

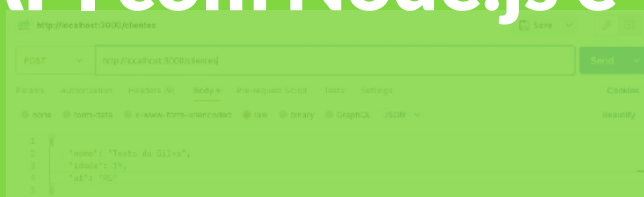


NODE.JS

Criando uma WebAPI com Node.js e MySQL



Escrito por **Luiz Duarte**
em 27/06/2023



JUNTE-SE A MAIS DE 34 MIL DEVS

Entre para minha lista e receba conteúdos exclusivos e com prioridade

Hoje vou ensinar como que você pode usar Node.js com MySQL para criar uma WebAPI. WebAPIs são a base do conhecimento para trabalhar com backend, então começar aprendendo como criar uma bem feita irá lhe gerar bons frutos na sua carreira.

Existem diversas maneiras de usar MySQL com Node. Uma vez que é um banco de dados muito popular, logo, é possível que você encontre muita informação diferente na Internet à respeito. Entenda que te mostrarei UMA das INÚMERAS opções de como conectar no MySQL com Node.js.

Veremos neste artigo:

1. Criando o banco de dados
2. Criando a API
3. Criando a listagem de clientes
4. Criando a pesquisa de um cliente
5. Excluindo um cliente
6. Adicionando um cliente
7. Atualizando um cliente
8. Indo Além

Opcionalmente, você pode assistir ao vídeo abaixo, que possui o mesmo conteúdo.

[Live] Aulão de Backend Node.js + MySQL



Então vamos lá!

Parte 1: Criando o banco de dados

Se você já possui um servidor de MySQL instalado na sua máquina com um banco de dados qualquer, use ele e pule para o próximo trecho após o vídeo. Caso contrário, o vídeo abaixo vai te ensinar a instalar o MySQL na sua máquina. Depois de instalar, abra a ferramenta MySQL Workbench (ou equivalente) e crie um schema (banco de dados) novo.



Como instalar MySQL no Windows



Agora que você já tem o banco pronto, vamos criar uma tabela nele e colocar alguns dados de exemplo. Aqui usaremos uma tabela de clientes que você deve criar visualmente com o MySQL Workbench ou executando o SQL abaixo, que deve servir como linha guia para a criação visual também:

```
1 CREATE TABLE IF NOT EXISTS Clientes(  
2   ID int NOT NULL AUTO_INCREMENT,  
3   Nome varchar(150) NOT NULL,  
4   Idade int NOT NULL,  
   UF char(2) NOT NULL,  
   PRIMARY KEY (ID)
```

Com banco e tabela preparados, adicione alguns clientes manualmente e podemos avançar para construção da aplicação.

Parte 2: Criando a API

Crie uma pasta para guardar os arquivos do seu projeto Node.js, você pode fazer isso pelo console se quiser, usaremos ele algumas vezes nesse tutorial. No exemplo abaixo, criei a pasta e depois entrei dentro dela.

```
1 mkdir nodejs-mysql-webapi  
2 cd nodejs-mysql-webapi
```

Agora execute no console o comando “npm init -y” que o próprio NPM (gerenciador de pacotes do Node) vai fazer a construção do arquivo **package.json**, que é o arquivo de configuração do projeto. Com o arquivo de configurações criado, vá no console novamente, na pasta do projeto e digite o seguinte comando para instalar os pacotes que vamos precisar para nosso projeto:

```
1 npm install mysql2 dotenv express
```

Esses pacotes servem para:

- ◇ mysql2: conectar e mandar comandos SQL para o banco;
- ◇ dotenv: gestão das configurações do projeto;
- ◇ express: web framework para construção da infraestrutura da API;

Na sequência, crie um arquivo .env (sem nome, apenas .env) na raiz do projeto e coloque dentro dele as seguintes variáveis, preenchendo conforme o comentário logo acima de cada uma.


```
1 # The webapi port. Ex: 3000
2 PORT=
3
4 #The Connection String to database. Ex: mysql://user:password@server:port/database
5 CONNECTION_STRING=
```

Agora vamos criar um arquivo index.js na pasta do projeto onde vamos criar o nosso servidor da API para tratar as requisições

que chegarão em breve. Vamos começar bem simples, apenas definindo as constantes locais que serão usadas mais pra frente e carregando o arquivo de configurações (.env):

```
1 require("dotenv").config();
2
3 const express = require('express');
4 const app = express();
5 const port = process.env.PORT;
```

