📦 Criação de Banco de Dados e Tabelas com `pg` (PostgreSQL + Node.js)

🧠 Conceito

Ao usar a biblioteca `pg` no Node.js, é possível realizar duas etapas importantes:

1. **Conectar ao banco `postgres`** (banco padrão do PostgreSQL) para verificar e criar o banco de dados da aplicação.
2. **Conectar ao banco recém-criado** para criar suas tabelas.

🛠️ Etapas do Processo

1. Conexão Inicial ao Banco `postgres`

Antes de criar qualquer banco de dados, é necessário se conectar ao PostgreSQL por meio do banco padrão `postgres`.

```
const { Client } = require('pg');

const entradaPostgre = {
  user: process.env.DB_USER,
  host: process.env.DB_HOST,
  database: 'postgres', // banco padrão do PostgreSQL
  password: process.env.DB_PASSWORD,
  port: process.env.DB_PORT,
};

const client = new Client(entradaPostgre);
await client.connect();

// Verifica se o banco já existe
const res = await client.query(
  `SELECT 1 FROM pg_database WHERE datname = $1`,
  [nomeDoBanco]
);

if (res.rowCount === 0) {
  await client.query(`CREATE DATABASE ${nomeDoBanco}`);
  console.log(`Banco de dados '${nomeDoBanco}' criado com sucesso.`);
} else {
  console.log(`Banco de dados '${nomeDoBanco}' já existe.`);
}

await client.end();
```

---

2. Conexão ao Novo Banco de Dados

Depois que o banco for criado, agora sim podemos nos conectar diretamente a ele para criar as tabelas necessárias.

```
const inicializacaoDB = {
  user: process.env.DB_USER,
  host: process.env.DB_HOST,
  database: process.env.DB_NAME, // o banco criado anteriormente
  password: process.env.DB_PASSWORD,
  port: process.env.DB_PORT,
};

const client = new Client(inicializacaoDB);
await client.connect();

// Verifica se a tabela 'cliente' existe
const tabCliente = await client.query(`
  SELECT 1 FROM information_schema.tables
  WHERE table_schema = 'public' AND table_name = 'cliente';
`);

if (tabCliente.rowCount === 0) {
  await client.query(`
    CREATE TABLE cliente (
      id SERIAL PRIMARY KEY,
      nome VARCHAR(100) NOT NULL,
      email VARCHAR(100) UNIQUE NOT NULL,
      senha VARCHAR(100) NOT NULL,
      ativo BOOLEAN NOT NULL DEFAULT true,
      criado_em TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );
  `);
  console.log('Tabela "cliente" criada com sucesso.');
} else {
  console.log('Tabela "cliente" já existe.');
}

await client.end();
```

☑ Conclusão

Essa abordagem garante que:

- Você só cria o banco se ele ainda **não existir**.
- Você só cria a tabela se ela ainda **não existir**.
- As conexões são bem definidas e fechadas corretamente.

Entendendo JWT no meu projeto

Recentemente, consegui compreender o funcionamento do **JWT (JSON Web Token)**. Ele funciona como uma **chave de acesso**, garantindo que apenas usuários autorizados possam acessar determinadas rotas do sistema — uma **forma de segurança importante em APIs**.

No meu projeto, implementei o JWT e, apesar de ter dado muito trabalho no começo, finalmente consegui entender como funciona.

Código para autenticação com JWT

```
const jwt = require('jsonwebtoken');
require('dotenv').config(); // Carrega as variáveis de ambiente do arquivo .env

function autenticarToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1]; // Espera formato
"Bearer TOKEN"

  if (!token) {
    return res.status(401).json({ erro: 'Token não fornecido' });
  }

  jwt.verify(token, process.env.JWT_SECRET, (err, usuario) => {
    if (err) return res.status(403).json({ erro: 'Token inválido' });

    req.usuario = usuario; // Salva os dados do usuário no request
    next(); // Prossegue para a próxima função/middleware
  });
}

module.exports = autenticarToken;
```

Dificuldade que enfrentei

No início, tive problemas ao tentar acessar o `id` do usuário autenticado. Estava fazendo:

```
req.user.id // errado
```

Quando na verdade, o correto era:

```
req.usuario.id // certo (conforme definido no middleware)
```

Esse pequeno detalhe fez bastante diferença e me ajudou a pegar corretamente os dados do usuário logado.

