

MANIPULAÇÃO DE DADOS E INTEGRAÇÃO COM BANCOS DE DADOS

Sumário:

1. Introdução à Manipulação de Dados em Node.js	2
1.1 Conceitos básicos sobre manipulação de dados no servidor	2
1.2 Introdução ao formato JSON como padrão de troca de dados	3
2. Integração com Bancos de Dados Relacionais (SQL)	5
2.1 Introdução ao MySQL	5
2.2 Criando a Base de Dados	7
2.3 Criando um Projeto de Exemplo	8
2.4 Configuração e Conexão com MySQL	9
2.5 Organização e Estrutura do Projeto	10
2.6 Testando as Rotas do Servidor HTTP	14
3. Integração com Bancos de Dados NoSQL	17
3.1 Introdução ao MongoDB	17
3.2 Criando a Base de Dados	21
3.3 Introdução ao Mongoose	23
3.4 Criando um Projeto de Exemplo	24
3.5 Configuração e Conexão com MongoDB	25
3.6 Organização e Estrutura do Projeto	26
3.7 Testando as Rotas do Servidor HTTP	30
Referências	33

1. Introdução à Manipulação de Dados em Node.js

A manipulação de dados é um dos aspectos fundamentais no desenvolvimento de aplicações web e sistemas que interagem com usuários e serviços externos. No contexto de aplicações em Node.js, essa prática envolve a leitura, escrita, atualização e organização de dados de forma eficiente no servidor, garantindo uma comunicação eficaz entre o servidor e o cliente.

Nesta seção, exploraremos os conceitos básicos sobre manipulação de dados no servidor, utilizando o Node.js como base. Veremos como o formato JSON (JavaScript Object Notation) se tornou o padrão de troca de dados devido à sua simplicidade e compatibilidade com diversas linguagens de programação. O JSON permite a representação de dados estruturados de maneira fácil de ler e escrever, o que facilita a transmissão de informações entre sistemas de maneira rápida e eficiente.

1.1 Conceitos básicos sobre manipulação de dados no servidor

A manipulação de dados no servidor refere-se à capacidade de um servidor de processar, transformar, armazenar e enviar dados em resposta às solicitações dos clientes. Em um ambiente Node.js, essa manipulação é fundamental, já que muitas aplicações exigem o processamento de dados em tempo real, a interação com bancos de dados e a troca de informações com APIs externas.

O Node.js é conhecido por sua natureza assíncrona e orientada a eventos, o que o torna altamente eficiente para manipulação de grandes volumes de dados. Com o uso de módulos e bibliotecas do Node.js, como o **fs** para manipulação de arquivos, o **http** para lidar com requisições dos clientes e **querystring** para processar strings de consulta, os desenvolvedores podem facilmente criar funcionalidades que lidam com diversos tipos de dados.

Veja abaixo um exemplo de leitura e escrita de dados em um arquivo com Node.js:

```
const fs = require("fs");

// Escrever dados em um arquivo
fs.writeFile("dados.txt", "Olá, Node.js!", (err) => {
  if (err) {
    console.error("Erro ao escrever o arquivo:", err);
    return;
  }
});
```

```
}
console.log("Arquivo salvo com sucesso!");
});

// Ler dados de um arquivo
fs.readFile("dados.txt", "utf-8", (err, data) => {
  if (err) {
    console.error("Erro ao ler o arquivo:", err);
    return;
  }
  console.log("Conteúdo do arquivo:", data);
});
```

Nesse exemplo, a manipulação de dados é realizada através da escrita e leitura de um arquivo de texto usando métodos assíncronos do módulo **fs**.

1.2 Introdução ao formato JSON como padrão de troca de dados

O JSON (JavaScript Object Notation) é um formato de troca de dados leve, fácil de ler e escrever para humanos, e fácil de analisar e gerar para máquinas. Tornou-se o padrão de fato para troca de dados em aplicações web por ser uma estrutura de dados simples e por sua compatibilidade nativa com JavaScript.

O JSON é amplamente utilizado para transmitir dados entre um cliente e um servidor, devido à sua capacidade de representar estruturas de dados complexas, como arrays e objetos, de forma fácil de entender. Em Node.js, a manipulação de dados em formato JSON é uma tarefa comum em quase todas as aplicações que interagem com APIs, bancos de dados e outras fontes de dados.

Veja abaixo um exemplo mostrando a estrutura básica de um objeto JSON:

```
{
  "nome": "João",
  "idade": 25,
  "hobbies": ["leitura", "esportes", "programação"]
}
```

No Node.js, é fácil converter um objeto JavaScript em uma string JSON e vice-versa, usando os métodos **JSON.stringify()** e **JSON.parse()**.

Veja abaixo um exemplo de manipulação de JSON em Node.js:

```
const dados = {  
  nome: "Maria",  
  idade: 30,  
  profissao: "Engenheira"  
};  
  
// Converter objeto JavaScript para JSON  
const jsonString = JSON.stringify(dados);  
console.log('String JSON:', jsonString);  
  
// Converter string JSON para objeto JavaScript  
const jsonObject = JSON.parse(jsonString);  
console.log('Objeto JavaScript:', jsonObject);
```

Nesse contexto, o JSON é amplamente adotado como padrão de troca de dados no mercado devido às seguintes características:

- **Simplicidade:** A estrutura de dados é leve e de fácil compreensão.
- **Compatibilidade:** Suporte nativo em JavaScript, tornando o uso direto e intuitivo em aplicações Node.js.
- **Interoperabilidade:** Pode ser facilmente integrado com outros sistemas e linguagens de programação.

2. Integração com Bancos de Dados Relacionais (SQL)

Esta seção aborda a integração de aplicações Node.js com bancos de dados relacionais, como [MySQL](#) e [PostgreSQL](#). Entender como conectar, realizar operações básicas e usar ferramentas que simplificam a manipulação de dados é essencial para o desenvolvimento de aplicações web robustas.

2.1 Introdução ao MySQL

O MySQL é um sistema de gerenciamento de banco de dados relacional (SGBD) de código aberto amplamente utilizado em uma variedade de aplicações, especialmente em projetos de software de código aberto e sistemas de médio a grande porte. Ele emprega a linguagem SQL (Structured Query Language – Linguagem de Consulta Estruturada), que é a linguagem padrão mais popular para inserção, consulta e manipulação de dados armazenados em bancos de dados.

