res: Response): Promise<void> {
    const { iduser, description, value } = req.body;
    const r:any = await query(
      "INSERT INTO spents(iduser, description, value) VALUES ($1,$2,$3) RETURNING id",
      [iduser, description, value]
    );
    res.json(r);
  }

  public async list(req: Request, res: Response): Promise<void> {
    const { iduser } = req.body;
    const r:any = await query(
      "SELECT id,description,value FROM spents WHERE iduser=$1 ORDER BY id DESC",
      [iduser]
    );
    res.json(r);
  }

  public async delete(req: Request, res: Response): Promise<void> {
    const { id } = req.body; // id do registro a ser excluído
    const r:any = await query(
      "DELETE FROM spents WHERE id = $1", [id]
    );
    res.json(r);
  }

  public async update(req: Request, res: Response): Promise<void> {
    const { id, description, value } = req.body;
    const r:any = await query(
      "UPDATE spents SET description=$2, value=$3 WHERE id=$1",
      [id,description, value]
    );
    res.json(r);
  }
}

export default new SpentController();
```

A seguir tem-se o código das rotas para os métodos das classes `UserController` e `SpentController`. Observe que os arquivos são semelhantes, com exceção da importação (sinalizadas em verde).

Arquivo: src/routes/user.ts

```
import { Router } from "express";
import controller from "../controllers/UserController";

const routes = Router();
-
```

```
routes.post('/', controller.create);
routes.get('/', controller.list);
routes.delete('/', controller.delete);
routes.put('/', controller.update);

export default routes;
```

Arquivo: src/routes/spent.ts

```
import { Router } from "express";
import controller from "../controllers/SpentController";

const routes = Router();

routes.post('/', controller.create);
routes.get('/', controller.list);
routes.delete('/', controller.delete);
routes.put('/', controller.update);

export default routes;
```

A seguir tem-se o código para as rotas.

Arquivo: src/routes/index.ts

```
import { Router, Request, Response } from "express";
import user from './user';
import spent from './spent';

const routes = Router();

routes.use("/usuario", user);
routes.use("/gasto", spent);

//aceita qualquer método HTTP ou URL
routes.use((_: Request, res: Response) => {
  res.json({ error: "Requisição desconhecida" });
});

export default routes;
```

A seguir tem-se o código para subir a aplicação na porta definida no arquivo .env.

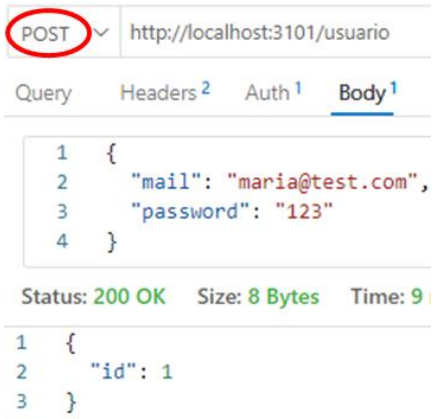
Arquivo: src/index.ts

```
import express from "express";
import dotenv from "dotenv";
import routes from "./routes";
```

```
dotenv.config();

const app = express();
const PORT = process.env.PORT || 3000;

app.use(express.json());

app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});