Dúvidas que ainda tenho

1. **Como o JWT realmente sabe se um token é válido?**
    - Ele decodifica o token usando a chave secreta (`process.env.JWT_SECRET`) e verifica se ele está assinado corretamente e se ainda está dentro do tempo de expiração.
2. **É seguro armazenar dados sensíveis dentro do token?**
    - Não. O token pode ser decodificado facilmente. Mesmo que ele esteja assinado e não possa ser modificado sem invalidar a assinatura, **os dados não são criptografados**. Guarde apenas informações necessárias e genéricas, como `id` e `nome`.
3. **O que acontece se alguém modificar o token?**
    - A assinatura do JWT será invalidada, e a verificação com `jwt.verify` falhará, retornando o erro `Token inválido`.
4. **Qual é a vantagem de usar JWT em vez de sessões (cookies)?**
    - JWT permite autenticação **sem estado (stateless)**. Isso significa que você não precisa armazenar informações do usuário no servidor, o que facilita escalabilidade. Ideal para APIs RESTful.

Usando o email do usuário para buscar seus dados após o login

Uma coisa que eu entendi é que posso usar o **email do usuário** como chave para localizar seu **ID** ou outros dados no backend. Isso facilita muito, especialmente depois que o login é realizado.

Exemplo: Salvando email e token após login (`login.pug`)

```
btnEntrar.addEventListener('click', async (event) => {
  event.preventDefault();

  const email = document.getElementById('email').value;
  const senha = document.getElementById('senha').value;

  try {
    const response = await fetch('/cliente/login', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
```

```
      },
      body: JSON.stringify({ email, senha })
    });

    const data = await response.json();

    if (!response.ok) {
      alert(data.erro || 'Erro ao fazer login');
      return;
    }

    // Salva o token e o email no localStorage
    localStorage.setItem('token', data.token);
    localStorage.setItem('email', email);
    localStorage.setItem('id', data.id);

    alert('Login realizado com sucesso!');
    window.location.href = '/dashboard';

  } catch (err) {
    console.error(err);
    alert('Erro de conexão com o servidor');
  }
});
```

Neste exemplo:

- O `email` é obtido com `document.getElementById('email').value`.
- Após o login bem-sucedido, o email é salvo com `localStorage.setItem('email', email)`.

Exemplo: Buscando os dados do cliente com base no email

```
document.addEventListener('DOMContentLoaded', async () => {
  const token = localStorage.getItem('token');
  const email = localStorage.getItem('email');
  console.log("Token: ", token);
  console.log("Email: ", email);

  if (!token) {
    alert('Você precisa estar logado.');
    window.location.href = '/';
    return;
  }

  try {
    const response = await fetch(`cliente/dados/email/${email}`, {
      method: 'GET',
      headers: {
        'Authorization': `Bearer ${token}`
```

```
      }
    });

    const data = await response.json();

    if (!response.ok) {
      alert(data.erro || 'Erro ao buscar dados');
      return;
    }

    console.log('Cliente logado:', data);

    // Se for um array, pega o primeiro item
    const cliente = Array.isArray(data) ? data[0] : data;

    // Mostra o nome do cliente no HTML
    const pDadosCliente = document.getElementById('dados-cliente');
    pDadosCliente.textContent = `Bem-vindo, ${cliente.nome}`;

  } catch (err) {
    console.error(err);
    alert('Erro de conexão com o servidor');
  }
});
```

Conclusão

Com esse processo:

- O email é salvo no `localStorage` durante o login.
- É possível usar esse email em rotas GET para buscar os dados do usuário.
- Isso simplifica a personalização do frontend com os dados do cliente logado.