Agora, logo abaixo do código acima, vamos configurar nossa aplicação (app) Express para usar o body parser do Express, permitindo que recebamos mais tarde POSTs no formato JSON e também uma rota de exemplo, que responderá às chamadas na raiz da webapi (/):

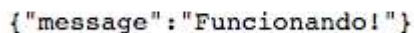


```
p.use(express.json());
p.get('/', (req, res) => res.json({ message: 'Funcionando!' }));
```

Note que eu fiz com que requisições que chegarem na raiz devam ser mandadas para a função que imprime uma mensagem. Por fim, adicionamos as linhas abaixo no final do arquivo que dão o start no servidor da API:

```
1 //inicia o servidor
2 app.listen(port);
3 console.log('API funcionando!');
```


Teste sua API executando via console o seu index.js com o comando 'node index.js'. Você deve ver a mensagem de 'API funcionando!' no console, e se acessar no navegador localhost:3000 deve ver o JSON default que deixamos na rota raiz!



API Funcionando

Parte 3: Criando a listagem de clientes

Agora, crie um arquivo db.js dentro dessa pasta, que será o arquivo que vai cuidar de toda comunicação com o banco de dados. Vamos começar carregando a dependência do MySQL, mais especificamente a pasta “promise” e criando um pool de conexões com ela:



```
const mysql = require('mysql2/promise');  
const client = mysql.createPool(process.env.CONNECTION_STRING);
```

A biblioteca mysql2 é superior à mysql em diversos aspectos e um deles é o suporte a promises, da forma que usaremos aqui neste tutorial. Se você é novo no mundo Javascript e não sabe o que é ou qual a vantagem das promises frente ao modelo tradicional de programação, recomendo o vídeo abaixo.

Callbacks, Promises e Async-Await no Node.js



Outra coisa digna de nota que a nossa conexão está sendo feita imediatamente ao carregar o módulo `db.js`, usando da configuração presente no arquivo `.env`. Além disso, ela não é uma conexão comum, mas sim um pool de conexões. Um pool é como se fosse um gerenciador de conexões, onde toda vez que usarmos ele (o objeto `client`), o pool vai verificar se possui uma conexão aberta. Se não possuir, ele irá criar. Se possuir, ele irá usar ela para envio dos comandos. Isso é algo crucial para performance de aplicações com bancos MySQL e nos exige também da responsabilidade em cuidar de conexões manualmente.

Para usar esse pool em nossas consultas e comandos SQL é muito fácil, como o código abaixo mostra, que deve ser colocado logo após a conexão. Atenção ao uso do índice 0 no array res, que serve para retornar somente as linhas do banco de dados.

```
1 async function selectCustomers() {  
2   const res = await client.query('SELECT * FROM clientes');  
3   return res[0];  
4 }  
5  
6 module.exports = { selectCustomers }
```

Agora que temos uma API minimamente estruturada e um módulo db.js com uma função para consultar clientes pronta, vamos importar o módulo db no index.js e adicionar uma rota /clientes que listará todos os clientes do banco de dados. Para isso, começaremos criando a rota /clientes logo abaixo da rota / (raiz):



```
p.get('/clientes', async (req, res) => {  
  const results = await db.selectCustomers();  
  res.json(results);  
})
```

Agora, ao executarmos novamente nosso projeto e ao acessarmos a URL localhost:3000/clientes, veremos todos os clientes cadastrados no banco de dados (no passo 2, lembra?):



localhost:3000/clientes

```
[{"ID":1,"Nome":"teste1","CPF":"12345678901"},  
{"ID":2,"Nome":"teste2","CPF":"09876543210"},  
{"ID":3,"Nome":"teste3","CPF":"12312312399"}]
```

E com isso finalizamos a listagem de todos clientes na nossa API!

Parte 4: Criando a pesquisa de um cliente

Agora, se o usuário quiser ver apenas um cliente, ele deverá passar o ID do mesmo na URL, logo após o /clientes. Para suportarmos isso na API, o primeiro passo é ir no db.js e criar uma nova função, como abaixo.

```
1 async function selectCustomer(id) {  
2   const res = await client.query('SELECT * FROM clientes WHERE ID=?', [id]);  
3   return res[0];  
4 }  
5  
6 module.exports = { selectCustomers, selectCustomer }
```

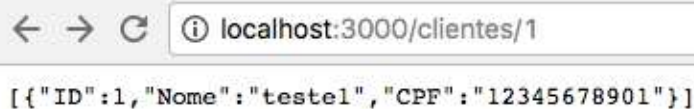


Repare aqui como eu recebo um id por parâmetro na função e passo ele dentro de um array após a consulta. Repare também que na consulta, onde deveria ter o id a ser usado como filtro, temos um '?'. Isso é chamado de Prepared Statement e na hora que for executado o comando, cada placeholder (?) é substituído por uma variável presente no segundo parâmetro do query, ou seja, o [id]. E antes que você pense que isso é apenas “perfumaria”, saiba que Prepared Statements nos protegem de SQL Injection, o tipo de ataque à banco de dados mais popular que existe (recomendo estudar sobre).