// define a rota para o pacote /routes
app.use(routes);
```

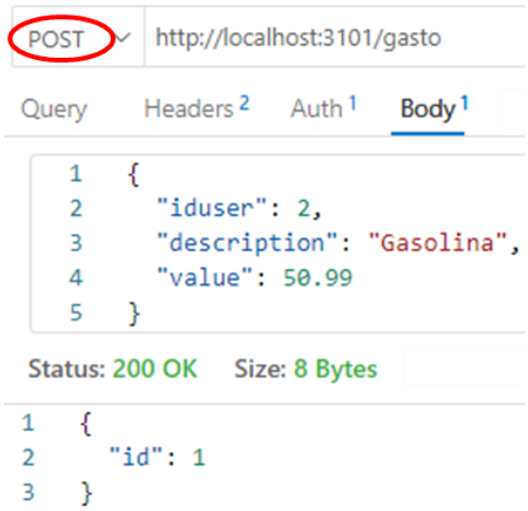
Testes das rotas:

Inserir um usuário:

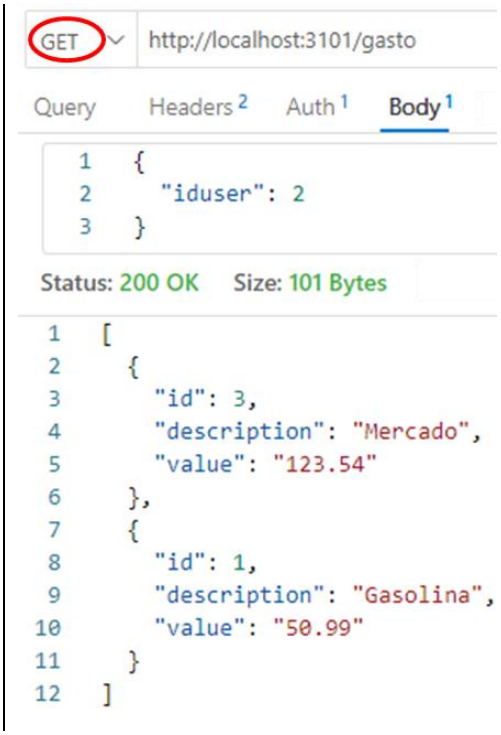


Listar os usuários:

Inserir um gasto:



Listar os gastos do usuário que possui id igual a 2:



ii. Conexão com o BD do SQLite

Crie um projeto na pasta sqlite ou qualquer outro nome. O projeto deverá ter os mesmos arquivos e pastas do projeto que utilizamos para fazer a conexão com o SGBD PostgreSQL. Faça as seguintes modificações no projeto, ao invés de instalar o pacote `pg` instalaremos o pacote `better-sqlite3`:

- No terminal, execute o comando `npm i better-sqlite3` para instalar a biblioteca que possui ferramentas para acessar o BD do SQLite (https://www.npmjs.com/package/better-sqlite3);
- No terminal, execute o comando `npm i -D @types/better-sqlite3` para instalar o pacote que contém as definições de tipos do pacote better-sqlite3;
- Coloque as variáveis a seguir no arquivo `.env`. No SQLite o BD é um arquivo, a variável DBNAME possui o nome do arquivo a ser criado na raiz do projeto:

```
PORT = 3102
DBNAME = bdaula.db
```

- Coloque o código a seguir no arquivo src/controllers/db.ts. A variável db é usada para manter a conexão, desta forma, a função `connection` retorna sempre a mesma conexão como BD.

Arquivo: src/controllers/db.ts

```
import Database, { Database as DatabaseProps } from "better-sqlite3";
import dotenv from "dotenv";
dotenv.config();

let db: DatabaseProps | null = null;
```

```
// retorna a conexão com o BD
export default function connection(): DatabaseProps {
  if (!db || !db.open) {
    const databasename = process.env.DBNAME || "test.db";
    db = new Database(`./${databasename}`);
  }
  return db;
}
```

- Coloque o código a seguir no arquivo src/controllers/init.ts. Utilize o comando `npm run init`, definido no arquivo package.json, para rodar o código do arquivo src/controllers/init.ts.

Arquivo: src/controllers/init.ts

```
import connection from "./db";

function init(){
  const db = connection();// obtém a conexão com o BD

  try {
    db.exec(`
        BEGIN;
        DROP TABLE IF EXISTS users;
        DROP TABLE IF EXISTS spents;
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            mail VARCHAR(50) NOT NULL,
            password VARCHAR(100) NOT NULL
        );
        CREATE TABLE IF NOT EXISTS spents (
          id INTEGER PRIMARY KEY AUTOINCREMENT,
          iduser INTEGER NOT NULL,
          description VARCHAR(50) NOT NULL,
          value DECIMAL(10, 2) NOT NULL,
          CONSTRAINT fk_iduser
            FOREIGN KEY (iduser)
            REFERENCES users (id)
            ON DELETE CASCADE
            ON UPDATE CASCADE
        );
        COMMIT;
    `);
  } catch (error:any) {
    console.error("Erro:", error.message);
  } finally {
    db.close(); // fecha a conexão
  }
};
```

