Objetivos:

I. Criar o banco de dados;

II. Criar o servidor;

III. Criar o arquivo de conexão com o PostgreSQL;

IV. CRUD – Create, Read, Update e Delete;

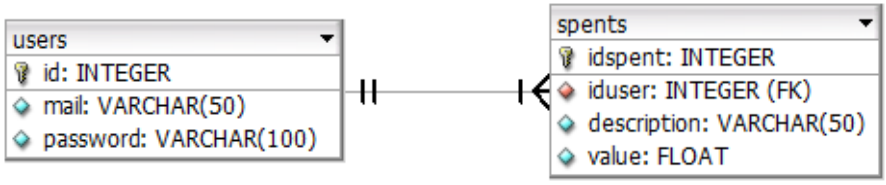
V. Processar as requisições HTTP e fazer o CRUD.

I. Criar o banco de dados

Acesse o pgAdmin e execute os seguintes passos para criar um banco de dados e as tabelas no SGBD PostgreSQL:

Passo 1 – Crie um banco de nome `bdaula` ou qualquer outro nome de preferência;

Passo 2 – Use os comandos a seguir para criar as tabelas `users` e `spents` (gastos) no `bdaula`:




```
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
);
```

II. Criar o servidor

Siga os passos a seguir para configurar o servidor Node.js com Express:

Passo 1 – Crie uma pasta de nome `server` ou qualquer outro nome no local de sua preferência do computador;

Passo 2 – Use o comando a seguir para criar o arquivo de configuração de um projeto Node.js:

```
npm init -y
```

Passo 3 – Use o comando a seguir para instalar as dependências:

```
npm i express dotenv cors pg
```

O pacote `pg` provê o código que facilita o acesso ao SGBD PostgreSQL.

Passo 4 – Crie o arquivo `.env` na raiz do projeto e coloque a variável:

```
PORT = 3020
```

Passo 5 – Crie a pasta `src` e o arquivo `index.js`. Por enquanto, deixe o arquivo vazio;

Passo 6 – Inclua na propriedade `scripts` do arquivo `package.json` a propriedade para rodarmos o projeto:

```
"scripts": {
  "start": "node ./src"
},
```

Passo 7 – Use o comando `npm run start` para executar o projeto no terminal do VS Code.

III. Criar o arquivo de conexão com o PostgreSQL

Para estabelecer a conexão com o BD PostgreSQL, crie um arquivo chamado `db.js` dentro da pasta `src` do seu projeto `server`. Esse arquivo será responsável por configurar e exportar a conexão com o banco , utilizando o módulo pg.

Conteúdo do arquivo `db.js`:

```
const { Pool } = require("pg");

const pool = new Pool({
  host: "localhost",
  user: "postgres", // Altere conforme o seu usuário
  password: "123", // Altere conforme a sua senha
  database: "bdaula", // Nome do banco criado no pgAdmin
  port: 5432, // Porta padrão do PostgreSQL
});

module.exports = pool;
```

Um *pool de conexões* é uma técnica utilizada para gerenciar e reutilizar conexões com o BD, evitando a criação e destruição constante dessas conexões. Isso é essencial para aplicações que realizam muitas operações no banco, especialmente em ambientes com múltiplos acessos simultâneos.

Conceitos principais:

- Conexões com o BD: cada vez que o sistema precisa consultar ou atualizar dados, ele precisa estabelecer uma conexão com o banco. Criar e encerrar conexões repetidamente gera sobrecarga e reduz o desempenho;
- Pooling de conexões: o pool mantém um conjunto de conexões abertas e disponíveis para uso. Em vez de criar uma nova conexão toda vez, a aplicação reutiliza uma já existente;

- Reutilização eficiente: ao terminar uma operação no banco, a conexão não é fechada, mas retornada ao pool para ser utilizada novamente em outras requisições.

