Apostila do curso

Node.js: Desenvolvimento de APIs Web com Banco de Dados Relacional

Celso Y. Ishida

2025

Apresentação

Olá! Bem-vindo ao mundo do Nodejs para a programação back-end de aplicações web.

Vídeo: boas vindas

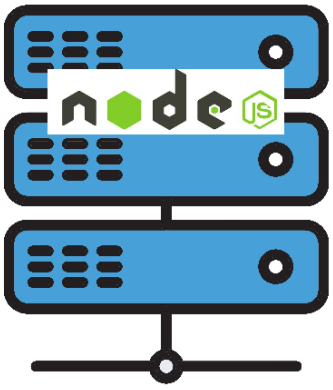
Não se preocupe se algumas palavras parecerem novas agora. Explicaremos tudo com calma e usando exemplos do mundo real para que você possa entender facilmente. O importante é ter curiosidade e vontade de aprender.

Este livro, revisado com auxílio de IA, resume os conceitos principais da linguagem e é complementado com os exercícios do curso de Node.js ministrado pelo professor Celso Yoshikazu Ishida e os vídeos indicados pelos QrCode. Os exercícios aparecem na ordem crescente de aprendizagem. Alguns exercícios são recomendados a serem feitos com o IDE + IA. O IDE (do inglês, é o Ambiente de Desenvolvimento Integrado) é um software de auxílio para a programação. Existem algumas recomendadas, veja a explicação: http://200.17.199.250/siteprototipo/curso/Node.js/qrj001.html

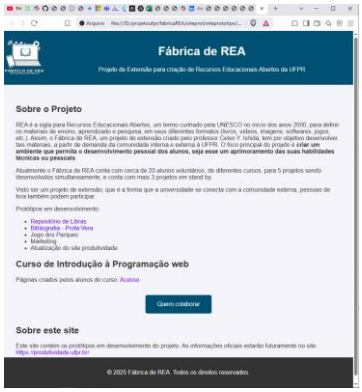
Utilizaremos um IDE e o auxílio de uma Inteligência Artificial para a programação de interfaces (Front-end), ou seja, código que serão executados na máquina do cliente e não no servidor da sua empresa. Aprenderemos a criar os programas em Node.js através de prompts e mais do que orquestrar (pedir para a IA criar o código), encorajamos a você entender os detalhes desta linguagem poderosa.

Então, prepare-se para essa jornada! Vamos explorar juntos o universo do Node.js e descobrir o poder que essa linguagem pode colocar nas suas mãos. Esta apostila deve ser lida em paralelo ao curso. Quem sabe, ao final deste, você não só entenderá o que é Node.js, mas também será capaz de criar suas próprias "mágicas" na web!

Vamos nessa?



E existe código Node.js que é executado no servidor (que é Node.js) e o Javascript que é executado a partir do HTML. Neste livro estamos interessados em código para ser executado no servidor



ÍNDICE

Sumário

Tópicos Essenciais para um Curso Básico de NodeJS

Olá a todos! Sejam muito bem-vindos ao nosso curso de Node.js Básico. Se você já tem familiaridade com HTML, CSS e Node.js, está no lugar certo para dar o próximo passo e mergulhar no mundo do desenvolvimento back-end.

Neste curso, vamos desmistificar o Node.js e construir uma base sólida para você criar suas próprias aplicações web. Deixaremos de lado o front-end por um momento para focar em como o Node.js permite que o Node.js funcione do lado do servidor, manipulando dados, interagindo com bancos de dados e servindo conteúdo para o seu front-end.

Prepare-se para entender como a internet realmente funciona "por trás das cenas" e para transformar suas ideias em aplicações web completas. Vamos começar!

