Integração do Projeto com Banco de Dados no Node.js

Desenvolvimento Web 2 - Análise e Desenvolvimento de Sistemas

Objetivos de Aprendizagem

Papel do Banco de Dados

Compreender como o banco de dados funciona dentro de uma aplicação Node.js e sua importância na persistência de informações.

Configurar Conexão

Aprender a configurar e estabelecer uma conexão segura entre Node.js e MySQL utilizando o VS Code como ambiente de desenvolvimento.

Comandos SQL

Executar comandos SQL básicos diretamente do Node.js, incluindo consultas, inserções e atualizações de dados.

Integração com React

Integrar o banco de dados com o frontend React, criando uma aplicação full-stack completa e funcional.

Por Que Precisamos de um Banco de Dados?

Aplicações modernas precisam armazenar informações de forma permanente e organizada. Sem um banco de dados, todos os dados seriam perdidos quando o servidor fosse reiniciado. É como tentar guardar informações importantes apenas na memória - quando você desliga o computador, tudo desaparece.

O banco de dados resolve esse problema, oferecendo persistência, organização e acesso rápido aos dados. Ele é o coração de qualquer aplicação que precise lembrar de informações entre diferentes sessões de uso.

O Que o Banco de Dados Armazena?

Usuários

Informações de cadastro, credenciais de acesso, perfis e preferências dos usuários da aplicação.

Tarefas

Listas de afazeres, status de conclusão, datas de entrega e prioridades de cada tarefa.

Produtos

Catálogo completo com descrições, preços, quantidades em estoque e categorias de produtos.

Transações

Histórico de compras, pagamentos, pedidos e todas as operações realizadas na aplicação.

Node.js e a Necessidade do Banco de Dados

❌ Sem Banco de Dados

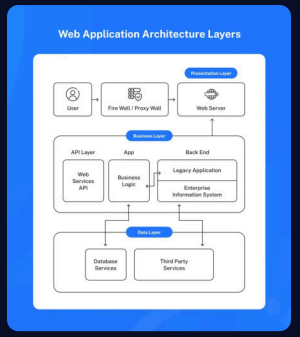
O Node.js, por si só, não possui mecanismo nativo para armazenar dados permanentemente. As variáveis e objetos criados durante a execução existem apenas na memória RAM.

- Dados perdidos ao reiniciar o servidor
- Impossível compartilhar dados entre requisições
- Sem histórico ou persistência
- Limitações de escalabilidade

✅ Com Banco de Dados

Ao integrar um banco de dados como o MySQL, transformamos nossa aplicação em um sistema robusto e confiável que mantém informações indefinidamente.

- Dados persistem entre reinicializações
- Múltiplos usuários acessam as mesmas informações
- Histórico completo de operações
- Aplicação escalável e profissional



Arquitetura da Aplicação Full-Stack

Frontend (React)

Interface do usuário que exibe e coleta informações

Backend (Node.js)

Servidor que processa requisições e se comunica com o banco

Banco de Dados (MySQL)

Armazena e organiza todos os dados da aplicação

Preparando o Ambiente

Antes de começar a integração, precisamos instalar as ferramentas necessárias para conectar nosso Node.js ao MySQL.

Pacotes Necessários para a Integração

Para conectar o Node.js ao MySQL, utilizaremos dois pacotes fundamentais que facilitam a comunicação entre nossa aplicação e o banco de dados.

Express

Framework web minimalista e flexível para Node.js que simplifica a criação de servidores e APIs. Ele gerencia rotas, requisições HTTP e middleware de forma elegante.

```
npm install express
```

O Express é fundamental para criar endpoints que o React irá consumir, tornando a comunicação entre frontend e backend muito mais simples e organizada.

MySQL2

Driver moderno e eficiente para conectar Node.js ao MySQL. Oferece suporte a Promises e async/await, tornando o código mais limpo e fácil de manter.

```
npm install mysql2
```

O MySQL2 é uma versão aprimorada do driver original, com melhor performance e suporte completo às funcionalidades modernas do JavaScript e MySQL.

