

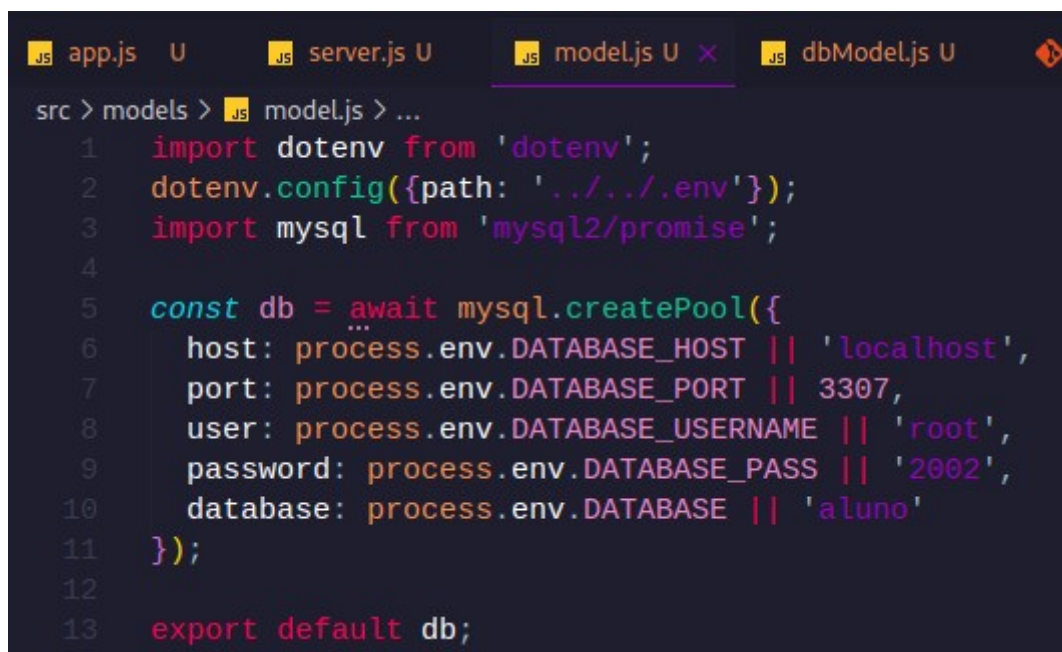
Criando conexão com node.js e SQL

Iremos criar uma conexão com uma database SQL.

Primeiramente, criamos o banco de dados usando sintaxe fora do node.js, na unha mesmo.

Então o node irá apenas trabalhar na comunicação e nós iremos trabalhar no models.

Criaremos dois arquivos model, um para configuração e outro para criar o modelo de como o database vai trabalhar com os dados



```
src > models > .js model.js > ...
1  import dotenv from 'dotenv';
2  dotenv.config({path: '../.env'});
3  import mysql from 'mysql2/promise';
4
5  const db = await mysql.createPool({
6    host: process.env.DATABASE_HOST || 'localhost',
7    port: process.env.DATABASE_PORT || 3307,
8    user: process.env.DATABASE_USERNAME || 'root',
9    password: process.env.DATABASE_PASS || '2002',
10   database: process.env.DATABASE || 'aluno'
11 });
12
13 export default db;
```

Vamos só entender algumas funções, após a importação da biblioteca 'mysql2/promise'

Usamos mysql2/promise porque ele permite trabalhar com async/await, tornando o código mais moderno e fácil de ler, em comparação com a abordagem baseada em callbacks.

E o usamos a função o .createPool – por alguns motivos:

Cria um conjunto de conexões (pool), gerenciando conexões automaticamente.

Cada requisição pega uma conexão disponível no pool e a libera após o uso.

Melhor performance para aplicações grandes.

Se for um projeto simples, invés do `.createPool()`, usamos o `.createConnection()`

Cria apenas uma conexão única com o banco.