Tópicos importantes

Para construir uma base sólida em Node.js, alguns tópicos são realmente essenciais. Organizamos este livro para o aprendizado progressivo através de exemplos. Aqui estão alguns tópicos essenciais para quem quer aprender a construir APIs com Node.js que acessam um banco de dados MySQL:

1. Introdução ao Node.js

- Configuração do ambiente (instalação do Node.js e npm)
- Estrutura de um projeto Node.js
- Uso do `package.json` e gerenciamento de dependências

2. Fundamentos do Express.js

- Criando um servidor com Express
- Configuração de rotas e middleware
- Tratamento de requisições e respostas

3. Conexão com MySQL no Node.js

- Configuração do MySQL e criação do banco de dados
- Uso do pacote `mysql2` para conectar o banco
- Operações básicas: SELECT, INSERT, UPDATE e DELETE

4. Criação de API REST

- Métodos HTTP (GET, POST, PUT, DELETE)
- Estruturação de endpoints RESTful
- Respostas em formato JSON

5. Autenticação e Segurança

- Uso de JWT para autenticação

- Proteção contra SQL Injection e ataques comuns
- Middleware para verificação de usuário

6. Manipulação de Dados e Validação

- Uso do `express-validator` para validar inputs
- Tratamento de erros e mensagens padronizadas

7. Arquitetura e Boas Práticas

- Separação de camadas (controllers, services e repositories)
- Uso de variáveis de ambiente (`dotenv`)
- Organização do código para facilitar manutenção

8. Deploy e Monitoramento

- Como hospedar uma API Node.js (Heroku, VPS, Railway)
- Logs e monitoramento com ferramentas como PM2
- Testes básicos de carga e otimizações

Esses são os pilares do Node.js puro. Dominando esses conceitos, você terá uma base sólida para construir interatividade e funcionalidades complexas diretamente no navegador.

Introdução ao Node.js para Programação de APIs

O Node.js revolucionou o desenvolvimento backend ao permitir que os desenvolvedores utilizem JavaScript para criar servidores e APIs robustas. Diferente de tecnologias tradicionais que trabalham de forma síncrona, o Node.js consegue trabalhar com a abordagem assíncrona e baseada em eventos, permitindo alto desempenho em aplicações que precisam lidar com múltiplas requisições simultâneas.

O que é o Node.js?

Node.js é um **ambiente de execução** (runtime) que permite **executar código JavaScript fora do navegador**, ou seja, no lado do servidor. Ele utiliza o **motor V8** do Google Chrome, que é um motor de JavaScript de alta performance, para compilar e executar o código rapidamente.

O Node.js é conhecido por sua **arquitetura orientada a eventos e não bloqueante (non-blocking I/O)**, o que o torna **extremamente eficiente e escalável** para lidar com um grande número de requisições simultâneas. Isso o torna uma excelente escolha para o desenvolvimento de:

- **APIs RESTful**
- **Microsserviços**
- **Aplicações em tempo real** (como chats e jogos online)
- **Servidores web**
- **Ferramentas de linha de comando**

Por que usar Node.js para criar APIs?

1. Alta performance: Seu modelo assíncrono permite processar milhares de requisições simultaneamente sem bloquear operações.

2. Comunidade ativa: Uma grande variedade de pacotes no NPM (Node Package Manager) facilita a criação de APIs sem precisar reinventar a roda.

3. Fácil integração com bancos de dados: Funciona bem com SQL (MySQL, PostgreSQL) e NoSQL (MongoDB, Firebase).

4. Uso de JavaScript no frontend e backend: Unificar a tecnologia facilita a curva de aprendizado e melhora a produtividade do time de desenvolvimento.

Construção de uma API com Node.js

O processo de criação de uma API envolve:

- Configuração do ambiente (instalação do Node.js).

- Uso de Express.js para facilitar a criação de rotas e endpoints (Um endpoint é um ponto de interação ou um endereço específico de uma API onde recursos podem ser acessados ou operações podem ser realizadas.

).

- Conexão com um banco de dados, como MySQL.

- Implementação de middleware para autenticação, segurança e validação de dados.

- Deploy em serviços na nuvem, como Heroku, Vercel ou AWS.

Revisão de JavaScript para Iniciantes em Node.js e MySQL

Se você está prestes a mergulhar na criação de APIs com Node.js e MySQL, é essencial revisar alguns conceitos fundamentais de JavaScript, já que o Node.js utiliza essa linguagem para a programação backend. Aqui está um guia rápido para reforçar o básico antes de avançar para a construção de APIs.

1. Variáveis e Tipos de Dados

Em JavaScript, usamos `let` e `const` para declarar variáveis (evite `var`, pois pode gerar problemas de escopo).