Para finalizar, vamos criar uma nova rota `/clientes/:id` para aceitar um parâmetro ID, como mostra o código abaixo. Certifique-se apenas de colocar este código ANTES da outra rota `/clientes` que usamos, caso contrário, como a rota anterior é mais genérica, ela irá monopolizar as requests.

```
1 app.get('/clientes/:id', async (req, res) => {  
2   const results = await db.selectCustomer(req.params.id);  
3   res.json(results);  
4 })
```

Note que aqui, para pegar o parâmetro `id` que virá na URL após `/clientes`, eu usei `req.params.id`. Mande rodar o projeto e teste no navegador, verá que está funcionando perfeitamente!



← → ↻ ⓘ localhost:3000/clientes/1

[{"ID":1,"Nome":"teste1","CPF":"12345678901"}]

E com isso terminamos a pesquisa por cliente.




Parte 5: Excluindo um cliente

Para excluir um cliente vamos fazer um processo parecido com o anterior. Começaremos criando uma função de exclusão no db.js, exportando-a ao final do módulo:

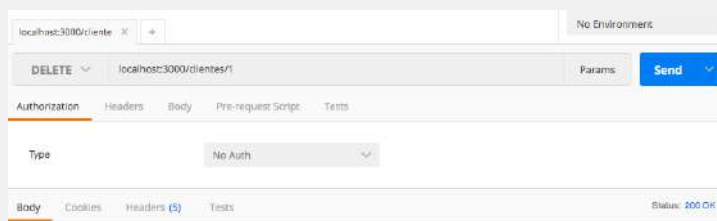
```
1 async function deleteCustomer(id) {  
2   return await client.query('DELETE FROM clientes where id=?;', [id]);  
3 }  
4  
5 module.exports = { selectCustomers, selectCustomer, deleteCustomer }
```

Depois, criaremos uma nova rota no index.js muito parecida com a de pesquisa de cliente, mas mudando o verbo HTTP de GET para DELETE, como manda o **protocolo HTTP**. Adicione a nova rota logo após as demais (repare que retorno o código 204, **ideal para esta situação**):



```
p.delete('/clientes/:id', async (req, res) => {  
  await db.deleteCustomer(req.params.id);  
  res.sendStatus(204);  
});
```

Para testar essa rota você tem duas alternativas, ou usa o **POSTMAN** para forjar um DELETE, como abaixo:



Ou fazer via console usando cURL (se tiver ele instalado na sua máquina):

```
1 curl -X DELETE http://localhost:3000/clientes/1
```

Em ambos os casos você deve obter uma resposta 200 OK ou 204 No Content (caso não tenha dado erro) e se mandar listar todos clientes novamente, verá que o cliente de id 1 (ou o número que você passou) sumiu.

Caso nunca tenha usado o Postman na sua vida, o vídeo abaixo pode ajudar.

Como usar o Postman para testar API pública/aberta



Parte 6: Adicionando um cliente

Agora vamos adicionar um novo cliente com um POST na rota /clientes. Comece pelo db.js, como sempre:

```
1 async function insertCustomer(customer) {  
2   const sql = 'INSERT INTO clientes(nome,idade,uf) VALUES (?, ?, ?)';  
3   const values = [customer.nome, customer.idade, customer.uf];  
4   await client.query(sql, values);  
5 }  
6  
7 module.exports = { selectCustomers, selectCustomer, deleteCustomer, insertCustomer }
```

Depois, adicione esta nova rota logo abaixo das anteriores no index.js.

```
1 app.post('/clientes', async (req, res) => {  
2   await db.insertCustomer(req.body);  
3   res.sendStatus(201);  
4 });
```



Nela, eu pego as variáveis que devem vir junto ao body do POST e depois passo elas para a função que possui um comando de INSERT que vai ser executado no banco de dados, usando novamente o conceito de Prepared Statements. Aqui, como temos diversos placeholders (?), cada um deles será substituído por um elemento no array values na mesma ordem em que aparecem na consulta SQL.

Para testar esse POST, você usar o POSTMAN ou cURL, como mencionado anteriormente, sendo que o JSON com os dados do cliente devem ser informados no corpo da requisição (body) e em header você

deve configurar para usar Content-Type como application/json.



Se testar agora vai ver que é possível inserir novos registros no banco de dados através de requisições POST.



Parte 7: Atualizando um cliente

E para finalizar o CRUD, vamos ver como podemos atualizar um cliente no banco de dados MySQL através da nossa API Node.js. Comece definindo a função de atualização no db.js.

```
1 async function updateCustomer(id, customer) {  
2   const sql = 'UPDATE clientes SET nome=?, idade=?, uf=? WHERE id=?';  
3   const values = [customer.nome, customer.idade, customer.uf, id];  
4   await client.query(sql, values);  
5 }  
6  
7 module.exports = { selectCustomers, selectCustomer, deleteCustomer, insertCustomer, updateCustomer }
```

Para fazer updates podemos usar os verbos PUT ou PATCH. O protocolo HTTP diz que devemos usar PUT se pretendemos passar todos os parâmetros da entidade que está sendo atualizada (o mais correto seria dizer “substituída”, neste caso), mas não vamos alterar jamais o ID, então usaremos PATCH nesta API. Crie uma rota PATCH em /clientes esperando o ID do cliente a ser alterado.

```
1 app.patch('/clientes/:id', async (req, res) => {  
2   await db.updateCustomer(req.params.id, req.body);  
3   res.sendStatus(200);  
4 })
```

No código acima, pegamos o ID que veio na URL e as demais informações que vieram no corpo da requisição. Depois passo para a função com o UPDATE, que vai executar o SQL. Para testar uma requisição PATCH, você pode usar o POSTMAN ou o cURL novamente, sem nenhuma novidade em relação ao que já fizemos. O resultado é que o cliente cujo ID você informar vai ter o seu nome, ou qualquer outro campo que passar, alterado para o que enviar no corpo da requisição. Note que se ele não existir, ocasionará um erro.

E com isso finalizamos o CRUD da nossa API Node.js que usa MySQL como persistência de dados.





Indo Além

Se você não curte muito a ideia de ficar usando SQL no meio dos seus códigos JS, experimente usar alguma biblioteca ORM (Object-Relational Mapping) como o [Sequelize](#).

Segurança é um fator muito importante em Web APIs profissionais e aqui não é exceção. Sugiro que dê uma olhada [neste artigo sobre JSON Web Token](#), que vai lhe ajudar com este tópico e também este aqui sobre [validação de input de dados](#).

Também vale a pena estar em um servidor bem configurado, [neste tutorial aqui](#) ensino como fazer isso na Amazon AWS, [tratar corretamente os erros com este tutorial](#) e [registrar os logs](#) da sua aplicação para poder investigar possíveis problemas.

Até a próxima!





TAGS:

mysql

nodejs

Artigos Relacionados





CRIPTO

**Integração com
Smart Contracts
com Node.js e
Web3.js**



NODE.JS

**Como compartilhar
Prisma
Schema/Client em
Monorepo**



NODE.JS

**Autenticação JSON
Web Token (JWT)
em NestJS**



NODE.JS



**Como usar
Variáveis de
Ambiente com
NestJS**

[Ver mais](#)

Olá, tudo bem?

O que você achou deste conteúdo? Conte nos comentários.

Escreva seu comentário...

Seu nome?

Seu email?

Postar

Entre para minha lista e receba conteúdos exclusivos e com prioridade

Junte-se a mais de 34 mil devs



Seu email

Enviar

Categorias

.NET (20)

Agile (106)

Android SDK (45)

Últimas Novidades

Como criar bot/robô trader para Uniswap V3 em Node.js

Tags

agile android

aws blo business c#

camunda carreira

[Carreira \(73\)](#)[Corona SDK \(2\)](#)[Cripto \(87\)](#)[Empreendedorismo \(66\)](#)[Livros \(52\)](#)[Mobile \(71\)](#)[Node.js \(210\)](#)[React Native \(14\)](#)[Web \(185\)](#)[Integração com Smart Contracts com Node.js e Web3.js](#)[Como criar uma nova criptomoeda usando Solidity, TypeScript e HardHat](#)[Como compartilhar Prisma Schema/Client em Monorepo](#)[Autenticação JSON Web Token \(JWT\) em NestJS](#)[corona](#)[criptomoedas](#)[digital ocean emp firebase](#)[heroku ios java jira](#)[mobile mongodb](#)[mssql mysql nestjs nextjs](#)[nodejs oauth](#)[performance phonegap](#)[postgresql prisma](#)[react redis regex](#)[resenha seguranca](#)[solidity sqlite tdd](#)[typescript web web3](#)

2010-23, LuizTools. Todos os direitos reservados.