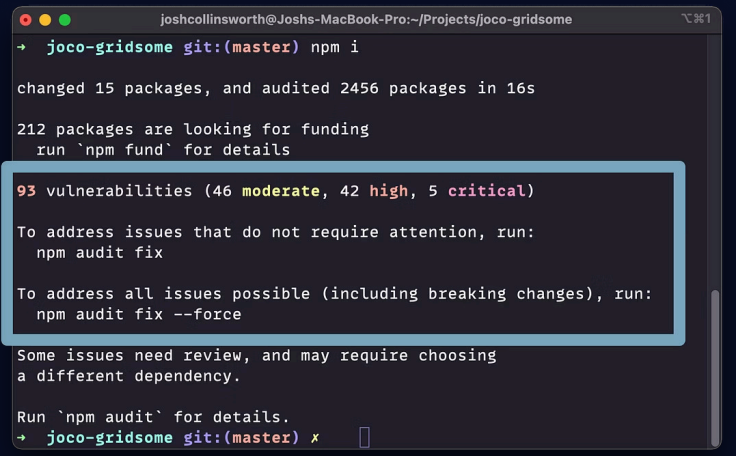
Comando de Instalação Completo

Execute o comando abaixo no terminal do VS Code para instalar ambos os pacotes de uma só vez:

```
npm install express mysql2
```

Este comando irá baixar os pacotes e suas dependências, adicionando-os ao arquivo package.json do seu projeto. Certifique-se de estar no diretório raiz do seu projeto Node.js antes de executar o comando.

Dica: Após a instalação, você verá uma pasta node_modules com todos os pacotes e um arquivo package-lock.json que garante versões consistentes.