Exemplo:

```
let nome = "Maria"; // Pode ser reatribuído
const idade = 25; // Valor fixo
```

Os principais tipos de dados incluem string, number, boolean, array e object.

2. Funções

Funções são essenciais para organizar código e evitar repetição.

```
function saudacao(nome) {
    return `Olá, ${nome}!`;
}

console.log(saudacao("João"));
```

Também podemos usar arrow functions:

```
const saudacao = (nome) => `Olá, ${nome}!`;
```

3. Objetos e Arrays

Manipular dados estruturados é essencial para comunicação com bancos de dados.

```
const usuario = { nome: "Carlos", idade: 30 };
console.log(usuario.nome); // Acesso à propriedade

const lista = ["Maçã", "Banana", "Pera"];
console.log(lista[1]); // Banana
```

4. Operações Assíncronas (Promises e Async/Await)

O Node.js lida frequentemente com operações assíncronas, como chamadas ao banco de dados.

Promise

Em JavaScript, especialmente no Node.js, uma Promise é um objeto que representa a eventual conclusão (ou falha) de uma operação assíncrona e seu valor resultante.

```
const pegarDados = () => {
    return new Promise((resolve) => {
```

```
        setTimeout(() => resolve("Dados carregados!"), 2000);
    });
};

pegarDados().then((mensagem) => console.log(mensagem));
```

Ou seja, sempre que tiver um `.then()`, você tem uma chance de processar os dados recebidos antes de passar para a próxima etapa. E esse processamento pode incluir validações, cálculos, ajustes em estruturas, filtros, ou até chamadas adicionais.

Agora, com async/await:

```
async function carregarDados() {
    const mensagem = await pegarDados();
    console.log(mensagem);
}

carregarDados();
```

O uso de async/await é mais intuitivo e melhora a legibilidade do código.

5. Módulos em JavaScript

No Node.js, você dividirá seu código em arquivos/modularizar funcionalidades.

Exportação e importação de módulos:

```
// arquivo.js
const mensagem = "Olá, mundo!";
module.exports = mensagem;

// app.js
const mensagemImportada = require("./arquivo");
console.log(mensagemImportada);
```

Isso será útil ao estruturar a API em múltiplos arquivos.

6. Comunicação com Banco de Dados

Para interagir com um banco de dados MySQL, você usará bibliotecas como `mysql2/promise`.

Conectar ao MySQL no Node.js:

```javascript

```
const mysql = require("mysql2/promise");

async function conectarBanco() {
    const conexao = await mysql.createConnection({
```
```

```
        host: "localhost",
        user: "root",
        password: "senha",
        database: "meu_banco"
    });

    console.log("Conectado ao banco de dados!");
}

conectarBanco();
```

Esse conceito será essencial ao criar APIs RESTful que armazenam e recuperam informações de um banco de dados.

Com essa revisão rápida de JavaScript, você já estará preparado para avançar na construção de uma API com Node.js e MySQL. Os conceitos de assíncronismo, módulos e interação com bancos de dados serão fundamentais no seu aprendizado.

Qrcode perguntas

Introdução ao Node.js

Se você quer começar a desenvolver APIs com Node.js e conectar ao banco de dados MySQL, o primeiro passo é entender o básico do ambiente de desenvolvimento. Aqui está o que você precisa saber:

Configuração do ambiente

Antes de mais nada, você precisa instalar o Node.js e o npm (gerenciador de pacotes do Node). Para isso:

- Baixe o instalador do Node.js no [site oficial](https://nodejs.org) e siga as instruções.

- Após a instalação, verifique se tudo está funcionando com os comandos no prompt de comando:

```
node -v   # Verifica a versão do Node.js
 npm -v    # Verifica a versão do npm
```

Estrutura de um projeto Node.js

Depois de configurar o ambiente, é importante entender a estrutura básica de um projeto Node.js. Normalmente, ele inclui:

- Pasta `src` ou `app`: contém os arquivos principais do projeto.

- Arquivo `server.js` ou `index.js`: ponto de entrada da aplicação.

- Pasta `routes`: define os endpoints da API.

- Pasta `controllers`: onde fica a lógica dos handlers das requisições.

- Pasta `models` ou `database`: interação com o banco de dados.

- Arquivo `.env`: variáveis de ambiente (como credenciais do banco).

