

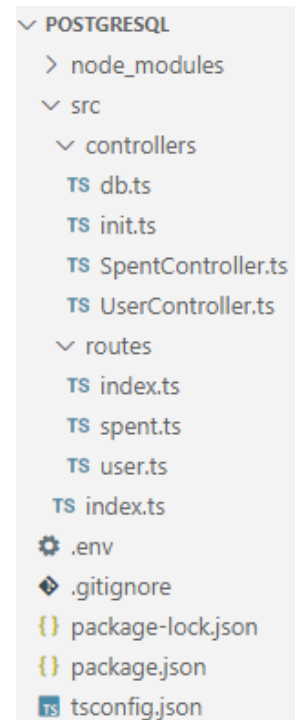
## Objetivos:

- I. Conexão com o SGBD PostgreSQL;
- II. Conexão com o BD do SQLITE.

Faremos dois projetos um para conectar com o SGBD PostgreSQL e com o BD do SQLite. Siga as instruções para criar o projeto para reproduzir o exemplo do SGBD PostgreSQL.

- a) Crie uma pasta de nome `postgresql` (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;
- b) Abra a pasta `postgresql` no VS Code e acesse o terminal do VS Code;
- c) No terminal, execute o comando `npm init -y` para criar o arquivo fundamental de um projeto Node, arquivo `package.json`;
- d) No terminal, execute o comando `npm i express` para instalar o pacote express;
- e) No terminal, execute o comando `npm i -D @types/express` para instalar o pacote que contém as definições de tipos do pacote express. Quando usamos um pacote é preciso ter acesso às declarações de tipo do pacote para que o TS saiba quais tipos de dados esperar do framework;
- f) No terminal, execute o comando `npm i dotenv` para instalar o pacote dotenv. As variáveis de ambientes são acessadas através do objeto `process.env`. Porém, as variáveis declaradas no arquivo `.env` não são carregadas pelo ambiente de execução do Node no objeto `process.env`. Usaremos o dotenv para carregar as variáveis do arquivo `.env` no objeto `process.env`;
- g) No terminal, execute o comando `npm i pg` para instalar a biblioteca que possui ferramentas para acessar o SGBD PostgreSQL (<https://www.npmjs.com/package/pg>);
- h) No terminal, execute o comando `npm i -D @types/pg` para instalar o pacote que contém as definições de tipos do pacote pg;
- i) No terminal, execute o comando `npm i -D ts-node ts-node-dev typescript` para instalar os pacotes ts-node, ts-node-dev e typescript como dependências de desenvolvimento;
- j) No terminal, execute o comando `tsc --init` para criar o arquivo de opções e configurações para o compilador TS (arquivo `tsconfig.json`);
- k) Crie o arquivo `.gitignore` na raiz do projeto e coloque a linha para ignorar a pasta `node_modules`;
- l) Crie o arquivo `.env` na raiz do projeto e coloque a seguinte variável de ambiente:  
`PORT = 3001`

Estrutura de pastas e arquivos do projeto:



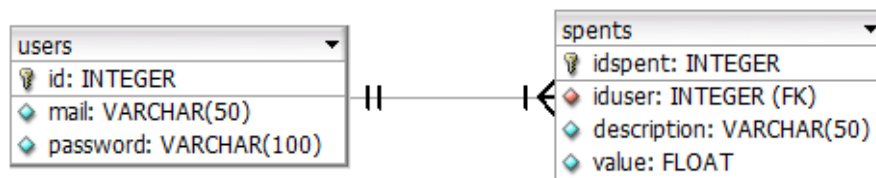
- m) Coloque as seguintes propriedades no arquivo package.json. Elas serão utilizadas para criar as tabelas no SGBD e rodar a aplicação:

```
"scripts": {
  "init": "ts-node ./src/controllers/init",
  "start": "ts-node ./src",
  "dev": "ts-node-dev ./src"
},
```

- n) Crie a pasta `src` na raiz do projeto;
- o) Crie as pastas `entities`, `controllers` e `routes` na pasta `src`;
- p) Crie os arquivos `index.ts` e `data-source.ts` na pasta `src` e os demais arquivos nas pastas `controllers`, `entities` e `routes`.