Benefícios do uso de *connection pool*:

- Melhor desempenho: reduz o tempo necessário para iniciar conexões com o banco;
- Menor consumo de recursos: evita o custo computacional de abrir/fechar conexões com frequência;
- Controle de carga: limita a quantidade de conexões simultâneas, evitando sobrecarga no BD.

Essa abordagem é recomendada para qualquer sistema que realize diversas interações com o banco, pois melhora significativamente a escalabilidade e a eficiência da aplicação.

IV. CRUD – Create, Read, Update e Delete

As operações de CRUD permitem criar, consultar, atualizar e remover registros em uma tabela do BD. No exemplo a seguir, utilizamos o método `query` do objeto `Pool` para executar comandos SQL no banco PostgreSQL.

Inserir registro na tabela (create)

O seguinte código insere um novo registro na tabela `users`:

```
const result = await db.query(
        "INSERT INTO users (mail, password) VALUES ($1, $2) RETURNING *",
        ["ana@teste.com","123"]
      );
console.log(result.rows[0]);
```

Explicação:

- O método `query` recebe dois argumentos:
  1. Uma string com o comando SQL, utilizando parâmetros posicionais `$1`, `$2` etc.;
  2. Um array com os valores a serem utilizados nos respectivos parâmetros.
- O termo `RETURNING *` instrui o banco a retornar todos os campos do registro inserido;
- A resposta da consulta estará na propriedade `rows`, que é um array. O primeiro elemento pode ser acessado com `result.rows[0]`.

O uso de promise para acessar o BD:

Especialmente ao lidar com operações assíncronas como acesso a BD, utilizamos os conceitos de Promises e async/await para garantir que a execução do código ocorra de maneira ordenada e previsível.

`db.query(...)`: é uma função assíncrona que envia um comando SQL ao PostgreSQL. Como essa operação envolve acesso ao disco e rede, ela pode levar algum tempo para ser concluída.

`await`: a palavra-chave await instrui o JavaScript a aguardar a conclusão da Promise retornada por db.query(...) antes de continuar a execução do código. Isso significa que a linha `console.log(result.rows[0])` só será executada após a conclusão da inserção no BD e o retorno dos dados inseridos. Sem o await, o JavaScript continuaria executando o restante do código imediatamente,

sem esperar a resposta da operação de inserção. Isso poderia gerar erros, como tentar acessar `result.rows[0]` antes que `result` estivesse disponível.

Listar registros da tabela (read)

O código a seguir recupera todos os registros da tabela `users`:

```
const result = await db.query("SELECT * FROM users");
console.log(result.rows);
```

Explicação:

- O comando `SELECT * FROM users` retorna todos os registros da tabela;
- A propriedade `rows` conterá um array, onde cada elemento representa um registro em formato de objeto JSON.

Atualizar registro da tabela (update)

O código a seguir atualiza na tabela `users` o registro que possui id=3:

```
const result = await db.query(
        "UPDATE users SET mail = $1, password = $2 WHERE id = $3 RETURNING *",
        ["maria@teste.com","123", 3]
);
if (result.rowCount === 0) {
  console.log({ erro: "Usuário não encontrado" });
} else {
  console.log(result.rows[0]);
}
```

Explicação:

- O comando `UPDATE` modifica os campos `mail` e `password` do usuário com o `id` informado;
- O método query utiliza os parâmetros passados no array `["maria@teste.com","123", 3]`;
- A propriedade `rowCount` indica quantas linhas foram afetadas pelo comando SQL;
- O `RETURNING *` retorna os dados atualizados do registro.

Excluir registro da tabela (delete)

O código a seguir exclui da tabela `users` o registro que possui id=3:

```
const result = await db.query(
  "DELETE FROM users WHERE id = $1 RETURNING *",
  [3]
);
if (result.rowCount === 0) {
  res.json({ erro: "Usuário não encontrado" });
} else {
  console.log(result.rows[0]);
}
```

Explicação:

- O comando DELETE remove o registro correspondente ao id informado;
- Caso nenhum registro seja encontrado, rowCount será igual a zero;
- Caso contrário, os dados do registro excluído são retornados na propriedade rows.