Uso do `package.json` e gerenciamento de dependências

O arquivo `package.json` é essencial para um projeto Node.js, pois gerencia as dependências e informações do projeto. Para criar um, execute:

```
npm init -y
```

Isso criará um arquivo `package.json` básico. Depois, você pode instalar pacotes como Express (framework para criar APIs) e mysql2 (para interagir com MySQL):

```
npm install express mysql2 dotenv
```

e já pode criar o primeiro arquivo de testes: teste.js

```
const express = require('express');
const app = express();
const port = 3000;
```

```
// Rota básica
app.get('/', (req, res) => {
    res.json({ mensagem: "Olá, mundo! Bem-vindo à minha API básica em Node.js!" });
});

app.get('/segundo', (req, res) => {
    res.json({ nomevar: "Teste bem sucedido! Segundo",
                data: new Date().toISOString()
     });
});

// Inicia o servidor
app.listen(port, () => {
    console.log(`Servidor rodando em http://localhost:${port}`);
});
```

Para executar:

```
node teste.js
```

É criado um servidor web para testes, para visualizar o resultado você pode acessar o endereço `http://localhost:3000` no navegador.

Explicação presencial: Execução teste.js e explicação sobre servidor, portas, rotas, json.

Explicação presencial: Execução teste.js e curso.js e explicação sobre servidor, portas, rotas, json.

Utilizando middleware

Middlewares são funções que processam requisições antes de serem enviadas às rotas. Por exemplo, para habilitar o uso de `JSON` no corpo da requisição:

```
app.use(express.json());
```

Podemos também criar um middleware personalizado:

```
app.use((req, res, next) => {
    console.log(`Requisição recebida em: ${req.url}`);
    next();
});
```

Isso permite registrar cada requisição feita à API no terminal.

Tratamento de requisições e respostas

Requisições GET e POST

Além de `GET`, podemos definir rotas `POST` para receber dados do cliente:

```
app.post('/usuarios', (req, res) => {
    const { nome } = req.body;
    res.status(201).json({ mensagem: `Usuário ${nome} criado com sucesso!` });
});
```

Certifique-se de enviar requisições `POST` com um corpo JSON:

```
{
    "nome": "Carlos"
}
```

Tratamento de erros

Para capturar erros e garantir que o servidor não quebre inesperadamente:

```
app.use((err, req, res, next) => {
    console.error(err);
    res.status(500).json({ mensagem: 'Erro interno do servidor' });
});
```

Execução middle.js e usuário.js explicação sobre servidor, portas, rotas, json.

Agora, vamos para os exercícios: qrnod001

API Rest

Uma API REST (Representational State Transfer) é um estilo de arquitetura para a comunicação entre sistemas, utilizando o protocolo HTTP. Ela permite que diferentes aplicações troquem informações de forma padronizada e eficiente.

Características principais

- Baseada em HTTP: Usa métodos como GET, POST, PUT e DELETE para interagir com os dados.
- Estruturada em recursos: Cada entidade (como usuários ou produtos) é representada por uma URL específica.
- Independente de plataforma: Pode ser consumida por qualquer sistema que suporte requisições HTTP.
- Respostas em JSON ou XML: JSON é o formato mais comum devido à sua simplicidade e compatibilidade.

Conexão com MySQL no Node.js

Até agora, construímos APIs que respondem com dados estáticos ou manipulados em memória. Mas a maioria das aplicações reais precisa armazenar e recuperar informações de forma persistente. É aí que os bancos de dados entram em jogo! Neste capítulo, vamos aprender a conectar sua API Node.js a um banco de dados MySQL, um dos bancos de dados relacionais mais populares e robustos.

Configuração do MySQL e Criação do Banco de Dados

Antes de conectar seu Node.js ao MySQL, você precisa ter um servidor MySQL instalado e um banco de dados configurado. E, aconselhamos o curso MySQL para quem não conhece nenhum SGBD (Sistema gerenciador de banco de dados).

Banco de dados utilizado

Usaremos duas tabelas: Curso e Usuario. Você pode utilizar o servidor de banco disponibilizado durante os cursos ou pode baixar o script para criação das tabelas no seu banco de dados local:

Uso do Pacote mysql2 para conectar o Banco

O módulo `mysql2` é uma biblioteca otimizada para trabalhar com bancos de dados MySQL dentro do Node.js. Ele suporta promises e consultas preparadas, tornando o uso mais eficiente e seguro.