No momento o projeto terá a estrutura mostrada ao lado.

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.

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). 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 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 spends 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 spends 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.

GET	http://localhost:3001/gasto		
Query	Headers <sup>2</sup>	Auth	<u>Body<sup>1</sup></u>
JSON Content			
1	{		
2	"idUser":	"0 or 1=1"	
3	}		

Arquivo: `src/controllers/UserController.ts`

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

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

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

```
}

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

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

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

  public async delete(req: Request, res: Response): Promise<Response> {
    const { id } = req.body; // id do registro a ser excluído
    const r:any = await query(
```

```

        "DELETE FROM spents WHERE id = $1", [id]
    );
    return res.json(r);
}

public async update(req: Request, res: Response): Promise<Response> {
    const { id, description, value } = req.body;
    const r:any = await query(
        "UPDATE spents SET description=$2, value=$3 WHERE id=$1",
        [id,description, value]
    );
    return 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( (_,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 routes from './routes';
import dotenv from "dotenv";
dotenv.config();

// será usado 3000 se a variável de ambiente não tiver sido definida
const PORT = process.env.PORT || 3000;
const app = express(); // cria o servidor e coloca na variável app
// suportar parâmetros JSON no body da requisição
app.use(express.json());
// inicializa o servidor na porta especificada
app.listen(PORT, () => {
  console.log(`Rodando na porta ${PORT}`);
});

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

## 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 = 3002
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<Response> {
        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);
        return res.json(r);
    } catch (e: any) {
        return res.json({ error: e.message });
    } finally {
        db.close(); // fecha a conexão
    }
}

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

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

public async update(req: Request, res: Response): Promise<Response> {
    const { id, mail, password } = req.body;
    const db = connection();
    const r = db
        .prepare("UPDATE users SET mail=?, password=? WHERE id=?")
        .run(mail, password, id);
    return 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<Response> {
        const { iduser, description, value } = req.body;
        const db = connection(); // precisa obter a conexão
        try {
            const r = db
                .prepare("INSERT INTO spends(iduser,description,value) VALUES(?,?,?)")
                .run(iduser, description, value);
            return res.json(r);
        } catch (e: any) {

```

```
    return res.json({ error: e.message });
  } finally {
    db.close(); // fecha a conexão
  }
}

public async list(req: Request, res: Response): Promise<Response> {
  const { iduser } = req.body;
  const db = connection();
  const r = db
    .prepare(
      "SELECT id,description,value FROM spends WHERE iduser=? ORDER BY id DESC"
    )
    .all(iduser);
  return res.json(r);
}

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

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

export default new SpentController();
```

- Os demais arquivos do projeto são iguais à versão utilizada para conectar ao SGBD PostgreSQL.

## Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

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

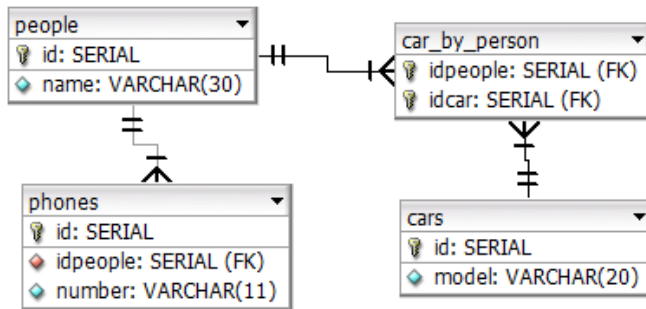
Exercício 2 - <https://youtu.be/KsZRgzZhMY>

Exercício 3 - [https://youtu.be/0z\\_S-2JzkqA](https://youtu.be/0z_S-2JzkqA)

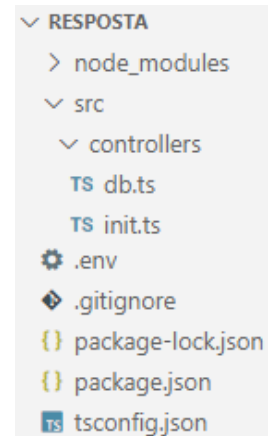
Exercício 4 - <https://youtu.be/Uxi6q6BB06Q>

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.