Segurança: uso de parâmetros

Todas as consultas utilizam parametrização com símbolos como $1, $2 etc. Essa prática evita ataques de injeção de SQL, pois os valores são tratados como parâmetros, e não como parte do comando SQL.

V. Processar as requisições HTTP e fazer o CRUD

O código a seguir define as rotas, trata os dados enviados e interage com o BD PostgreSQL por meio do pool de conexões criado no arquivo src/db.js.

Coloque o código a seguir no arquivo src/index.js. do seu projeto server:

```
// Importa os módulos necessários
const express = require("express");
const dotenv = require("dotenv");
const cors = require("cors");
const db = require("./db"); // Importa a conexão com o BD PostgreSQL

// Carrega as variáveis de ambiente do arquivo .env
dotenv.config();

// Instancia a aplicação Express
const app = express();

// Define a porta que será usada pelo servidor
const PORT = process.env.PORT || 3000;

// Middleware para permitir o uso de JSON no corpo das requisições
app.use(express.json());

// Middleware para permitir requisições de diferentes origens (CORS)
app.use(cors());

// Inicia o servidor e escuta na porta definida
app.listen(PORT, function () {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});

// Rota POST para criar um novo usuário
app.post("/users", async function(req, res) {
  const { mail, password } = req.body;

  // Verifica se os campos obrigatórios foram enviados
```

```
  if (!mail || !password) {
    res.json({ erro: "Campos obrigatórios não preenchidos" });
  } else {
    try {
      // Insere os dados na tabela e retorna o registro criado
      const result = await db.query(
        "INSERT INTO users (mail, password) VALUES ($1, $2) RETURNING *",
        [mail, password]
      );
      res.json(result.rows[0]);
    } catch (error) {
      console.error("Erro ao inserir usuário:", error.message);
      res.json({ erro: "Erro interno do servidor" });
    }
  }
});

// Rota GET para listar todos os usuários
app.get("/users", async function(req, res) {
  try {
    const result = await db.query("SELECT * FROM users");
    res.json(result.rows); // Retorna todos os registros da tabela users
  } catch (error) {
    console.error("Erro ao buscar usuários:", error.message);
    res.json({ erro: "Erro interno do servidor" });
  }
});

// Rota PUT para atualizar os dados de um usuário específico
app.put("/users/:id", async function(req, res) {
  const { id } = req.params; // Obtém o id da URL
  const { mail, password } = req.body; // Obtém os dados enviados no corpo da requisição

  try {
    // Atualiza os dados do usuário com base no id informado
    const result = await db.query(
      "UPDATE users SET mail = $1, password = $2 WHERE id = $3 RETURNING *",
      [mail, password, id]
    );

    // Verifica se algum registro foi atualizado
    if (result.rowCount === 0) {
      res.json({ erro: "Usuário não encontrado" });
    } else {
      res.json(result.rows[0]);
    }
  } catch (error) {
    console.error("Erro ao atualizar usuário:", error.message);
    res.json({ erro: "Erro interno do servidor" });
```

```
  }
});

// Rota DELETE para excluir um usuário específico
app.delete("/users/:id", async function(req, res) {
  const { id } = req.params;

  try {
    // Exclui o usuário com o id informado
    const result = await db.query(
      "DELETE FROM users WHERE id = $1 RETURNING *",
      [id]
    );

    // Verifica se o usuário foi encontrado e excluído
    if (result.rowCount === 0) {
      res.json({ erro: "Usuário não encontrado" });
    } else {
      console.log(result.rows[0]); // Exibe o usuário excluído no terminal
      res.json({ message: "Usuário excluído com sucesso" });
    }
  } catch (error) {
    console.error("Erro ao excluir usuário:", error.message);
    res.json({ erro: "Erro interno do servidor" });
  }
});
```

Explicações complementares

- O uso de `await` só é permitido dentro de funções assíncronas, ou seja, que estejam declaradas com a palavra-chave `async`;
- Na estrutura de rotas RESTful, cada rota criada corresponde a uma operação específica sobre os dados da tabela `users`:
  - POST /users: cria um novo usuário;
  - GET /users: lista todos os usuários cadastrados;
  - PUT /users/:id: atualiza os dados de um usuário específico;
  - DELETE /users/:id: remove um usuário com base no seu id.
- O uso da parametrização com `$1`, `$2` etc., em todas as consultas SQL é uma medida essencial contra injeções de SQL.

Comandos para testar as rotas

- Listar os registros da tabela `users`:

```
curl -X GET http://localhost:3020/users
```

- Inserir um registro na tabela `users`:

```
curl -X POST http://localhost:3020/users -H "Content-Type: application/json" -d "{\"mail\": \"ana@teste.com\", \"password\": \"123\"}"
```

Observação: copie e cole esse comando em um bloco de notas, pois o comando precisa estar em uma única linha para ser usado no prompt de comandos do Windows.

- Atualizar o registro que possui `id=1` na tabela `users`:

```
curl -X PUT http://localhost:3020/users/1 -H "Content-Type: application/json" -d "{\"mail\": \"maria@teste.com\", \"password\": \"abc\"}"
```

- Excluir o registro que possui `id=1` na tabela `users`:

```
curl -X DELETE http://localhost:3020/users/1
```

Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

Acesso ao BD - https://youtu.be/L6afYQJO7PE

Exercícios - https://youtu.be/Ps7MiBrz-fI

Exercício 1 – Implemente, no arquivo `src/index.js`, um endpoint HTTP do tipo POST para inserir um novo registro na tabela `spents`, associando-o a um usuário existente por meio do campo `iduser`. O corpo da requisição (body) deverá conter os dados necessários para o cadastro da despesa. Certifique-se de validar os campos obrigatórios e utilizar a função query com parametrização para evitar falhas de segurança como a injeção de SQL.

Comando para testar:

```
curl -X POST http://localhost:3020/spents -H "Content-Type: application/json" -d "{\"iduser\": \"1\", \"description\": \"mercado\", \"value\": \"54.32\"}"
```

Exercício 2 – Implemente, no arquivo `src/index.js`, um endpoint HTTP do tipo GET para listar todos os registros da tabela `spents` associados a um determinado usuário. O identificador do usuário (`iduser`) deve ser fornecido como parâmetro na URL da requisição. O endpoint deve retornar todas as despesas vinculadas ao usuário ordenadas pela descrição.

Comando para listar todas as despesas do usuário que possui `id=1`:

```
curl -X GET http://localhost:3020/spents/user/1
```

Exercício 3 – Implemente, no arquivo `src/index.js`, um endpoint HTTP do tipo PUT para atualizar um registro existente na tabela `spents`. O identificador da despesa (`id`) deve ser fornecido como parâmetro na URL da requisição. O corpo da requisição (body) deverá conter os dados a serem atualizados: `iduser`, `description` e `value`.

A operação de atualização deve ser feita utilizando a função query do módulo pg, com uso de parametrização para evitar falhas de segurança como a injeção de SQL. O endpoint deve retornar o registro atualizado ou uma mensagem de erro, caso o registro não seja encontrado.

Comando para atualizar a despesa que possui `id=1`:

```
curl -X PUT http://localhost:3020/spents/1 -H "Content-Type: application/json" -d "{\"iduser\": \"1\", \"description\": \"supermercado\", \"value\": \"154.32\"}"
```

Exercício 4 – Implemente, no arquivo `src/index.js`, um endpoint HTTP do tipo DELETE para remover um registro existente da tabela `spents`. O identificador da despesa (`id`) deve ser fornecido como parâmetro na URL da requisição. A operação de exclusão deve ser realizada utilizando a função query com uso de parametrização para garantir a segurança contra injeção de SQL. O endpoint deve retornar uma mensagem informando o sucesso da operação ou um erro, caso a despesa não seja encontrada.

```
curl -X DELETE http://localhost:3020/spents/1
```