Passos para conectar ao banco de dados

Instalar o pacote `mysql2`

Antes de começar, instale o módulo necessário no Prompt de comando:

```
npm install mysql2
```

Importar o módulo e criar a conexão

O primeiro passo é importar `mysql2` e estabelecer uma conexão com o banco de dados:

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

// Configuração da conexão
const connection = mysql.createConnection({
    host: 'localhost',  // Endereço do servidor MySQL
```

```
    user: 'root',       // Nome de usuário do banco
    password: 'senha',  // Senha do banco
    database: 'meu_banco' // Nome do banco de dados
});

// Conectar ao banco
connection.connect(err => {
    if (err) {
        console.error('Erro ao conectar:', err);
        return;
    }
    console.log('Conectado ao MySQL!');
});
```

Obs: se for utilizar o exemplo fornecido, ‘meu_banco’ deve ser substituído por ‘cursonode’ abra o arquivo .sql e procure para

Executando consultas

Depois de estabelecer a conexão, você pode executar comandos SQL:

```
connection.query('SELECT * FROM usuarios', (err, results) => {
    if (err) {
        console.error('Erro na consulta:', err);
        return;
    }
    console.log('Resultados:', results);
});

// Fechar conexão após uso
connection.end();
```

Usando Promises para consultas assíncronas

Caso prefira utilizar async/await, você pode usar `mysql2/promise`:

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

async function conectar() {
    const connection = await mysql.createConnection({
        host: 'localhost',
        user: 'root',
        password: 'senha',
        database: 'meu_banco'
    });

    const [rows] = await connection.execute('SELECT * FROM usuarios');
    console.log(rows);
```

```
    await connection.end();
}

conectar();
```

Usar `mysql2` no Node.js é uma abordagem eficiente para conectar-se ao MySQL. A versão `promise` ajuda a manter um código mais organizado e fácil de gerenciar.

Usar Promises com `mysql2` pode trazer várias vantagens, mas também há algumas considerações importantes. Vamos analisar os principais pontos:

Vantagens de usar Promises com mysql2

- Código mais legível e organizado: Com `async/await`, evitamos o encadeamento excessivo de `.then()`, deixando o código mais intuitivo.
- Fluxo assíncrono mais previsível: As consultas não bloqueiam a execução do restante do código, melhorando a eficiência da aplicação.
- Melhor tratamento de erros: Com `try/catch`, capturar erros se torna mais simples e direto.
- Facilidade de integração com frameworks modernos: Muitos frameworks modernos (como Express) lidam bem com `async/await`, tornando a integração mais fluida.

Desvantagens e considerações

- Top-level await pode bloquear módulos: Se usado fora de uma função `async`, pode retardar a inicialização do sistema.
- Menor compatibilidade com versões antigas do Node.js: Algumas versões mais antigas não suportam `async/await`, exigindo ajustes no código.
- Manutenção de conexão com o banco: Como `Promises` não encerram automaticamente a conexão, é necessário garantir que `connection.end()` seja chamado corretamente.

Boas práticas

- Sempre use `await` dentro de funções `async` para evitar problemas de bloqueio.
- Evite misturar `.then()` e `async/await` no mesmo código, para manter a consistência.
- Use tratamento de erros com `try/catch`, para evitar falhas silenciosas no banco de dados.
- Gerencie conexões corretamente, utilizando pools de conexão para melhorar a escalabilidade:

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

async function conectar() {
    const pool = mysql.createPool({
        host: 'localhost',
        user: 'root',
        password: 'senha',
        database: 'meu_banco',
        waitForConnections: true,
        connectionLimit: 10,
```

```
        queueLimit: 0
    });

    const [rows] = await pool.query('SELECT * FROM usuarios');
    console.log(rows);

    pool.end(); // Finaliza o pool corretamente
}

conectar();
```

Agora vamos para os exercícios

O que são Variáveis de Ambiente?

São valores armazenados fora do código-fonte, evitando que senhas, chaves de API e outras informações sensíveis fiquem expostas.

Instalação do dotenv

Para facilitar o gerenciamento dessas variáveis, usamos o pacote dotenv. Primeiro, instale com:

```
npm install dotenv
```

Configuração do dotenv

Crie um arquivo chamado `.env` na raiz do projeto e adicione suas configurações:

```
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=senha_super_secreta
DB_DATABASE=meu_banco
```

Depois, no arquivo do servidor (ex: `server.js`), importe e configure o dotenv:

```
require('dotenv').config();

const mysql = require('mysql2');

// Criação da conexão com MySQL usando variáveis de ambiente
const connection = mysql.createConnection({
    host: process.env.DB_HOST,
    user: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE
});

connection.connect(err => {
    if (err) {
        console.error('Erro ao conectar:', err);
        return;
    }
    console.log('Conectado ao MySQL!');
});
```

Boas práticas

- Nunca exponha o arquivo `.env` no repositório. Adicione `*.env` ao `.gitignore` para evitar problemas de segurança.

- Use variáveis para diferentes ambientes (desenvolvimento, teste, produção).

- Evite hardcoded (informações sensíveis direto no código), sempre prefira variáveis de ambiente.

Essa abordagem torna sua API mais segura e profissional!

Agora, tente implementar isso no seu arquivo

Pool de Conexões no Node.js com MySQL

Em aplicações Node.js que interagem com bancos de dados MySQL, é essencial gerenciar conexões corretamente para evitar problemas de desempenho e sobrecarga. Uma prática recomendada em ambientes de produção é usar um pool de conexões em vez de abrir e fechar conexões individuais a cada requisição.

O que é um Pool de Conexões?

O pool de conexões permite que a aplicação mantenha um conjunto de conexões ativas reutilizáveis, evitando a sobrecarga do banco de dados. Em vez de abrir e fechar conexões a cada requisição, o pool gerencia automaticamente as conexões disponíveis e aloca para processos que precisam delas.

Benefícios do Uso de Pool de Conexões

- Eficiência: Reduz a latência ao evitar a abertura repetitiva de novas conexões.

- Melhor desempenho: Requisições podem ser atendidas mais rapidamente, já que a conexão está pronta para uso.

- Evita gargalos: Previne a sobrecarga do banco de dados ao limitar o número de conexões simultâneas.

Como funciona o pool de conexões na prática?

1. Quando um usuário faz uma requisição à API, o servidor verifica se há uma conexão disponível no pool.

2. Se houver uma conexão livre, ela é usada para executar a consulta no banco de dados.

3. Após a execução da consulta, a conexão retorna ao pool para ser reutilizada por outra requisição.

4. Se todas as conexões estiverem ocupadas, novas requisições ficam na fila até que uma conexão seja liberada.

Exemplo simplificado

Imagine que você definiu um `connectionLimit: 10`, ou seja, seu pool pode gerenciar até 10 conexões simultâneas. Se sua API receber 100 usuários tentando acessar dados ao mesmo tempo:

- Os primeiros 10 usuários terão suas requisições atendidas imediatamente.

- Os demais 90 usuários ficarão na fila até que uma conexão seja liberada para processar suas requisições.

Isso evita criar conexões excessivas e mantém o desempenho da aplicação sob controle. Se sua aplicação precisa atender muitos usuários ao mesmo tempo, pode aumentar o `connectionLimit` ou otimizar consultas para liberar conexões mais rapidamente.

Configurações Importantes do Pool

- `connectionLimit`: Define o número máximo de conexões simultâneas no pool.

- `waitForConnections`: Se `true`, a solicitação de conexão aguarda um slot disponível no pool.

- `queueLimit`: Número máximo de requisições que podem ficar na fila esperando por uma conexão.

Próximos Passos

Com essas operações básicas, você já pode construir APIs completas que interagem com um banco de dados MySQL. Lembre-se de organizar seu código, separando as rotas em arquivos diferentes e, em aplicações maiores, criando camadas de serviço para a lógica de negócio e repositórios para a interação com o banco de dados.

Para continuar aprimorando suas habilidades, considere explorar:

- **Tratamento de erros mais robusto:** Implementar blocos `try-catch` ou um middleware de tratamento de erros global.
- **ORMs (Object-Relational Mappers):** Ferramentas como Sequelize ou TypeORM abstraem as consultas SQL, permitindo que você interaja com o banco de dados usando objetos JavaScript.

Com essa base, você está pronto para construir APIs mais dinâmicas e persistentes com Node.js e MySQL!

Operações básicas SQL

Linguagem SQL é a linguagem para manipulação das informações dentro de um banco de dados relacional. A seguir, mostraremos exemplos de cada um dos comandos SQL e logo em seguida o código da API em Nodejs para execução dos mesmos.

SELECT

Exemplo do comando para selecionar as colunas Campus e Nome de todos os registros com a coluna Id menor do que 10. Os registros são retornados em ordem crescente da coluna Campus.