Estrutura atual do projeto:



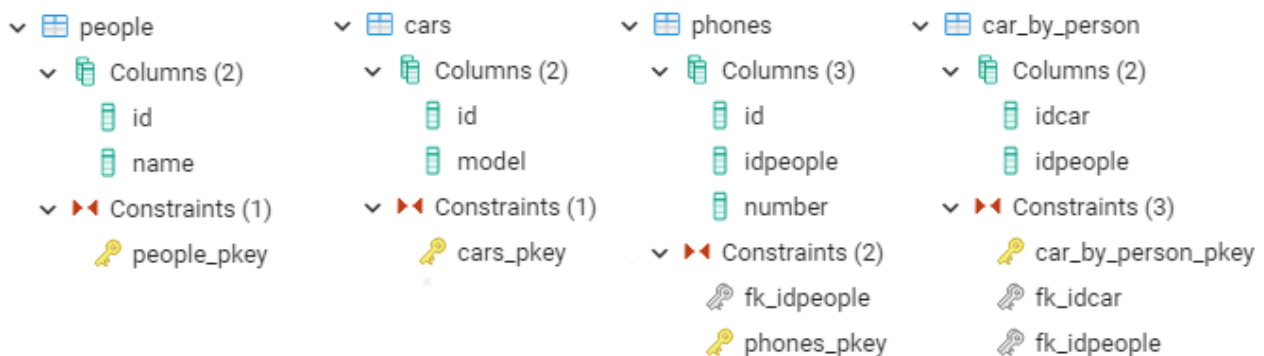
Os parâmetros de conexão com o SGBD deverão estar em variáveis de ambiente declaradas no arquivo .env. A seguir tem-se o conteúdo do arquivo .env:

```

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

Observação: as variáveis de ambiente são acessadas usando o objeto process.env. Porém, os valores das variáveis declaradas no arquivo .env são sempre texto. Desta forma, o conteúdo da variável PORT terá de ser convertido de string para inteiro ao ser usado. Sugere-se usar parseInt(process.env.PORT || "").

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 caminho **/pessoa**. A seguir tem-se exemplos de uso das rotas.

Estrutura atual do projeto:



#### HTTP POST

POST	http://localhost:3003/pessoa		
Query	Headers <sup>2</sup>	Auth	Body <sup>1</sup>
JSON	XML	Text	Form
<pre>1  { 2    "name": "Maria" 3  }</pre>			
Status: 200 OK		Size: 8 Bytes	
Response	Headers <sup>6</sup>	Cookies	
<pre>1  { 2    "id": 1 3  }</pre>			

#### HTTP GET

GET	http://localhost:3003/pessoa
Status: 200 OK	Size: 73 Bytes
Response	Headers <sup>6</sup>
1	[
2	{
3	"id": 3,
4	"name": "Clara"
5	},
6	{
7	"id": 2,
8	"name": "José"
9	},
10	{
11	"id": 1,
12	"name": "Maria"
13	}
14	]

#### HTTP PUT

PUT	http://localhost:3003/pessoa		
Query	Headers <sup>2</sup>	Auth	Body <sup>1</sup>
JSON	XML	Text	Form
1	{		
2	"name": "Paulo",		
3	"id": 2		
4	}		
Status: 200 OK		Size: 14 Bytes	
Response	Headers <sup>6</sup>		
1	{		
2	"rowcount": 1		
3	}		

#### HTTP DELETE

DELETE	http://localhost:3003/pessoa		
Query	Headers <sup>2</sup>	Auth	<u>Body<sup>1</sup></u>
<u>JSON</u>	XML	Text	Form
1	{		
2	"id": 1		
3	}		
Status: 200 OK    Size: 14 Bytes			
<u>Response</u>	Headers <sup>6</sup>	Cookies	
1	{		
2	"rowcount": 1		
3	}		

**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

POST	http://localhost:3003/carro		
Query	Headers <sup>2</sup>	Auth	Body <sup>1</sup>
JSON	XML	Text	Form
<pre>1 { 2   "model": "Corsa" 3 }</pre>			
Status: 200 OK		Size: 8 Bytes	
Response	Headers <sup>6</sup>		
<pre>1 { 2   "id": 1 3 }</pre>			

#### HTTP GET

GET	http://localhost:3003/carro		
Query	Headers <sup>2</sup>	Auth	Body <sup>1</sup>
JSON	XML	Text	Form