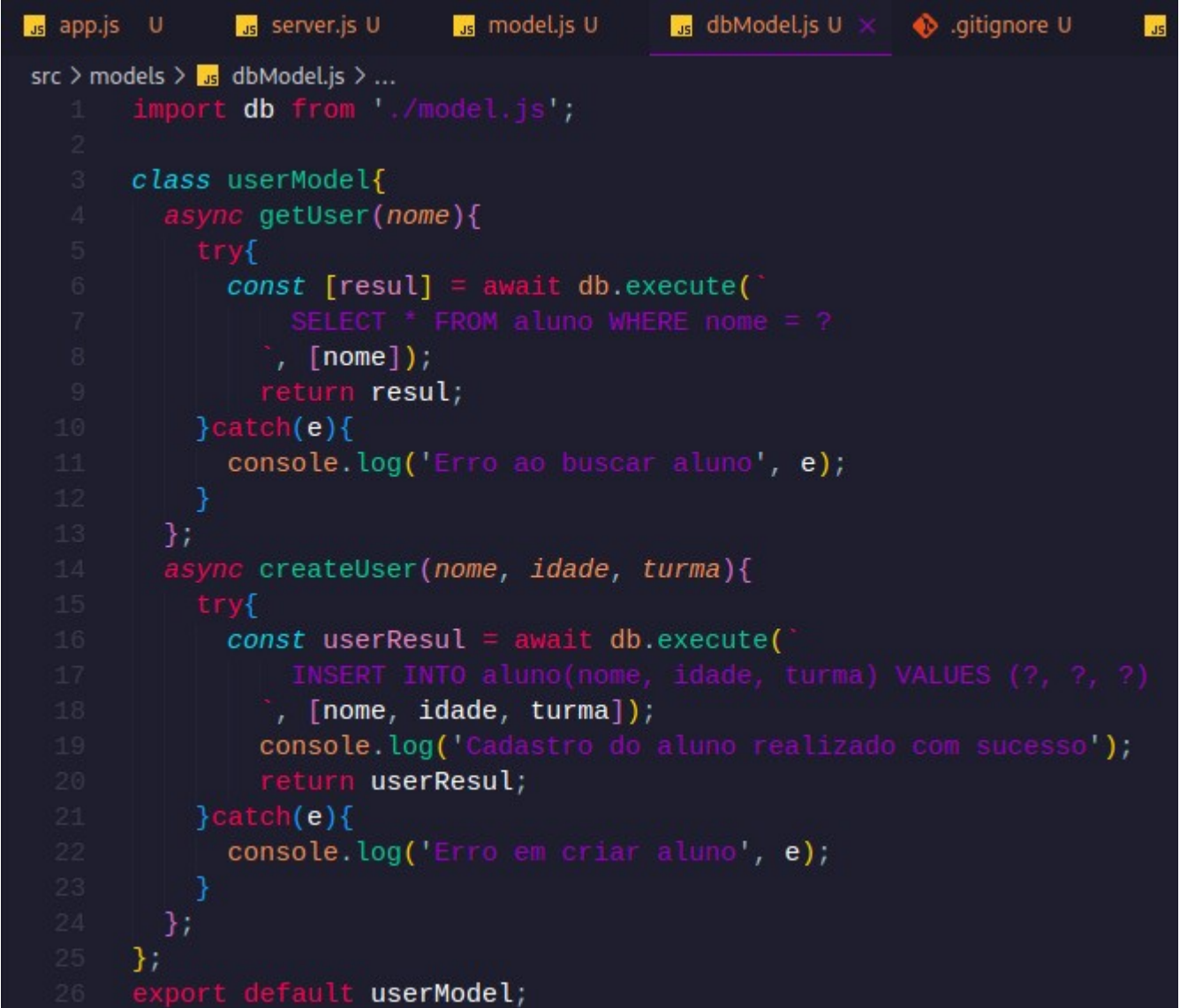
Toda consulta feita usa essa mesma conexão.

Se houver muitas requisições simultâneas, pode sobrecarregar o banco.

✓ Ideal para testes e pequenas aplicações.

⚠ Pode causar problemas de performance se houver muitas requisições simultâneas.

Agora iremos criar um model para fazer CRUD



```
src > models > dbModel.js > ...
1  import db from './model.js';
2
3  class userModel{
4    async getUser(nome){
5      try{
6        const [resul] = await db.execute(`
7          SELECT * FROM aluno WHERE nome = ?
8          `, [nome]);
9        return resul;
10     }catch(e){
11       console.log('Erro ao buscar aluno', e);
12     }
13   };
14   async createUser(nome, idade, turma){
15     try{
16       const userResul = await db.execute(`
17         INSERT INTO aluno(nome, idade, turma) VALUES (?, ?, ?)
18         `, [nome, idade, turma]);
19       console.log('Cadastro do aluno realizado com sucesso');
20       return userResul;
21     }catch(e){
22       console.log('Erro em criar aluno', e);
23     }
24   };
25 };
26 export default userModel;
```

Vamos em etapas. Basicamente .execute() permite que você possa executar comandos SQL.

Eu sei que você deve tá se perguntando o que é esse ponto de interrogação(?) no código. Os "?" nas queries SQL são placeholders para evitar SQL Injection e facilitar a inserção de dados.

👉 O primeiro argumento é a query SQL:

```
INSERT INTO users (name, email) VALUES (?, ?)
```

👉 O segundo argumento é um array de valores [name, email], que substituirá os ? na ordem correspondente.

OBS: o await db.execute() pode retornar diversas array's, por esse motivo fizemos atribuição via desestruturação para pegar apenas um array

Arquivo controll



```
src > controllers > dbControll.js > ...
1  import userModel from '../models/dbModel.js';
2
3  const aluno = async (req, res) => {
4    try {
5      const users = await new userModel().getUser('jean');
6      res.json(users);
7    } catch (error) {
8      console.log('Erro ao buscar aluno', error);
9      res.status(500).json({ error: 'Erro ao buscar aluno' });
10   }
11 };
12
13 export default aluno;
```

Como estamos trabalhando nesse projeto como API-REST Iremos enviar os dados em formato JSON para o cliente. Nesse caso consultamos o id

```

1  import { Router } from 'express';
2  import aluno from '../controllers/dbControll.js';
3
4  const router = new Router();
5
6  router.get('/', aluno);
7
8  export default router;

```

Fazendo o controle para que o 'home' seja o retorna do JSON