```
init();
```

- Coloque os códigos a seguir nos arquivos UserController.ts e SpentController.ts. Observe que os cabeçalhos dos métodos são semelhantes aos utilizados no SGBD PostgreSQL.

  O método `prepare` recebe o comando SQL e retorna um objeto do tipo `Statement`.

  O objeto `Statement` possui, entre outros, os métodos `run`, `get` e `all`:

  - Método run: é usado para executar um comando SQL que não retorna um conjunto de resultados, como o SELECT. Ele é adequado para instruções como INSERT, UPDATE e DELETE, que não precisam retornar dados;
  - Método get: é usado para executar um comando SQL que retorna uma única linha de resultado. Ele é adequado para obter uma única linha de um SELECT;
  - Método all: é usado para executar um comando SQL que retorna várias linhas de resultados. Ele é apropriado para consultas SELECT que podem retornar várias linhas de dados.

  Na versão usando SQLite, os marcadores de posição de parâmetro são identificados por `?` e serão substituídos pelos valores passados na chamada dos métodos `run`, `get` e `all`. No exemplo a seguir, o 1º parâmetro `?` do comando SQL será substituído pelo valor do 1º parâmetro fornecido no método `run` e assim sucessivamente.

```
const r = db
  .prepare("INSERT INTO users(mail,password) VALUES(?,?)")
  .run(mail, password);
```

Arquivo: src/controllers/UserController.ts

```
import { Request, Response } from "express";
import connection from "./db";

class UserController {
  public async create(req: Request, res: Response): Promise<void> {
    const { mail, password } = req.body;
    const db = connection(); // precisa obter a conexão
    try {
      const r = db
        .prepare("INSERT INTO users(mail,password) VALUES(?,?)")
        .run(mail, password);
      res.json(r);
    } catch (e: any) {
      res.json({ error: e.message });
    } finally {
      db.close(); // fecha a conexão
    }
  }
```

```
  public async list(_: Request, res: Response): Promise<void> {
    const db = connection();
    const r = db.prepare("SELECT id,mail FROM users ORDER BY mail").all();
    res.json(r);
  }

  public async delete(req: Request, res: Response): Promise<void> {
    const { id } = req.body; // id do registro a ser excluído
    const db = connection();
    const r = db.prepare("DELETE FROM users WHERE id = ?").run(id);
    res.json(r);
  }

  public async update(req: Request, res: Response): Promise<void> {
    const { id, mail, password } = req.body;
    const db = connection();
    const r = db
      .prepare("UPDATE users SET mail=?, password=? WHERE id=?")
      .run(mail, password, id);
    res.json(r);
  }
}

export default new UserController();
```

Arquivo: src/controllers/SpentController.ts

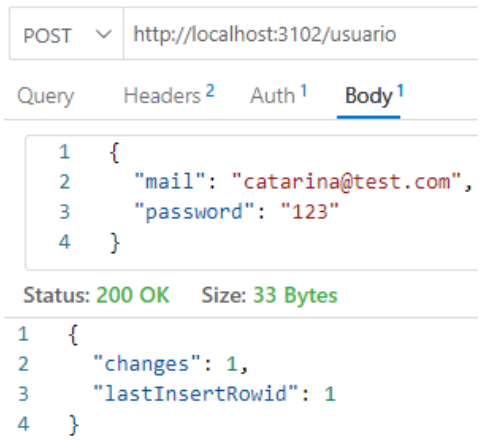
```
import { Request, Response } from "express";
import connection from "./db";

class SpentController {
  public async create(req: Request, res: Response): Promise<void> {
    const { iduser, description, value } = req.body;
    const db = connection(); // precisa obter a conexão
    try {
      const r = db
        .prepare("INSERT INTO spents(iduser,description,value) VALUES(?,?,?)")
        .run(iduser, description, value);
      res.json(r);
    } catch (e: any) {
      res.json({ error: e.message });
    } finally {
      db.close(); // fecha a conexão
    }
  }

  public async list(req: Request, res: Response): Promise<void> {
    const { iduser } = req.body;
```

```
    const db = connection();
    const r = db
      .prepare(
        "SELECT id,description,value FROM spents WHERE iduser=? ORDER BY id DESC"
      )
      .all(iduser);
    res.json(r);
  }

  public async delete(req: Request, res: Response): Promise<void> {
    const { id } = req.body; // id do registro a ser excluído
    const db = connection();
    const r = db.prepare("DELETE FROM spents WHERE id = ?").run(id);
    res.json(r);
  }

  public async update(req: Request, res: Response): Promise<void> {
    const { id, description, value } = req.body;
    const db = connection();
    const r = db
      .prepare("UPDATE spents SET description=?, value=? WHERE id=?")
      .run(description, value, id);
    res.json(r);
  }
}