```
select Campus, Nome
from Curso
where Id < 10
order by Campus
```

Código Node.js para a API que executa a consulta, note a presença do GET:

```
// API
app.get('/cursos', (req, res) => {
  const sql = `SELECT Campus, Nome FROM Curso WHERE Id < 10 ORDER BY Campus`;
  db.query(sql, (err, results) => { // executa a consulta
    if (err) {
      console.error('Query error:', err);
      return res.status(500).json({ error: 'Database error' });
    }
    res.json(results);
  });
});
```

Veja a explicação da criação de uma API para o comando select:

INSERT

Inserção de um registro com na tabela Usuario. Note o formato da data e os campos numéricos (DDD e telefone). Neste exemplo o campo Obs está com valor nulo.

```
INSERT INTO Usuario (Nome, Email, CPF, DDD, Telefone, DataNiver, Obs)
```

```
VALUES ('Ana Silva', 'ana.silva@example.com', '12345678901', 11, 987654321, '1990-
05-12', null);
```

API para executar a inserção do usuário, note que é POST. E no retorno da mensagem, indica o ID do novo registro:

```
// API para inserir usuário
app.post('/cursosi', express.json(), (req, res) => {
  const { Nome, Email, CPF, DDD, Telefone, DataNiver, Obs } = req.body;
  const sql = `INSERT INTO Usuario (Nome, Email, CPF, DDD, Telefone, DataNiver, Obs)
VALUES (?, ?, ?, ?, ?, ?, ?)`;
  const values = [Nome, Email, CPF, DDD, Telefone, DataNiver, Obs];
  db.query(sql, values, (err, result) => {
    if (err) {
      console.error('Insert error:', err);
      return res.status(500).json({ error: 'Database error', sqlMessage:
err.sqlMessage, sqlState: err.sqlState, code: err.code });
    }
    res.json({ success: true, insertedId: result.insertId });
  });
});
```

Veja a interação para a criação da API de insert e como o agente pode ajudar na manipulação dos erros.

Agora um pequeno desafio: crie uma API para selecionar dados da tabela Usuario e, assim, ficar mais fácil para verificar se o insert foi executado corretamente.

UPDATE

Atualização do telefone e campo Obs do registro com Id 4 da tabela Usuario

```
UPDATE Usuario
SET Telefone = 912345678, Obs = 'Telefone atualizado em 2025'
WHERE Id = 4;
```

Código Node para Update:

```
// API para atualizar usuário
```

```
app.put('/cursosu', express.json(), (req, res) => {
  const { Id, Telefone, Obs } = req.body;
  const sql = `UPDATE Usuario SET Telefone = ?, Obs = ? WHERE Id = ?`;
  const values = [Telefone, Obs, Id];
  db.query(sql, values, (err, result) => {
    if (err) {
      console.error('Update error:', err);
      return res.status(500).json({ error: 'Database error', sqlMessage:
err.sqlMessage, sqlState: err.sqlState, code: err.code });
    }
    console.log('Rows affected:', result.affectedRows, 'Id:', Id);
    res.json({ success: true, affectedRows: result.affectedRows });
  });
});
```

Veja a criação de uma API para update: qrnod015

DELETE

Apagando o registro com Id igual a 5.

```
DELETE FROM Usuario
WHERE Id = 5;
```

Desafio: crie uma API para apagar um registro na tabela. Verifique o resultado:

Agora vamos para alguns conceitos que foi apresentado:

Métodos HTTP

Os métodos HTTP são essenciais para a comunicação entre cliente e servidor em uma API REST. Os principais métodos são:

- GET: Recupera dados do servidor sem alterar seu estado.
- POST: Envia dados ao servidor para criação de um novo recurso.

- PUT: Atualiza um recurso existente no servidor.
- DELETE: Remove um recurso do servidor.

Estruturação de Endpoints RESTful

Uma API REST deve seguir boas práticas na definição de seus endpoints:

- Utilizar nomes de recursos no plural, como `/users`, `/products`.
- Seguir uma estrutura lógica, garantindo previsibilidade, como `/users/{id}` para obter um usuário específico.
- Evitar verbos na URL, já que a ação é definida pelo método HTTP. Exemplo: usar `POST /users` em vez de `/createUser`.

Respostas em Formato JSON

O JSON (JavaScript Object Notation) é o formato padrão para troca de dados em APIs REST. É leve, fácil de ler e compatível com diversas linguagens. Uma resposta típica em JSON pode ter a seguinte estrutura:

```
{
  "id": 1,
  "name": "João Silva",
  "email": "joao@email.com"
}
```

Agora vamos para os exercícios:

Autenticação e Segurança em APIs REST

Garantir a segurança de uma API REST é fundamental para proteger dados e evitar ataques maliciosos. Aqui estão três aspectos essenciais:

Uso de JWT para Autenticação

O JWT (JSON Web Token) é um método popular para autenticação em APIs. Ele funciona assim:

1. O usuário faz login e recebe um token JWT.

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  if (username === user.username && password === user.password) {
    const token = jwt.sign({ id: user.id, username: user.username }, SECRET_KEY,
{ expiresIn: '1h' });
    return res.json({ token });
  }
  res.status(401).json({ error: 'Invalid credentials' });
});
2.
```

2. Esse token é enviado em cada requisição para autenticação.
3. O servidor valida o token e permite o acesso ao recurso.

```
// Middleware to verify JWT
function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];
  if (!token) return res.sendStatus(401);
  jwt.verify(token, SECRET_KEY, (err, user) => {
    if (err) return res.sendStatus(403);
    req.user = user;
    next();
  });
}

// Protected route
app.get('/protected', authenticateToken, (req, res) => {
  res.json({ message: 'É uma área protegida só mostra esta frase se
authenticateToken verificar o token com sucesso!', user: req.user });
});
```

Agora, baixe e execute o código completo:

Proteção contra SQL Injection e Ataques Comuns

Algumas medidas para evitar vulnerabilidades em APIs:

SQL Injection

Use ORMs (como Sequelize) ou consultas parametrizadas para evitar inserção de código malicioso.

```
const [rows] = await pool.query('SELECT * FROM Usuario WHERE Id = ?', [id]);
```

O uso do '?' e do segundo parâmetro dentro de um array evita este problema

Cross-Site Scripting (XSS)

Escape inputs do usuário antes de exibi-los. Para evitar sempre use a função xss para filtrar as informações recebidas do usuário. Exemplo de uso

```
const xss = require('xss');
...
app.post('/comentario', (req, res) => {
  // Extração e sanitização dos campos
  const nome = req.body.nome ? xss(req.body.nome) : '';
  const comentario = req.body.comentario ? xss(req.body.comentario) : '';
  const outroCampo = req.body.outroCampo ? xss(req.body.outroCampo) : '';
  ...
```

- Cross-Site Request Forgery (CSRF): Utilize tokens CSRF para evitar requisições mal-intencionadas.

Explicações: 

- Rate Limiting: Restrinja o número de requisições por IP para evitar ataques de força bruta.

Uma das maneiras mais práticas de implementar rate limiting (limitação de requisições) em uma API Node.js com Express é usando o pacote express-rate-limit. Ele ajuda a proteger sua aplicação contra abusos, como ataques de força bruta e excesso de chamadas por IP.

```
const express = require('express');
const rateLimit = require('express-rate-limit');
const app = express();

// Middleware para limitar 100 requisições por IP a cada 15 minutos
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutos
  max: 100, // máximo de 100 requisições por IP
```

```
  message: {
    status: 429,
    erro: 'Muitas requisições — tente novamente mais tarde.'
  }
});

app.use(limiter);
```

CONTINUA...