Estrutura do Projeto

01

Criar pasta do projeto

Organize seu projeto em uma estrutura de pastas clara e bem definida

02

Inicializar npm

Execute npm init -y para criar o package.json com configurações padrão

03

Instalar dependências

Instale express e mysql2 conforme visto anteriormente

04

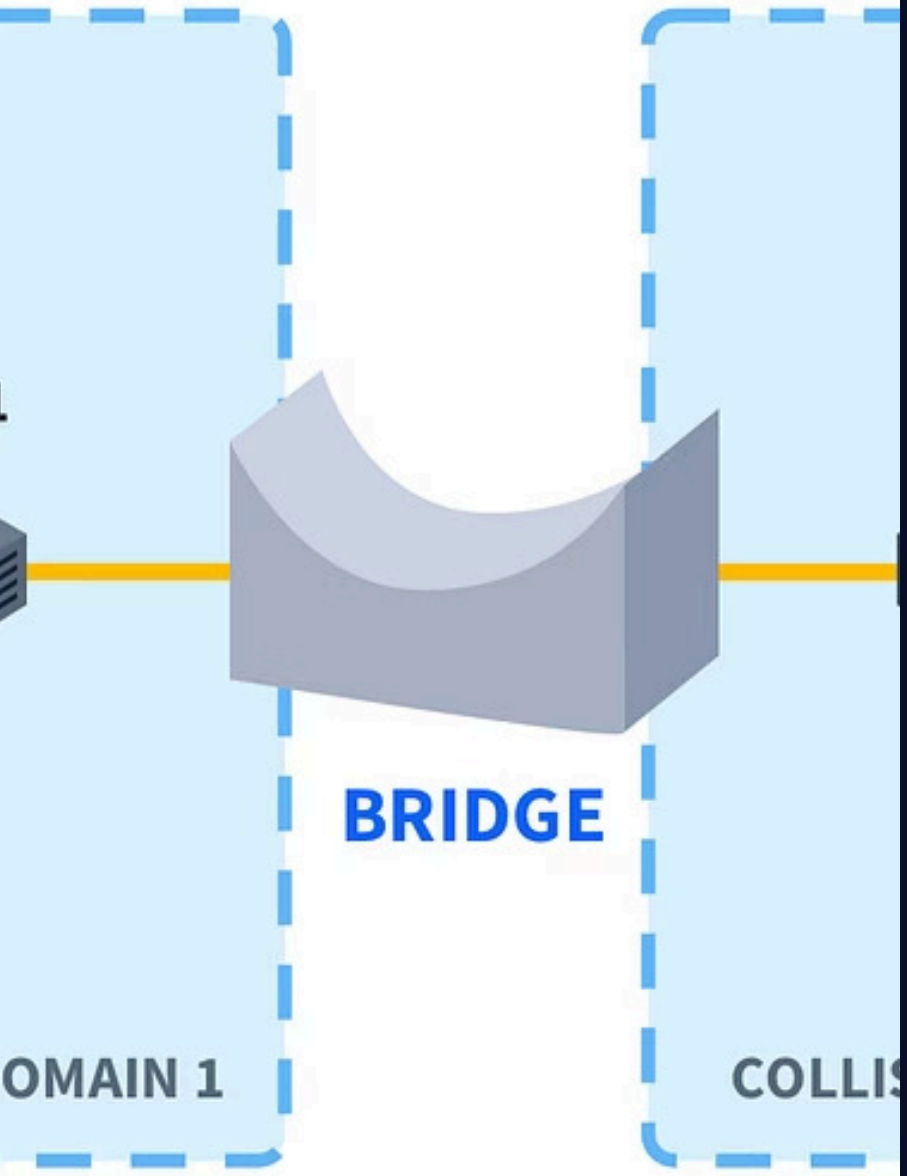
Criar arquivos principais

Crie server.js para o servidor e db.js para a conexão com banco

05

Configurar VS Code

Abra a pasta no VS Code e configure o ambiente de desenvolvimento



Conectando o Node.js ao Banco

Agora vamos estabelecer a conexão entre nossa aplicação Node.js e o banco de dados MySQL.

Criando o Arquivo de Conexão

O primeiro passo é criar um arquivo dedicado para gerenciar a conexão com o banco de dados. Vamos criar um arquivo chamado **db.js** que será responsável por estabelecer e exportar a conexão.

```
const mysql = require('mysql2');

const connection = mysql.createConnection({
 host: 'localhost',
 user: 'root',
 password: 'sua_senha',
 database: 'meu_banco'
});

connection.connect((err) => {
 if (err) {
 console.error('Erro ao conectar: ' + err.stack);
 return;
 }
 console.log('Conectado com ID ' + connection.threadId);
});

module.exports = connection;
```

Parâmetros de Conexão

host

Define o endereço do servidor MySQL. Use 'localhost' para desenvolvimento local ou o IP/domínio do servidor em produção.

user

Nome do usuário do MySQL. Por padrão é 'root', mas em produção recomenda-se criar um usuário específico com permissões limitadas.

password

Senha do usuário MySQL. Nunca compartilhe essa senha em repositórios públicos. Use variáveis de ambiente em produção.

database

Nome do banco de dados que você deseja usar. Certifique-se de que esse banco já foi criado no MySQL antes de conectar.

Tratamento de Erros na Conexão

Por Que Tratar Erros?