<pre>1  { 2    "model": "Fusca" 3  }</pre>			
Status: 200 OK Size: 74 Bytes			
Response	Headers <sup>6</sup>		
<pre>1  [ 2    { 3      "id": 1, 4      "model": "Corsa" 5    }, 6    { 7      "id": 3, 8      "model": "Fusca" 9    }, 10   { 11     "id": 2, 12     "model": "Uno" 13   } 14 ]</pre>			

#### HTTP PUT

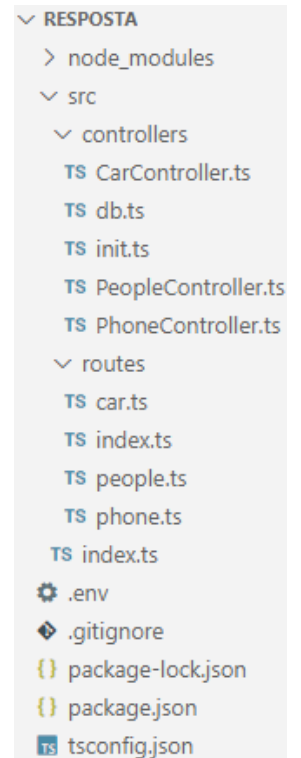
PUT	http://localhost:3003/carro		
Query	Headers <sup>2</sup>	Auth	Body <sup>1</sup>
JSON	XML	Text	Form
<pre>1  { 2    "model": "Onix", 3    "id": 2 4  }</pre>			
Status: 200 OK Size: 14 Bytes			
Response	Headers <sup>6</sup>		
<pre>1  { 2    "rowcount": 1 3  }</pre>			

#### HTTP DELETE

DELETE	http://localhost:3003/carro		
Query	Headers <sup>2</sup>	Auth	Body <sup>1</sup>
JSON	XML	Text	Form
<pre>1 { 2   "id": 1 3 }</pre>			
Status: 200 OK   Size: 14 Bytes			
Response	Headers <sup>6</sup>		
<pre>1 { 2   "rowcount": 1 3 }</pre>			

**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

POST	http://localhost:3003/telefone		
Query	Headers <sup>2</sup>	Auth	Body <sup>1</sup>
JSON	XML	Text	Form
<pre>1 { 2   "number": "1239456789", 3   "idpeople": 2 4 }</pre>			
Status: 200 OK		Size: 8 Bytes	
Response	Headers <sup>6</sup>	Cookies	
<pre>1 { 2   "id": 1 3 }</pre>			

#### HTTP GET

GET	http://localhost:3003/telefone		
Query	Headers <sup>2</sup>	Auth	Body <sup>1</sup>
JSON	XML	Text	Form
<pre>1  { 2    "idpeople": 2 3  }</pre>			
Status: 200 OK    Size: 64 Bytes			
Response	Headers <sup>6</sup>		
<pre>1  [ 2    { 3      "id": 3, 4      "number": "12911223344" 5    }, 6    { 7      "id": 1, 8      "number": "1239456789" 9    } 10 ]</pre>			

#### HTTP PUT

PUT	http://localhost:3003/telefone		
Query	Headers <sup>2</sup>	Auth	Body <sup>1</sup>
JSON	XML	Text	Form
<pre>1 { 2   "id": 3, 3   "number": "12911220000" 4 }</pre>			
Status: 200 OK		Size: 14 Bytes	
Response	Headers <sup>6</sup>		
<pre>1 { 2   "rowcount": 1 3 }</pre>			

#### HTTP DELETE

DELETE	http://localhost:3003/telefone		
Query	Headers <sup>2</sup>	Auth	Body <sup>1</sup>
JSON	XML	Text	Form
<pre>1 { 2   "id": 1 3 }</pre>			
Status: 200 OK Size: 14 Bytes			
Response	Headers <sup>6</sup>		
<pre>1 { 2   "rowcount": 1 3 }</pre>			



**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:

