# 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.

- 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.

Exemplo de Promise:

const pegarDados = () => {

    return new Promise((resolve) => {

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

    });

};

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

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:

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.

# 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 teste: 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

Acesse ‘http://localhost:3000’ no navegador

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

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:



# 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 que pode ser baixado:

## 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!');

});

#### 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 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: qrcode002



## 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.

# 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