export default new SpentController();
```

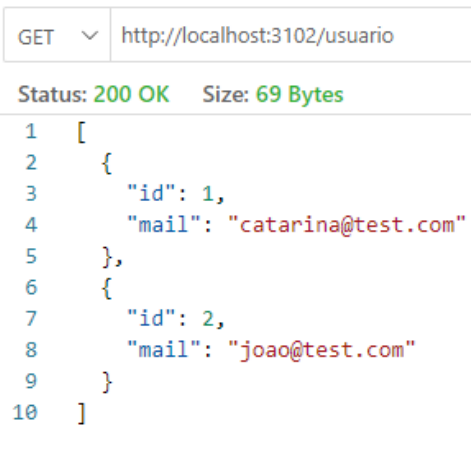
- Os demais arquivos do projeto são iguais à versão utilizada para conectar ao SGBD PostgreSQL;
- Antes de subir o servidor é necessário executar o comando `npm run init` para criar as tabelas no BD.

Testes das rotas:

Inserir um usuário:



Listar os usuários:



Inserir um gasto:

POST http://localhost:3102/gasto

Query Headers 2 Auth 1 Body 1

```
{
  "iduser": 1,
  "description": "Mercado",
  "value": 78.25
}
```

Status: 200 OK Size: 33 Bytes

```
{
  "changes": 1,
  "lastInsertRowid": 1
}
```

Listar os gastos do usuário que possui id igual a 1:

GET http://localhost:3102/gasto

Query Headers 2 Auth 1 Body 1

```
{
  "iduser": 1
}
```

Status: 200 OK Size: 91 Bytes

```
[
  {
    "id": 2,
    "description": "Cinema",
    "value": 20
  },
  {
    "id": 1,
    "description": "Mercado",
    "value": 78.25
  }
]
```

iii. Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

Exercício 1 - https://youtu.be/Tg8Aj-P1IVA

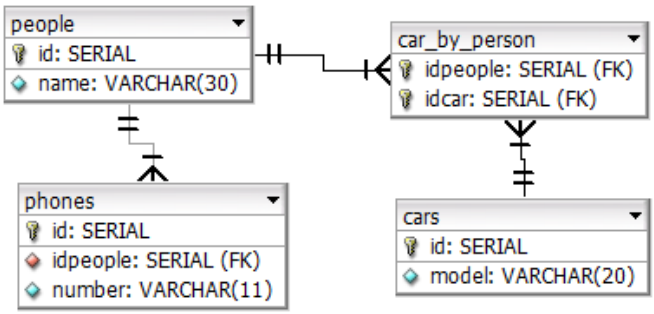
Exercício 2 - https://youtu.be/KsZRgzZhxMY

Exercício 3 - https://youtu.be/0z_S-2JzkqA

Exercício 4 - https://youtu.be/Uxi6q6BB06Q

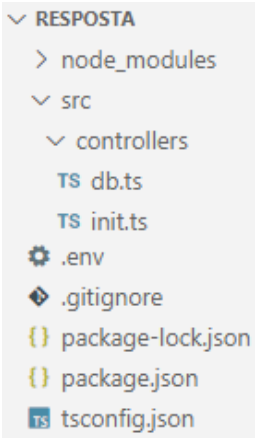
Exercício 5 - https://youtu.be/fi-Brfpq50U

Exercício 1 – Criar um projeto para fazer a conexão com o SGBD PostgreSQL com a estrutura mostrada ao lado. No arquivo db.ts deverão estar os parâmetros de conexão com o SGBD e no arquivo init.ts deverão estar os comandos SQL para criar as tabelas a seguir.



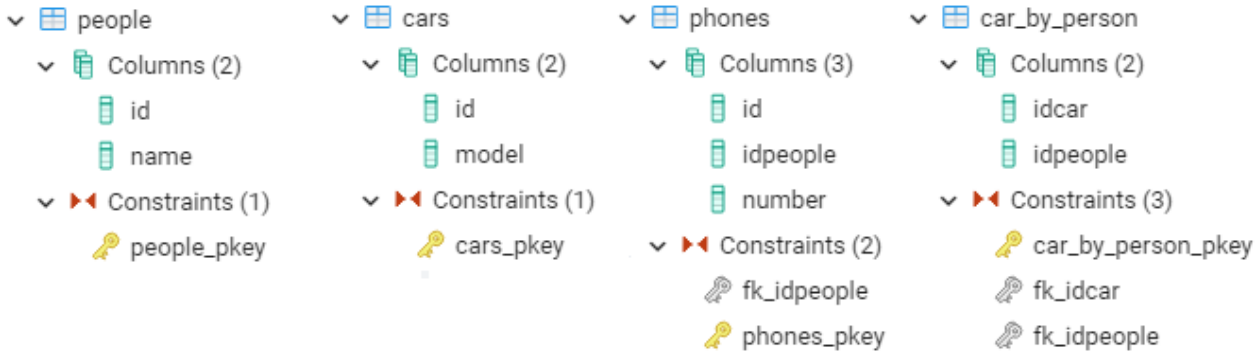
Os parâmetros de conexão com o SGBD deverão estar em variáveis de ambiente declaradas no arquivo .env. A seguir tem-se o conteúdo do arquivo .env:

Estrutura atual do projeto:



```
PORT = 3003
USER = postgres
HOST = localhost
DATABASE = bdaula
PASSWORD = 123
SGBDPORT = 5432
```

O resultado deverá ser as seguintes tabelas no SGBD PostgreSQL após executar o comando `npm run init`:



Observações:

- As constraints (restrições) people_pkey, cars_pkey, phones_pkey e car_by_person_pkey são as chaves primárias.
- As constraints fk_idpeople e fk_idcar foram os nomes dados às constraints de criação das chaves estrangeiras nos comandos de criação das tabelas.

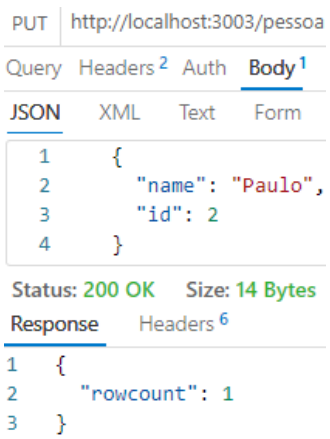
Exercício 2 – Codificar o controlador e as rotas para fazer o CRUD na tabela people. As rotas deverão usar os métodos HTTP GET (listar), POST (criar), PUT (atualizar) e DELETE e o caminho /pessoa. A seguir tem-se exemplos de uso das rotas.

Estrutura atual do projeto:

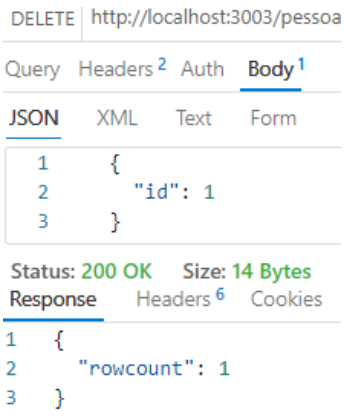
HTTP POST

HTTP GET

HTTP PUT



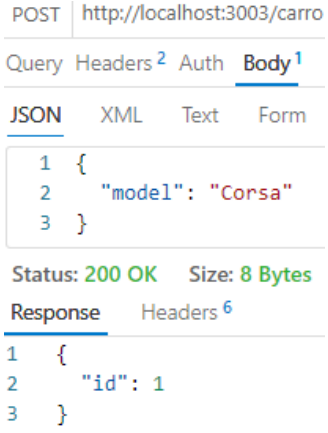
HTTP DELETE



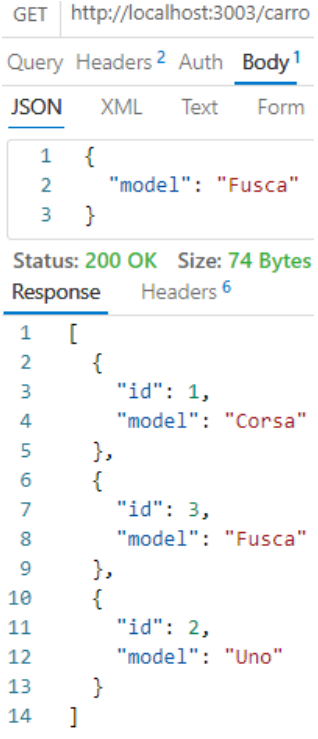