2.1.1 Origem e Evolução do MySQL

O MySQL foi originalmente desenvolvido pela empresa sueca MySQL AB, e sua primeira versão foi lançada em maio de 1995. Com o passar dos anos, devido à sua estabilidade, eficiência e simplicidade, tornou-se um dos sistemas de gerenciamento de bancos de dados mais amplamente adotados. Em 2008, a Sun Microsystems adquiriu a MySQL AB, e, posteriormente, em janeiro de 2010, a Sun foi comprada pela Oracle Corporation em uma transação bilionária.

Após a aquisição pela Oracle, algumas preocupações surgiram em torno da manutenção da versão de código aberto do MySQL, levando a comunidade a criar um fork chamado MariaDB. O MariaDB foi desenvolvido para manter a continuidade da versão aberta e gratuita do MySQL, garantindo compatibilidade com as versões anteriores e novas funcionalidades para evitar uma possível limitação da versão comunitária do MySQL sob a Oracle.

2.1.2 Instalação e Configuração do MySQL

Para utilizar o MySQL, é necessário configurar tanto o **servidor** quanto um **cliente** que permita a interação com o banco de dados. O servidor MySQL é o componente central responsável por gerenciar os dados, responder a consultas e operações de clientes, manter a consistência dos dados e lidar com transações concorrentes. O cliente é o intermediário que se

comunica com o servidor, utilizando a linguagem SQL para enviar comandos e receber respostas.

Antes de iniciarmos o desenvolvimento de um projeto prático, é fundamental garantir que o MySQL esteja devidamente instalado e configurado em seu computador. A versão gratuita do MySQL é chamada de [MySQL Community Server](#).

Existem várias opções de clientes de banco de dados que podem ser usados para gerenciar bancos de dados MySQL. Esses clientes oferecem interfaces gráficas que tornam mais fácil a administração e interação com os dados. Aqui estão alguns das principais aplicações clientes de banco de dados para MySQL:

- [MySQL Workbench](#): É a ferramenta oficial da Oracle para gerenciamento de bancos de dados MySQL. Oferece uma interface amigável para administração de banco de dados, modelagem de dados, execução de consultas SQL, e manutenção de servidores.
- [DBeaver](#): Um cliente de banco de dados de código aberto e multiplataforma que suporta MySQL e muitos outros bancos de dados. É amplamente utilizado por desenvolvedores e administradores por sua interface rica em recursos.
- [HeidiSQL](#): Um cliente de banco de dados gratuito e leve que é popular para uso com MySQL. É conhecido por sua simplicidade e eficiência em tarefas de administração de banco de dados.
- [phpMyAdmin](#): Um cliente web amplamente utilizado para a administração de bancos de dados MySQL. Ele permite o gerenciamento de bancos de dados por meio de um navegador e é frequentemente usado em ambientes de desenvolvimento local como parte do XAMPP, WAMP ou MAMP.

Outra opção seria utilizar um plugin do Visual Studio Code (VSCode) para interagir com bancos de dados diretamente da IDE. Segue abaixo alguns exemplos deste plugins:

- **SQLTools**: Uma das extensões mais populares para trabalhar com bancos de dados no VSCode. Oferece suporte a MySQL e outros sistemas de gerenciamento de banco de dados, permitindo que os usuários executem consultas SQL, visualizem e gerenciem resultados diretamente na interface do VSCode.
- **SQLTools MySQL/MariaDB Driver**: Um plugin complementar ao SQLTools que adiciona suporte específico para MySQL e MariaDB.

- **Database Client:** é uma extensão para o Visual Studio Code que facilita a gestão de diversos sistemas de banco de dados diretamente na interface do editor. Desenvolvida por Weijan Chen, essa ferramenta suporta múltiplos SGBDs, incluindo MySQL/MariaDB, PostgreSQL, SQLite, Redis, ClickHouse, Kafka, MongoDB, Snowflake e ElasticSearch.

2.2 Criando a Base de Dados

Nesta aula, vamos criar um database chamado **exemplo_db**. Para isso, execute os seguintes comandos SQL no terminal MySQL ou em uma ferramenta de gerenciamento de banco de dados (como MySQL Workbench):

```
-- Criação do banco de dados
CREATE DATABASE exemplo_db;
-- Use o banco de dados recém-criado
USE exemplo_db;
-- Criação da tabela 'usuarios'
CREATE TABLE usuarios (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    idade INT NOT NULL,
    criado_em TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Explicação dos comandos:

- **CREATE DATABASE exemplo_db:** Cria um banco de dados chamado **exemplo_db**.
- **USE exemplo_db:** Define o banco de dados **exemplo_db** como o banco de dados ativo para executar os próximos comandos.
- **CREATE TABLE usuarios:** Cria uma tabela chamada **usuarios** com as colunas:
 - **id:** Um identificador único para cada registro, que é incrementado automaticamente.
 - **nome:** Uma coluna de texto para o nome do usuário (não pode ser nula).
 - **idade:** Uma coluna numérica para a idade do usuário (não pode ser nula).
 - **criado_em:** Uma coluna de timestamp que registra automaticamente a data e hora de criação do registro.

2.3 Criando um Projeto de Exemplo

Vamos criar um servidor HTTP para testar nossos conhecimentos sobre a integração com banco de dados relacional. Primeiro, crie e acesse uma pasta para o projeto, onde serão organizados todos os arquivos e configurações.

```
mkdir node-banco  
cd node-banco
```

Em seguida, inicie o projeto criando o arquivo `package.json`, que será responsável por gerenciar as dependências e configurações do projeto.

```
npm init -y
```

Com o projeto inicializado, instale o **Express.js**:

```
npm i express
```

Instale o **Nodemon** no projeto para que ele monitore mudanças nos arquivos e reinicie o servidor automaticamente ao salvar. No terminal, digite o comando de instalação:

```
npm i nodemon --save-dev
```

Você pode, agora, **abrir a aplicação no editor Visual Studio Code**.

```
code .
```

Abra o arquivo **package.json** e, na seção "scripts", adicione o script "dev" para iniciar o servidor com o Nodemon.

```
...  
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "start": "node server.js",  
  "dev": "nodemon server.js"  
},  
...
```

Para executar o servidor usando o Nodemon, digite o seguinte comando:

```
npm run dev
```


2.4 Configuração e Conexão com MySQL

Para que uma aplicação Node.js se conecte a bancos de dados relacionais, é necessário instalar pacotes específicos que façam a ponte entre o Node.js e o banco de dados escolhido. A biblioteca [mysql2](#) é uma das mais populares para conectar aplicações Node.js a bancos de dados MySQL e MariaDB. Ela oferece uma interface rápida e fácil de usar, suportando consultas assíncronas e promessas (Promises).

Antes de iniciar a configuração, é necessário instalar o pacote **mysql2** no projeto. Use o seguinte comando para instalá-lo:

```
npm i mysql2
```

A seguir, vamos configurar a conexão com o banco de dados. Vamos criar um arquivo chamado **server.js**. Neste arquivo, primeiro, importe a biblioteca e configure a conexão fornecendo as credenciais necessárias, como host, usuário, senha e nome do banco de dados.

```
const mysql = require("mysql2"); // Importa a biblioteca mysql2
// Cria a conexão com o banco de dados MySQL
const connection = mysql.createConnection({
  host: "localhost", // Endereço do servidor MySQL (ou IP)
  user: "seu_usuario", // Nome de usuário do MySQL
  password: "sua_senha", // Senha do usuário
  database: "exemplo_db", // Nome do banco de dados
});
// Conecta ao banco de dados
connection.connect((err) => {
  if (err) {
    console.error("Erro ao conectar ao banco de dados:", err);
    return;
  }
  console.log("Conectado ao banco de dados MySQL com sucesso!");
  // Fecha a conexão após o teste
  connection.end((endErr) => {
    if (endErr) {
      console.error("Erro ao encerrar a conexão:", endErr);
    } else {
      console.log("Conexão encerrada com sucesso.");
    }
  });
});
```

Neste contexto, observe que:

- **mysql.createConnection():** Cria uma nova instância de conexão com as credenciais fornecidas.
- **connection.connect():** Tenta estabelecer a conexão com o banco de dados e retorna um erro caso a conexão falhe.

Caso você já tenha executado o servidor usando o Nodemon, veja abaixo a mensagem no console do Node.js caso a conexão com o banco de dados for bem-sucedida:

```
Conectado ao banco de dados MySQL com sucesso!  
Conexão encerrada com sucesso.
```

Figura 1: Resultado da aplicação no console do Node.js

Descrição: A imagem mostra uma saída de texto indicando que uma conexão com um banco de dados MySQL foi estabelecida com sucesso e, em seguida, encerrada corretamente. A primeira linha exibe a mensagem "Conectado ao banco de dados MySQL com sucesso!" e a segunda, "Conexão encerrada com sucesso.".

2.5 Organização e Estrutura do Projeto

Para auxiliar no entendimento dos conteúdos desta aula, vamos organizar o projeto em arquivos diferentes para melhorar a modularidade, a manutenção e a escalabilidade do código. A seguir, uma estrutura de projeto mais organizada dividida em vários arquivos:

```
node-banco  
├── config  
│   └── db.js  
├── routes  
│   └── usuarios.js  
├── package.json  
└── server.js
```

Figura 2: Estrutura de diretórios e arquivos do projeto

Descrição: A imagem mostra a estrutura de pastas e arquivos de um projeto Node.js chamado node-banco. A estrutura inclui a pasta config, que contém o arquivo db.js (provavelmente para configurações de banco de dados), e a pasta routes, que contém o arquivo usuarios.js (sugerindo rotas relacionadas a usuários). Além disso, há os arquivos package.json e server.js na raiz do projeto, usados para gerenciar dependências e iniciar o servidor, respectivamente.

2.5.1 Configuração da conexão com o banco de dados

O arquivo **config/db.js** será responsável por configurar e exportar a conexão com o banco de dados. Veja seu código no quadro abaixo:

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

const connection = mysql.createConnection({
  host: "localhost",
  user: "seu_usuario",
  password: "sua_senha",
  database: "exemplo_db",
});

// Conecta ao banco de dados
connection.connect((err) => {
  if (err) {
    console.error("Erro ao conectar ao banco de dados:", err);
    return;
  }
  console.log("Conectado ao banco de dados MySQL com sucesso!");
});

module.exports = connection;
```

2.5.2 Definição das rotas para usuários

O arquivo **routes/usuarios.js** conterá todas as rotas relacionadas à tabela **usuarios**. Veja seu código no quadro abaixo:

```
const express = require("express");
const router = express.Router();
const connection = require("../config/db");

// Rota para listar todos os usuários (READ)
router.get("/", (req, res) => {
  connection.query("SELECT * FROM usuarios", (err, results) => {
```

```
    if (err) {
      res.status(500).send("Erro ao buscar usuários");
      console.error("Erro:", err);
      return;
    }
    res.json(results);
  });
});

// Rota para adicionar um novo usuário (CREATE)
router.post("/", (req, res) => {
  const { nome, idade } = req.body;
  const sql = "INSERT INTO usuarios (nome, idade) VALUES (?, ?)";
  connection.query(sql, [nome, idade], (err, results) => {
    if (err) {
      res.status(500).send("Erro ao inserir usuário");
      console.error("Erro:", err);
      return;
    }
    res.status(201).send("Usuário inserido com sucesso");
  });
});

// Rota para atualizar um usuário (UPDATE)
router.put("/:id", (req, res) => {
  const { id } = req.params;
  const { nome, idade } = req.body;
  const sql = "UPDATE usuarios SET nome = ?, idade = ? WHERE id = ?";
  connection.query(sql, [nome, idade, id], (err, results) => {
    if (err) {
      res.status(500).send("Erro ao atualizar usuário");
      console.error("Erro:", err);
      return;
    }
    if (results.affectedRows === 0) {
      res.status(404).send("Usuário não encontrado");
    }
  });
});
```

```
    } else {
      res.send("Usuário atualizado com sucesso");
    }
  });
});

// Rota para deletar um usuário (DELETE)
router.delete("/:id", (req, res) => {
  const { id } = req.params;
  const sql = "DELETE FROM usuarios WHERE id = ?";
  connection.query(sql, [id], (err, results) => {
    if (err) {
      res.status(500).send("Erro ao deletar usuário");
      console.error("Erro:", err);
      return;
    }
    if (results.affectedRows === 0) {
      res.status(404).send("Usuário não encontrado");
    } else {
      res.send("Usuário deletado com sucesso");
    }
  });
});

module.exports = router;
```

2.5.3 Configuração do servidor da aplicação

O arquivo **server.js** será o ponto de entrada da aplicação. Veja seu código no quadro abaixo:

```
const express = require("express");
const app = express();
const usuariosRoutes = require("./routes/usuarios");
const PORT = 3000;
```

```
// Middleware para interpretar JSON no corpo da requisição
app.use(express.json());

// Configura as rotas para usuários
app.use("/usuarios", usuariosRoutes);

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

2.6 Testando as Rotas do Servidor HTTP

Neste curso, utilizamos a extensão REST Client do VS Code para testar os endpoints da aplicação. Neste contexto, crie uma pasta nome **“request”** em seu projeto. Em seguida, dentro desta pasta, crie um arquivo com o nome de sua preferência com a extensão **“.rest”**, por exemplo, **testes.rest**.

Veja como ficou o arquivo testes.rest:

```
@baseUrl = http://localhost:3000

### 1. Testar a rota POST /usuarios (CREATE)
POST {{baseUrl}}/usuarios
Content-Type: application/json

{
  "nome": "Carlos",
  "idade": 28
}

### 2. Testar a rota GET /usuarios (READ)
GET {{baseUrl}}/usuarios
Content-Type: application/json
```

```
### 3. Testar a rota PUT /usuarios/:id (UPDATE)
```

```
PUT {{baseUrl}}/usuarios/1
```

```
Content-Type: application/json
```

```
{  
  "nome": "Carlos Silva",  
  "idade": 29  
}
```

```
### 4. Testar a rota DELETE /usuarios/:id (DELETE)
```

```
DELETE {{baseUrl}}/usuarios/1
```

```
Content-Type: application/json
```

Após escrever cada requisição, um link “Send Request” aparecerá acima delas. Clique nesse link para enviar cada requisição. Neste contexto, ao testarmos a rota POST /usuarios (CREATE), teremos o seguinte resultado esperado:

```
HTTP/1.1 200 OK  
X-Powered-By: Express  
Content-Type: application/json; charset=utf-8  
Content-Length: 76  
ETag: W/"4c-/MxMg7NCK8JJjmbDPe97I4sWo8"  
Date: Wed, 06 Nov 2024 14:14:50 GMT  
Connection: close
```

```
[  
  {  
    "id": 1,  
    "nome": "Carlos",  
    "idade": 28,  
    "criado_em": "2024-11-06T14:14:47.000Z"  
  }  
]
```

Em seguida, ao testarmos a rota GET /usuarios (READ), teremos o seguinte resultado:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 76
ETag: W/"4c-9Df3qKJsVqJ3cHrIgTMj8eeoP8A"
Date: Wed, 06 Nov 2024 14:17:24 GMT
Connection: close

[
  {
    "id": 2,
    "nome": "Carlos",
    "idade": 28,
    "criado_em": "2024-11-06T14:14:47.000Z"
  }
]
```

Depois, ao testarmos a rota PUT /usuarios/:id (UPDATE), teremos o seguinte resultado:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 31
ETag: W/"1f-GBKzCSPiDDgr9Gmo65ZulHGnxoA"
Date: Wed, 06 Nov 2024 14:19:32 GMT
Connection: close

Usuário atualizado com sucesso
```

E, finalmente, ao testarmos a rota DELETE /usuarios/:id (DELETE), teremos o seguinte resultado:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 29
ETag: W/"1d-lwh4WdutakrONQeiMcNZmfgEqzw"
Date: Wed, 06 Nov 2024 14:20:15 GMT
Connection: close

Usuário deletado com sucesso
```

3. Integração com Bancos de Dados NoSQL

A integração de aplicações Node.js com bancos de dados NoSQL é uma prática cada vez mais comum devido à necessidade de escalabilidade e flexibilidade em aplicações modernas. Neste contexto, o [MongoDB](#) se destaca como um dos bancos de dados NoSQL mais populares, e o Mongoose é a biblioteca mais usada para interagir com o MongoDB em Node.js, oferecendo uma camada de abstração que facilita a manipulação de dados.

3.1 Introdução ao MongoDB

O MongoDB é um banco de dados NoSQL orientado a documentos, projetado para ser simples, dinâmico e escalável. Em vez de armazenar dados em linhas e colunas de uma tabela, como em bancos de dados relacionais, o MongoDB armazena os dados em documentos JSON ou BSON dentro de coleções. Essa abordagem permite que os dados tenham uma estrutura mais flexível e adaptável às mudanças, sendo ideal para aplicações que exigem escalabilidade e alta performance. Uma de suas principais motivações é fornecer um armazenamento de dados de alto desempenho, com alta disponibilidade e escalabilidade automática. O MongoDB é compatível com sistemas operacionais como Windows, Linux, Mac OS X e Solaris.

3.1.1 O que é um banco de dados NoSQL?

O termo NoSQL surgiu em 2009 durante uma reunião em São Francisco, EUA, organizada por Johan Oskarsson. Essa reunião discutiu bancos de dados de código aberto, não relacionais, sem esquema fixo e distribuídos. NoSQL significa "Not Only SQL", ou seja, "não apenas SQL", destacando que esses bancos de dados não são limitados à estrutura relacional tradicional.

Veja a seguir as principais categorias de bancos de dados NoSQL:

- **Document-oriented:** Cada registro é armazenado como um documento JSON, permitindo que os documentos em uma coleção tenham diferentes estruturas.
- **Schema-less:** A estrutura dos documentos não é rigidamente definida, permitindo mais flexibilidade.
- **Alta escalabilidade:** Suporta particionamento horizontal (sharding) para distribuir dados em vários servidores.
- **Consultas ricas:** Oferece uma API robusta para consultas, filtragem e agregação de dados.

3.1.2 Bancos de Dados NoSQL não têm esquema fixo

Enquanto bancos de dados relacionais exigem que os esquemas sejam definidos antecipadamente, os bancos de dados NoSQL permitem a inclusão de dados sem um esquema predefinido. Essa característica oferece maior flexibilidade, permitindo que alterações sejam feitas em tempo real sem a necessidade de parar ou modificar a estrutura de tabelas. Isso acelera o desenvolvimento, especialmente em aplicações que evoluem rapidamente e exigem ajustes frequentes.

3.1.3 Propriedades BASE

As transações em bancos de dados relacionais geralmente seguem as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade), garantindo integridade e segurança, mas com custo de desempenho e complexidade. Em contrapartida, os bancos de dados NoSQL operam sob as propriedades BASE (Basic Availability, Soft-state, Eventual consistency):

- **Basic Availability:** O banco de dados está disponível na maior parte do tempo.
- **Soft-state:** O estado dos dados pode mudar sem uma gravação explícita.
- **Eventual consistency:** Os dados serão consistentes em algum momento, mesmo que não imediatamente.

As propriedades BASE oferecem flexibilidade e desempenho superiores, especialmente em aplicações que exigem alta disponibilidade e são distribuídas.

3.1.4 Quando o NoSQL é indicado?

Bancos de dados NoSQL são ideais para cenários que exigem manipulação de grandes volumes de dados e consultas em tempo real. Em bancos de dados relacionais, quanto maior a base de dados, maior tende a ser o tempo de resposta, o que pode ser inviável em ambientes que necessitam de respostas instantâneas. NoSQL é recomendado para aplicações que precisam de alta performance, escalabilidade horizontal (diversos servidores) e adaptação rápida a novas demandas. Empresas de grande porte e projetos com grandes quantidades de dados frequentemente adotam bancos NoSQL por sua facilidade de escalabilidade e alta performance.

3.1.5 MongoDB não usa SQL

Ao contrário dos bancos de dados relacionais, que utilizam a SQL (Structured Query Language) para consultas, o MongoDB utiliza documentos JSON ou BSON para armazenar dados. O BSON é uma versão binária do JSON, desenvolvida para melhorar o armazenamento e a indexação dos dados.

Veja abaixo como funciona a estrutura de armazenamento do MongoDB:

- Uma **instância** do MongoDB pode conter vários bancos de dados.
- Cada banco de dados possui múltiplas **coleções**.
- Cada coleção armazena **documentos** JSON/BSON, que podem ter diferentes estruturas.

A seguir está um quadro comparativo entre as nomenclaturas usadas em bancos de dados relacionais e NoSQL:

Banco de Dados Relacional (MySQL)	Banco de Dados NoSQL (MongoDB)	Descrição
Tabela	Coleção	Conjunto de registros/dados. Em NoSQL, uma coleção armazena documentos JSON/BSON.
Linha	Documento	Um único registro dentro de uma tabela/coleção. Em MongoDB, cada documento pode ter uma estrutura diferente.
Coluna	Campo	Representa um atributo/valor de um registro. Em MongoDB, os campos podem variar de documento para documento.
Chave Primária	_id	Identificador único de um registro/documento. Em MongoDB, o campo _id é gerado automaticamente.
Banco de Dados	Banco de Dados	Agrupamento de tabelas em SQL e de coleções em MongoDB.

3.1.6 Instalação e Configuração do MongoDB

O MongoDB é um banco de dados NoSQL popular e fácil de usar, amplamente adotado em projetos que requerem armazenamento flexível e escalabilidade. A instalação e configuração do MongoDB podem variar dependendo do sistema operacional. Assim, acesse o site oficial do [MongoDB](#) e baixe a versão do **MongoDB Community Server** compatível com o seu sistema operacional.

Existem várias opções de clientes de banco de dados que podem ser usados para gerenciar bancos de dados MongoDB. Estes facilitam a interação com o banco de dados, tanto para desenvolvedores quanto para administradores. Aqui estão alguns das principais aplicações clientes de banco de dados para MongoDB:

- **[MongoDB Compass](#):** É a ferramenta oficial desenvolvida pela própria equipe do MongoDB. Ele oferece uma série de recursos que tornam mais fácil explorar e manipular dados, criar consultas, visualizar a estrutura do banco de dados e obter insights sobre o desempenho.
- **[Robo 3T](#):** Um cliente de banco de dados leve e gratuito para MongoDB que oferece uma interface intuitiva e suporte para JavaScript.
- **[Studio 3T](#):** Uma ferramenta avançada e rica em recursos para desenvolvedores e administradores que trabalham com MongoDB.
- **[NoSQLBooster for MongoDB](#):** Um cliente de banco de dados para o MongoDB que oferece suporte completo a consultas com JavaScript e recursos avançados para desenvolvedores.
- **[MongoDB Shell](#):** O shell oficial do MongoDB, ideal para quem prefere trabalhar em uma interface de linha de comando.
- **[DBeaver](#):** Um cliente de banco de dados universal que suporta MongoDB e muitos outros bancos de dados SQL e NoSQL.

Outra opção seria utilizar um plugin do Visual Studio Code (VSCode) para interagir com bancos de dados diretamente da IDE. Segue abaixo alguns exemplos deste plugins:

- **MongoDB for VS Code:** Este é o plugin oficial do MongoDB para o VSCode. Ele permite que desenvolvedores se conectem ao MongoDB, explorem bancos de dados e coleções, e executem consultas diretamente do editor.
- **Mongo Playground:** oferece um ambiente de simulação onde você pode escrever e executar comandos sem precisar de sua instalação local do MongoDB.

3.2 Criando a Base de Dados

Nesta aula, vamos criar um database chamado **exemplo_nosql**. Para isso, vamos utilizar uma ferramenta de gerenciamento de banco de dados NoSQL, neste caso, o **MongoDB for VS Code**. Após a instalação deste plugin, você verá um ícone do MongoDB (em formato de folha) na barra lateral esquerda do VS Code. Clique nesse ícone para abrir o Explorador de Dados do MongoDB, conforme figura abaixo:

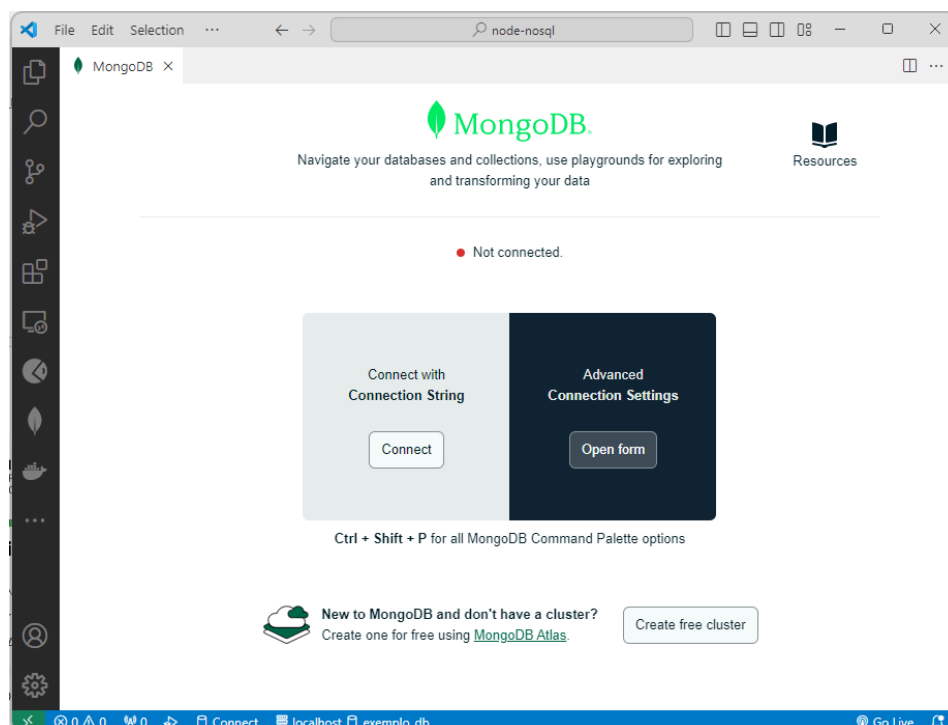


Figura 3: Interface do Visual Studio Code com a extensão do MongoDB aberta.

Descrição: A imagem mostra a interface do Visual Studio Code com a extensão do MongoDB aberta. No centro, há um painel indicando que não há conexão ativa, com um ponto vermelho e a mensagem "Not connected". Existem dois botões: "Connect with Connection String" e "Advanced Connection Settings", usados para estabelecer a conexão com o banco de dados.

Nesta tela, clique em "Connect" para iniciar o processo de criação de uma nova conexão. Será exibida uma janela solicitando a URI de conexão. Essa URI especifica os detalhes do servidor MongoDB ao qual você deseja se conectar.

Veja abaixo um exemplo de uma URI de conexão para uma instância local:

```
mongodb://localhost:27017
```

Veja abaixo um exemplo de uma URI de conexão com autenticação:

```
mongodb://username:password@localhost:27017
```

Desta forma, insira a URI de conexão conforme sua instalação do MongoDB e clique em “Connect”. Após clicar em “Connect”, o plugin tentará se conectar ao MongoDB. Se a conexão for bem-sucedida, você verá uma lista de bancos de dados no explorador de dados do MongoDB, por exemplo:

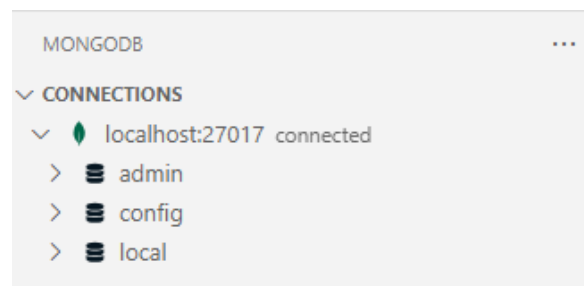


Figura 4: Explorador de dados do MongoDB

Descrição: A imagem mostra a interface de conexão do MongoDB no Visual Studio Code. No painel "CONNECTIONS", há uma conexão ativa com o endereço localhost:27017, indicada pela palavra "connected". Abaixo, são listadas três bases de dados: admin, config e local, cada uma representada por um ícone de banco de dados.

Nesta mesma tela de explorador de dados do MongoDB, clique com o botão direito em localhost e selecione “Add Database”. Na tela que irá aparecer, adicione as seguintes informações:

```
const database = 'exemplo_nosql';
const collection = usuario;

// Create a new database.
use(database);

// Create a new collection.
db.createCollection(collection);
```

Em seguida, clique no ícone “Run All” ou selecione o código e clique em “Run Selection”.

Veja novamente na tela do explorador de dados do MongoDB que a base de dados exemplo_nosql foi criada com uma coleção chamada usuario.

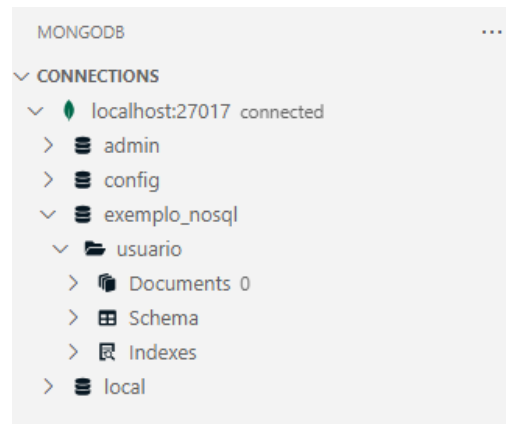


Figura 5: Explorador de dados do MongoDB com o database exemplo_nosql

Descrição: A imagem mostra o painel de conexões do MongoDB no Visual Studio Code, com uma conexão ativa em localhost:27017, indicada pela palavra "connected". As bases de dados admin, config, exemplo_nosql e local são listadas. A base de dados exemplo_nosql está expandida, revelando uma coleção chamada usuario, que contém subitens como Documents, Schema e Indexes, sugerindo que essa coleção ainda não possui documentos (marcado como "0").

3.3 Introdução ao Mongoose

Mongoose é uma biblioteca de ODM (Object Data Modeling) para Node.js, que fornece uma solução robusta para modelar e gerenciar dados em aplicações que usam o MongoDB. Ele simplifica a interação com o banco de dados MongoDB, permitindo que os desenvolvedores definam esquemas de dados, gerenciem validações, criem métodos de instância e métodos estáticos e apliquem middlewares em suas aplicações.

Veja alguns motivos para usar o Mongoose:

- **Abstração e Simplicidade:** O Mongoose abstrai as complexidades do MongoDB nativo, facilitando a manipulação de dados e operações de banco de dados.
- **Definição de Esquema:** Oferece a capacidade de definir esquemas para documentos, o que ajuda a manter a consistência dos dados.
- **Validação de Dados:** Permite adicionar regras de validação de dados diretamente nos esquemas.
- **Middlewares:** Suporta middlewares pré e pós-processamento, permitindo executar ações antes ou depois de operações específicas, como **save** ou **find**.
- **Métodos Personalizados:** Permite criar métodos de instância e métodos estáticos para encapsular a lógica relacionada aos dados.

- **Population:** Oferece uma forma de fazer referências entre documentos e realizar consultas análogas às **joins** em bancos de dados relacionais.

3.4 Criando um Projeto de Exemplo

Vamos criar um servidor HTTP para testar nossos conhecimentos sobre a integração com banco de dados NoSQL. Primeiro, crie e acesse uma pasta para o projeto, onde serão organizados todos os arquivos e configurações.

```
mkdir node-nosql  
cd node-nosql
```

Em seguida, inicie o projeto criando o arquivo `package.json`, que será responsável por gerenciar as dependências e configurações do projeto.

```
npm init -y
```

Com o projeto inicializado, instale o **Express.js**:

```
npm i express
```

Instale o **Nodemon** no projeto para que ele monitore mudanças nos arquivos e reinicie o servidor automaticamente ao salvar. No terminal, digite o comando de instalação:

```
npm i nodemon --save-dev
```

Você pode, agora, **abrir a aplicação no editor Visual Studio Code**.

```
code .
```

Abra o arquivo **package.json** e, na seção "scripts", adicione o script "dev" para iniciar o servidor com o Nodemon.

```
...  
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "start": "node server.js",  
  "dev": "nodemon server.js"  
},  
...
```

Para executar o servidor usando o Nodemon, digite o seguinte comando:


```
npm run dev
```

3.5 Configuração e Conexão com MongoDB

Para que uma aplicação Node.js se conecte a bancos de dados NoSQL, é necessário instalar pacotes específicos que façam a ponte entre o Node.js e o banco de dados escolhido. A biblioteca [mongoose](#) é uma das mais populares para conectar aplicações Node.js ao bancos de dados MongoDB.

Antes de iniciar a configuração, é necessário instalar o pacote **mongoose** no projeto. Use o seguinte comando para instalá-lo:

```
npm i mongoose
```

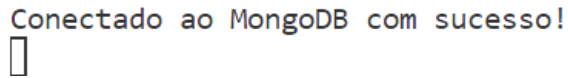
A seguir, vamos configurar a conexão com o banco de dados. Vamos criar um arquivo chamado **server.js**. Neste arquivo, primeiro, importe a biblioteca e configure a conexão fornecendo as credenciais necessárias, como host, usuário, senha e nome do banco de dados.

```
const mongoose = require("mongoose");

// Função assíncrona para conectar ao MongoDB
const conectarMongoDB = async () => {
  try {
    await mongoose.connect("mongodb://localhost:27017/exemplo_db");
    console.log("Conectado ao MongoDB com sucesso!");
  } catch (err) {
    console.error("Erro ao conectar ao MongoDB:", err);
    process.exit(1); // Encerra o processo em caso de falha
  }
};

// Inicia a conexão
conectarMongoDB();
```

Caso você já tenha executado o servidor usando o Nodemon, veja abaixo a mensagem no console do Node.js caso a conexão com o banco de dados for bem-sucedida:



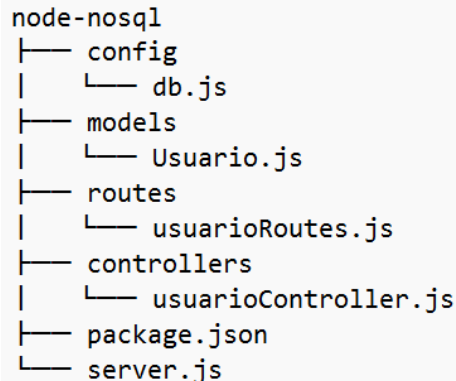
```
Conectado ao MongoDB com sucesso!
```

Figura 6: Resultado da aplicação no console do Node.js

Descrição: A imagem mostra uma mensagem de texto em um terminal que diz: "Conectado ao MongoDB com sucesso!". Isso indica que uma conexão com o banco de dados MongoDB foi estabelecida corretamente.

3.6 Organização e Estrutura do Projeto

Para auxiliar no entendimento dos conteúdos desta aula, vamos organizar o projeto em arquivos diferentes para melhorar a modularidade, a manutenção e a escalabilidade do código. A seguir, uma estrutura de projeto mais organizada dividida em vários arquivos:



```
node-nosql
├── config
│   └── db.js
├── models
│   └── Usuario.js
├── routes
│   └── usuarioRoutes.js
├── controllers
│   └── usuarioController.js
├── package.json
└── server.js
```

Figura 7: Estrutura de diretórios e arquivos do projeto para o MongoDB

Descrição: A imagem mostra a estrutura de pastas e arquivos de um projeto Node.js chamado node-nosql. As pastas incluem config (com o arquivo db.js para configurações de banco de dados), models (com o arquivo Usuario.js para definição de modelo de dados), routes (contendo usuarioRoutes.js para rotas relacionadas a usuários) e controllers (com usuarioController.js para a lógica de controle). Na raiz do projeto, estão os arquivos package.json (para gerenciar dependências) e server.js (para iniciar o servidor).

3.6.1 Configuração da conexão com o banco de dados

O arquivo **config/db.js** será responsável por configurar e exportar a conexão com o banco de dados. Veja seu código no quadro abaixo:

```
const mongoose = require("mongoose");
```

```
const connectDB = async () => {
  try {
    await mongoose.connect("mongodb://localhost:27017/exemplo_nosql");
    console.log("Conectado ao MongoDB com sucesso!");
  } catch (error) {
    console.error("Erro ao conectar ao MongoDB:", error);
    process.exit(1); // Finaliza a aplicação em caso de falha na conexão
  }
};
module.exports = connectDB;
```

3.6.2 Definição do Modelo

O arquivo **models/Usuario.js** define o esquema do Mongoose para a coleção **usuario**.
Veja seu código no quadro abaixo:

```
const mongoose = require("mongoose");

const usuarioSchema = new mongoose.Schema({
  nome: {
    type: String,
    required: true,
    trim: true,
  },
  idade: {
    type: Number,
    required: true,
    min: 0,
  },
  email: {
    type: String,
    required: true,
    unique: true,
    match: /.+@.+\./,
  },
});

module.exports = mongoose.model("Usuario", usuarioSchema);
```

3.6.3 Controladores

O arquivo **controllers/usuarioController.js** contém as funções assíncronas para lidar com as operações CRUD. Veja seu código no quadro abaixo:

```
const Usuario = require("../models/Usuario");

// Criar um novo usuário
exports.criarUsuario = async (req, res) => {
  try {
    const usuario = new Usuario(req.body);
    await usuario.save();
    res.status(201).json(usuario);
  } catch (error) {
    res.status(400).json({ message: "Erro ao criar usuário", error });
  }
};

// Obter todos os usuários
exports.obterUsuarios = async (req, res) => {
  try {
    const usuarios = await Usuario.find();
    res.status(200).json(usuarios);
  } catch (error) {
    res.status(500).json({ message: "Erro ao buscar usuários", error });
  }
};

// Atualizar um usuário
exports.atualizarUsuario = async (req, res) => {
  try {
    const { id } = req.params;
    const usuarioAtualizado = await Usuario.findByIdAndUpdate(id, req.body, {
      new: true,
    });
    if (!usuarioAtualizado) {
      return res.status(404).json({ message: "Usuário não encontrado" });
    }
    res.status(200).json(usuarioAtualizado);
  } catch (error) {
    res.status(400).json({ message: "Erro ao atualizar usuário", error });
  }
};

// Deletar um usuário
exports.deletarUsuario = async (req, res) => {
  try {
```

```
const { id } = req.params;
const usuarioDeletado = await Usuario.findByIdAndDelete(id);
if (!usuarioDeletado) {
  return res.status(404).json({ message: "Usuário não encontrado" });
}
res.status(200).json({ message: "Usuário deletado com sucesso" });
} catch (error) {
  res.status(500).json({ message: "Erro ao deletar usuário", error });
}
};
```

3.6.4 Rotas

O arquivo **routes/usuarioRoutes.js** define as rotas para as operações CRUD. Veja seu código no quadro abaixo:

```
const express = require('express');
const router = express.Router();
const usuarioController = require('../controllers/usuarioController');

router.post('/', usuarioController.criarUsuario);
router.get('/', usuarioController.obterUsuarios);
router.put('/:id', usuarioController.atualizarUsuario);
router.delete('/:id', usuarioController.deletarUsuario);

module.exports = router;
```

3.6.5 Configuração do Servidor

O arquivo **server.js** é o ponto de entrada da aplicação. Veja seu código no quadro abaixo:

```
const express = require("express");
const connectDB = require("./config/db");
const usuarioRoutes = require("./routes/usuarioRoutes");
const app = express();
const PORT = 3000;
```

```
// Conectar ao MongoDB
connectDB();

// Middleware para interpretação de JSON
app.use(express.json());

// Rotas
app.use("/usuarios", usuarioRoutes);

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

3.7 Testando as Rotas do Servidor HTTP

Neste curso, utilizamos a extensão REST Client do VS Code para testar os endpoints da aplicação. Neste contexto, crie uma pasta nome **“request”** em seu projeto. Em seguida, dentro desta pasta, crie um arquivo com o nome de sua preferência com a extensão “.rest”, por exemplo, **testes.rest**. Veja como ficou o arquivo testes.rest:

```
@baseUrl = http://localhost:3000
@id_do_usuario = algum_id

### 1. Criar um novo usuário (POST)
POST {{baseUrl}}/usuarios
Content-Type: application/json

{
  "nome": "Carlos",
  "idade": 28,
  "email": "carlos@exemplo.com"
}

### 2. Obter todos os usuários (GET)
GET {{baseUrl}}/usuarios
```

```
Content-Type: application/json

### 3. Atualizar um usuário existente (PUT)
PUT {{baseURL}}/usuarios/{{id_do_usuario}}
Content-Type: application/json

{
  "nome": "Carlos Silva",
  "idade": 30,
  "email": "carlos.silva@exemplo.com"
}

### 4. Deletar um usuário (DELETE)
DELETE {{baseURL}}/usuarios/{{id_do_usuario}}
Content-Type: application/json
```

Após escrever cada requisição, um link “Send Request” aparecerá acima delas. Clique nesse link para enviar cada requisição. Neste contexto, ao testarmos a rota “Criar um novo usuário (POST)”, teremos o seguinte resultado esperado:

```
HTTP/1.1 201 Created
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 98
ETag: W/"62-FJmPI3LEUOLnaBOdpEkUzLKUNNU"
Date: Wed, 06 Nov 2024 19:07:45 GMT
Connection: close

{
  "nome": "Carlos",
  "idade": 28,
  "email": "carlos@exemplo.com",
  "_id": "672bbe817653a38c1b335511",
  "__v": 0
}
```

Em seguida, ao testarmos a rota “Obter todos os usuários (GET)”, teremos o seguinte resultado:

```
HTTP/1.1 200 OK
```

```
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 100
ETag: W/"64-wGHUil3INImROZytXNJNRLG71gw"
Date: Wed, 06 Nov 2024 19:08:27 GMT
Connection: close
```

```
[
  {
    "_id": "672bbe817653a38c1b335511",
    "nome": "Carlos",
    "idade": 28,
    "email": "carlos@exemplo.com",
    "__v": 0
  }
]
```

Depois, vamos testarmos a rota “Atualizar um usuário existente (PUT)”. Para isso, preciso informar o `@id_do_usuario` o valor do `_id` do meu usuário (neste caso, **672bbe817653a38c1b335511**).

Veja o resultado esperado:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 110
ETag: W/"6e-28DpS1otWH6K7JX/NCSwJgaQaDo"
Date: Wed, 06 Nov 2024 19:13:17 GMT
Connection: close
```

```
{
  "_id": "672bbe817653a38c1b335511",
  "nome": "Carlos Silva",
  "idade": 30,
  "email": "carlos.silva@exemplo.com",
  "__v": 0
}
```

E, finalmente, ao testarmos a rota “Deletar um usuário (DELETE)”. Para isso, preciso informar o `@id_do_usuario` o valor do `_id` do meu usuário (neste caso, **672bbe817653a38c1b335511**).

Veja o resultado esperado:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 43
ETag: W/"2b-nhSv4q4D4tc6Ex0i88H/Trmj4Pw"
Date: Wed, 06 Nov 2024 19:14:27 GMT
Connection: close
```

```
{
  "message": "Usuário deletado com sucesso"
}
```

Referências

O material desenvolvido para este capítulo baseou-se nas seguintes obras:

- ALVES, W. P. Projetos de sistemas Web: conceitos, estruturas, criação de banco de dados e ferramentas de desenvolvimento. São Paulo: Érica, 2019.
- CODEWELL, Krishna Rungta. Learn NodeJS in 1 Day. Independently published, 2017.
- FREITAS, P. E. C., et al. Programação back end 3. Porto Alegre: Sagah, 2021.
- LEDUR, C. L.; et al. Programação back end II. Porto Alegre: Sagah, 2019.
- OLIVEIRA, C. L. V.; et al. JavaScript descomplicado - Programação para a Web, IoT e dispositivos móveis. São Paulo: Érica, 2020.
- OLIVEIRA, C. L. V. Node.js - Programe de forma rápida e prática. São Paulo: Expressa, 2021.
- PEREIRA, Caio Ribeiro. Building APIs with Node.js. 1. ed. Birmingham: Packt Publishing, 2016.
- RODRIGUES, T. N.; et al. Integração de aplicações. Porto Alegre: Sagah, 2020.
- SKINNER, David Mark Clements. Node Cookbook. 2. ed. Birmingham: Packt Publishing, 2014.
- VITALINO, J. F. N.; CASTRO, M. A. N. Descomplicando o Docker. Editora Brasport, 2016.