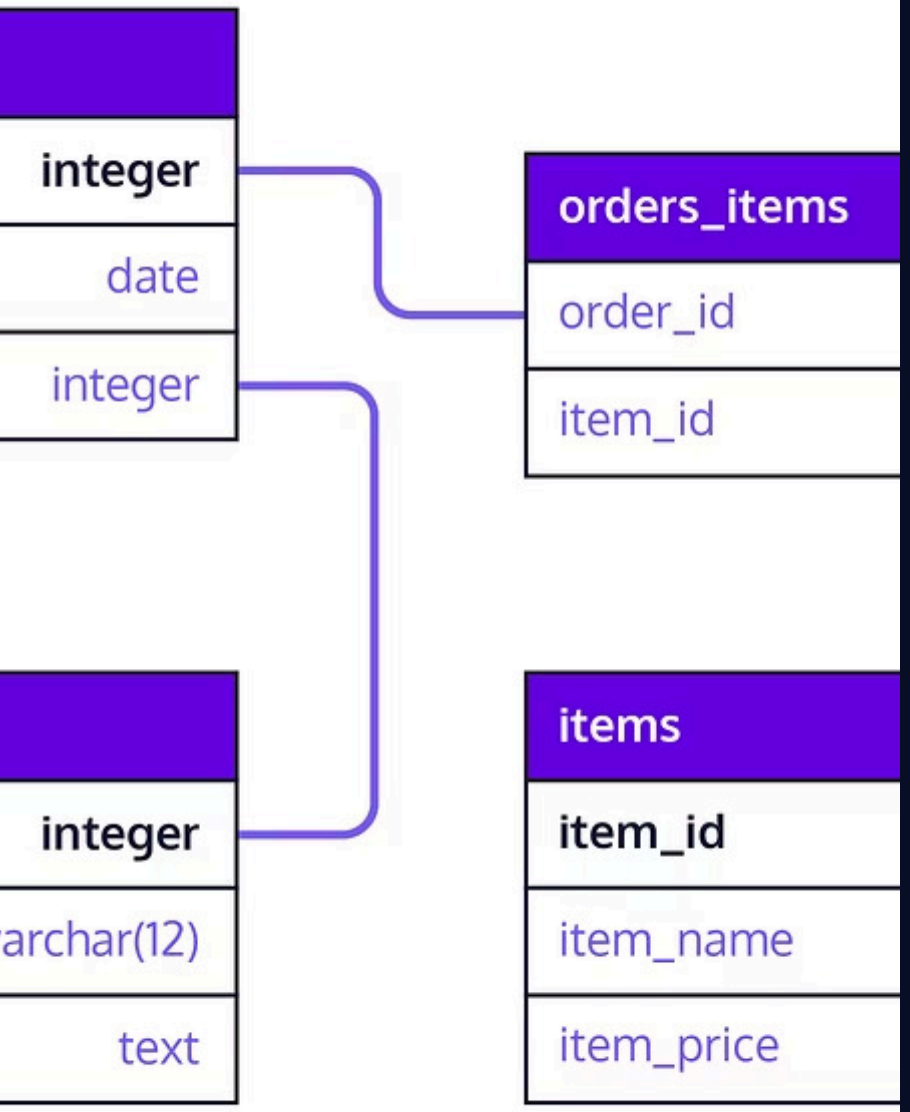
O tratamento de erros é essencial para identificar problemas de conexão rapidamente. Sem ele, sua aplicação pode falhar silenciosamente, dificultando a depuração.

Erros comuns incluem senha incorreta, banco inexistente, ou servidor MySQL desligado.

Callback de Erro

```
connection.connect((err) => {
  if (err) {
    console.error('Erro: ' + err.stack);
    return;
  }
  console.log('Conectado!');
});
```

O callback recebe um objeto de erro que contém detalhes sobre o que falhou.



Criando o Banco e a Tabela

Antes de executar consultas, precisamos criar a estrutura do nosso banco de dados.

Ferramentas para Gerenciar o MySQL

MySQL Workbench

Interface gráfica oficial do MySQL com recursos avançados de modelagem, design e administração de bancos.

DBeaver

Ferramenta universal gratuita que suporta diversos bancos de dados, incluindo MySQL, com interface intuitiva.

MySQL CLI

Linha de comando do MySQL para quem prefere trabalhar diretamente com comandos SQL no terminal.

Criando o Banco de Dados

O primeiro passo é criar o banco de dados que irá armazenar nossas tabelas. Execute o comando SQL abaixo em uma das ferramentas mencionadas:

```
CREATE DATABASE IF NOT EXISTS meu_banco
```

Este comando cria um banco chamado "meu_banco". O **IF NOT EXISTS** evita erros caso o banco já exista.

Selecionando o Banco de Dados

Após criar o banco, você precisa selecioná-lo antes de criar tabelas:

```
USE meu_banco;
```

Este comando informa ao MySQL que todos os comandos seguintes serão executados dentro deste banco específico. É como abrir uma pasta antes de criar arquivos dentro dela.

Criando uma Tabela de Usuários

Vamos criar uma tabela simples para armazenar informações de usuários. Esta tabela servirá como exemplo para nossas consultas no Node.js:

```
CREATE TABLE usuarios (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nome VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  senha VARCHAR(255) NOT NULL,
);
```

Esta estrutura cria uma tabela com campos essenciais: ID único gerado automaticamente, nome completo, email único para login, senha e registro automático da data de cadastro.

Entendendo os Tipos de Dados

INT AUTO_INCREMENT PRIMARY KEY

Cria um número inteiro que se incrementa automaticamente a cada novo registro. É a chave primária única de cada usuário.

VARCHAR(tamanho)

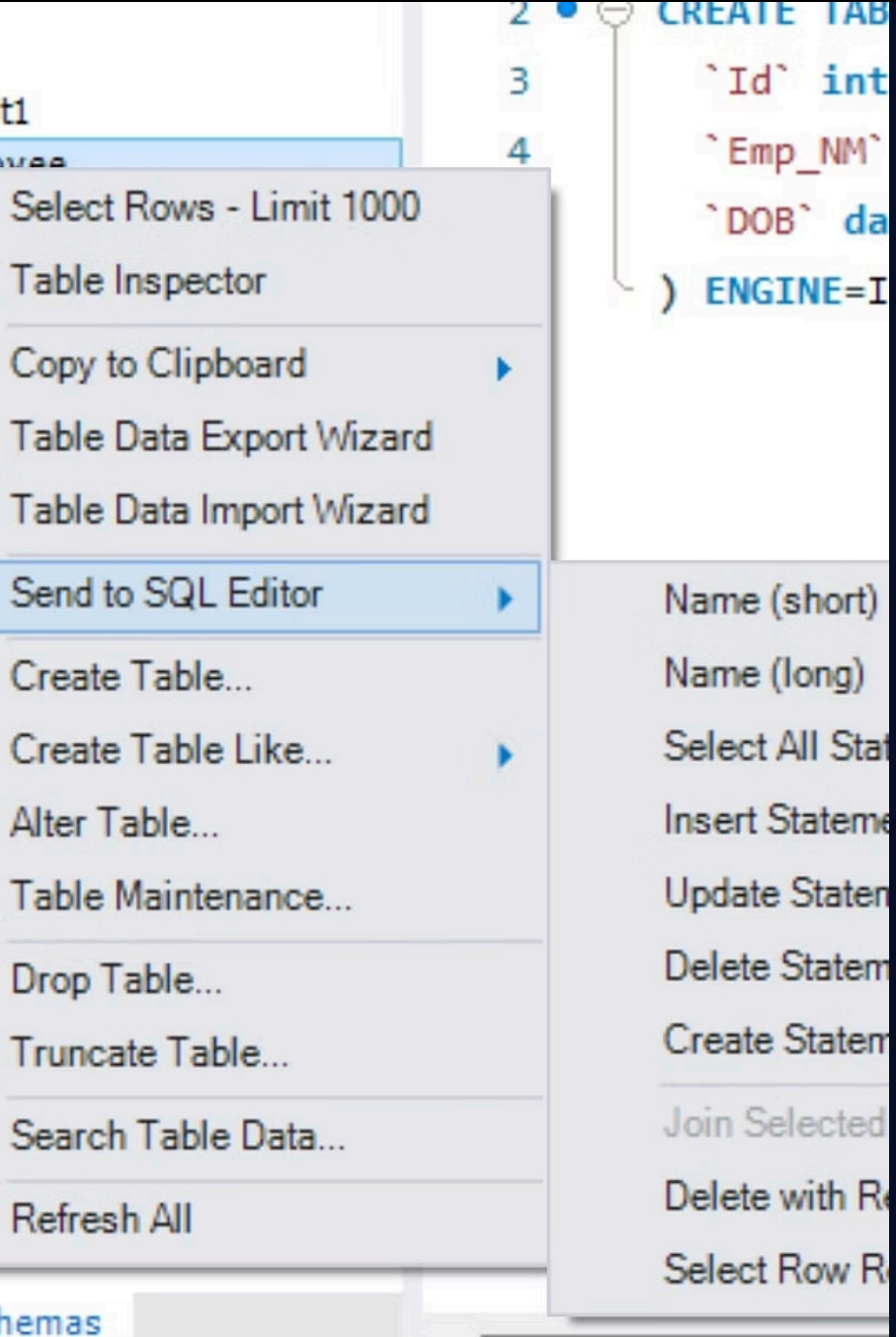
Armazena texto de tamanho variável até o limite especificado. Ideal para nomes, emails e textos curtos.

UNIQUE

Garante que não haverá valores duplicados naquela coluna. Perfeito para emails, evitando múltiplos cadastros com mesmo email.

NOT NULL

Torna o campo obrigatório, impedindo que seja salvo vazio. Essencial para dados críticos como nome e email.



Inserindo Dados de Teste

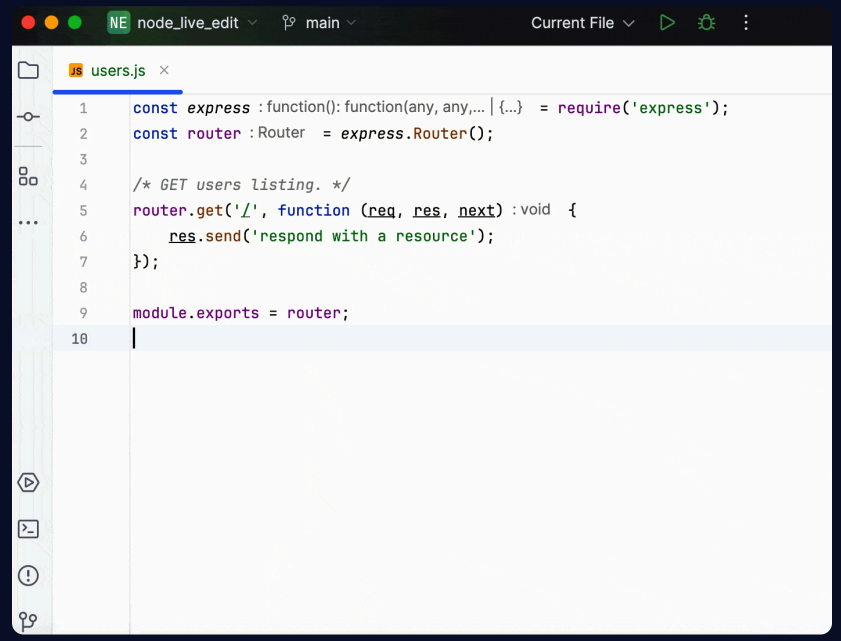
Para testarmos nossas consultas no Node.js, vamos inserir alguns registros de exemplo na tabela de usuários:

```
INSERT INTO usuarios (nome, email, senha) VALUES
('João Silva', 'joao@email.com', 'senha123'),
('Maria Santos', 'maria@email.com', 'senha456'),
('Pedro Oliveira', 'pedro@email.com', 'senha789');
```

Estes três registros servirão como dados de teste. Em uma aplicação real, as senhas devem sempre ser criptografadas antes de serem armazenadas.

Testando Consultas no Node.js

Agora que temos banco e dados, vamos executar consultas SQL diretamente do Node.js.



O Método query() do MySQL2

O método **query()** é a forma principal de executar comandos SQL no Node.js. Ele aceita uma string SQL e uma função de callback que recebe os resultados.

```
connection.query('SELECT * FROM usuarios', (error, results, fields) => {
  if (error) throw error;
  console.log(results);
});
```

Este método é assíncrono, o que significa que o código continua executando enquanto aguarda a resposta do banco de dados. O callback é chamado quando os resultados estão prontos.

Anatomia do Callback

error

Primeiro parâmetro contém informações sobre erros. Se for null, a consulta foi bem-sucedida.

fields

Terceiro parâmetro contém metadados sobre as colunas retornadas, como tipo e nome dos campos.

results

Segundo parâmetro contém os dados retornados pela consulta, geralmente um array de objetos.

Exemplo: SELECT - Buscar Todos os Usuários

Vamos criar um arquivo **testeSelect.js** para buscar todos os usuários do banco:

```
import connection from '../config/db.js';

// Função para listar todas as tarefas
const listarTarefas = (callback) => {
  const sql = 'SELECT * FROM tarefas';
  connection.query(sql, (err, results) => {
    if (err) {
    console.error('Erro na consulta:', error);
    return;
  }
    else callback(null, results);
  });
};
```

Execute com: node testeSelect.js

Exemplo: INSERT - Adicionar Novo Usuário

Para inserir um novo registro no banco de dados:

```
const criarUsuario = (nome, email, callback) => {

const novoUsuario = {
  nome: nome,
  email: email,
};



connection.query(
  'INSERT INTO usuarios SET ?',
  novoUsuario,
  (error, results) => {
    if (err) {
    console.error('Erro na consulta:', error);
    return;
  }
    else callback(null, results);
  });
};
```

O método **SET ?** permite passar um objeto JavaScript que será convertido automaticamente em SQL.

Exemplo: UPDATE - Atualizar Dados

Para modificar registros existentes no banco:

```
...

connection.query(
  'UPDATE usuarios SET nome = ? WHERE email = ?',
  [novoNome, emailUsuario],
  (error, results) => {
    if (err) {
    console.error('Erro na consulta:', error);
    return;
  }
    else callback(null, results);
  });
};
```

Use múltiplos placeholders **?** para passar vários valores de forma segura.

Exemplo: DELETE - Remover Registros

Para excluir registros do banco de dados:

```
...

connection.query(
  'DELETE FROM usuarios WHERE email = ?',
  [emailParaDeletar],
  (error, results) => {
    if (err) {
    console.error('Erro na consulta:', error);
    return;
  }
    else callback(null, results);
  });
};
```

Atenção: Sempre use WHERE em DELETE para evitar apagar toda a tabela acidentalmente!

Boas Práticas de Segurança

1

Nunca exponha credenciais

Use variáveis de ambiente (.env) para armazenar senhas e configurações sensíveis. Nunca commite arquivos .env no Git.

2

Sempre use placeholders (?)

Previna SQL Injection utilizando prepared statements com placeholders em vez de concatenar strings SQL.

3

Criptografe senhas

Use bcrypt para fazer hash de senhas antes de armazená-las. Nunca salve senhas em texto plano no banco.

4

Valide dados de entrada

Sempre valide e sanitize dados recebidos do frontend antes de inseri-los no banco de dados.

5

Configure CORS adequadamente

Use o middleware cors no Express para controlar quais origens podem acessar sua API.

6

Feche conexões adequadamente

Use connection pools em produção e sempre feche conexões para evitar vazamento de recursos.

Recapitulando

Hoje você aprendeu a integrar Node.js com MySQL, criando uma aplicação full-stack completa:

→ Compreendeu o papel fundamental do banco de dados

Entendeu por que aplicações precisam de persistência e como o banco resolve esse problema

→ Configurou a conexão Node.js + MySQL

Instalou pacotes, criou arquivo de conexão e estabeleceu comunicação com o banco

→ Executou comandos SQL do Node.js

Praticou SELECT, INSERT, UPDATE e DELETE usando o método query()

→ Integrou banco de dados com React

Criou API REST com Express e consumiu os dados no frontend React

Continue praticando e explorando as possibilidades dessa integração poderosa. Bons estudos!