Exercício 3 – Codificar o controlador e as rotas para fazer o CRUD na tabela cars. As rotas deverão usar os métodos HTTP GET (listar), POST (criar), PUT (atualizar) e DELETE e caminho /carro. A seguir tem-se exemplos de uso das rotas.

Estrutura atual do projeto:

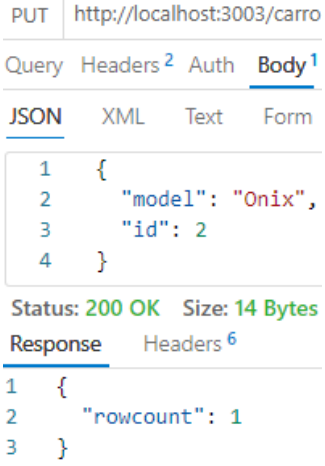
HTTP POST



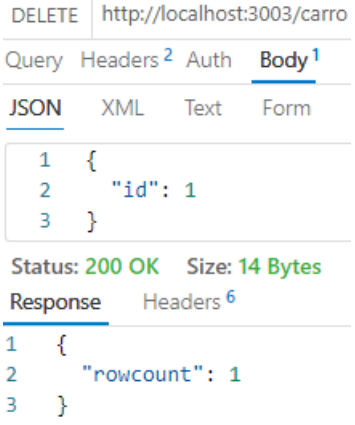
HTTP GET



HTTP PUT

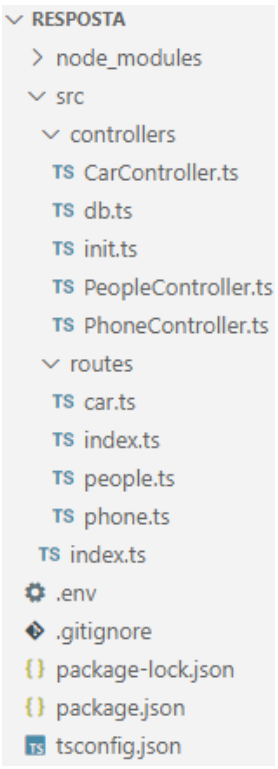


HTTP DELETE

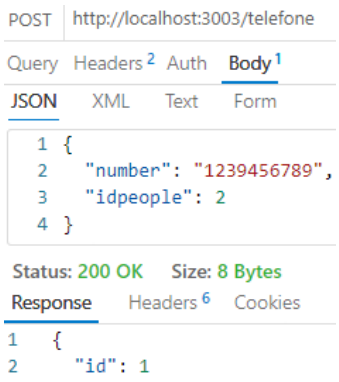


Exercício 4 – Codificar o controlador e as rotas para fazer o CRUD na tabela phones. As rotas deverão usar os métodos HTTP GET (listar), POST (criar), PUT (atualizar) e DELETE e caminho /telefone. A seguir tem-se exemplos de uso das rotas.

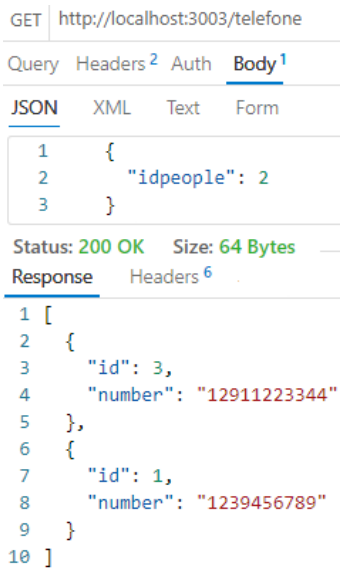
Estrutura atual do projeto:



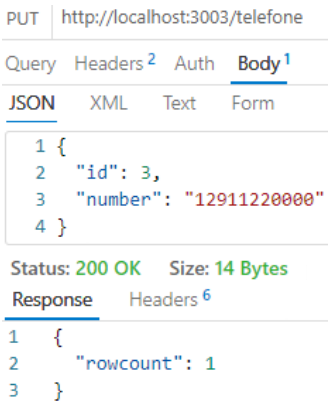
HTTP POST



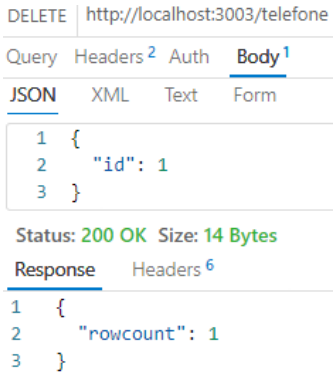
HTTP GET



HTTP PUT



HTTP DELETE



Exercício 5 – Codificar o controlador e as rotas para fazer as operações de insert, select e delete na tabela car_by_person. As rotas deverão usar os métodos HTTP GET (listar), POST (criar) e DELETE e caminho /carro_por_pessoa.

Estrutura atual do projeto: