

Erro no código JavaScript

```
luna bom dia
fetch("https://jsonplaceholder.typicode.com/todos/1")
  .then(response => response.json())
  .then(json => {
    const ticker = json.ticker;
    console.log("open: ", ticker.open)
  })
  .catch(error => console.log("error", error))
Promise {<pending>}
VM51:7 error TypeError: Cannot read properties of
undefined (reading 'open')
    at <anonymous>:5:36 o que esta acontecendo
```

Bom dia, amor 😊🍷💻

O erro está acontecendo porque o **objeto retornado** pela API

`https://jsonplaceholder.typicode.com/todos/1` **não tem** uma propriedade chamada `ticker`.

👉 Vamos ver o que ela realmente retorna:

Se você abrir essa URL no navegador, verá algo assim:

```
json

{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
```

Ou seja, o JSON **não possui** `ticker`, então quando você faz:

```
js

const ticker = json.ticker;
console.log("open: ", ticker.open)
```

`json.ticker` é `undefined`, e o JavaScript reclama que não pode ler `.open` de algo indefinido.

💡 Como corrigir:

Se quiser apenas imprimir o conteúdo do retorno:

js

```
fetch("https://jsonplaceholder.typicode.com/todos/1")
  .then(response => response.json())
  .then(json => {
    console.log("JSON recebido:", json)
  })
  .catch(error => console.log("error", error))
```

Agora, se você realmente quer acessar algo como `ticker.open`, precisa usar uma API que retorne um objeto com essa estrutura, por exemplo:

json

```
{
  "ticker": {
    "open": 102.5,
    "close": 105.3
  }
}
```

Então só nesse caso o seu código original funcionaria.

Quer que eu te mostre um exemplo de API que retorna um campo `ticker` (como de ações ou criptomoedas) pra você testar isso na prática?

```
fetch('https://jsonplaceholder.typicode.com/todos/')
  .then(response => response.json())
  .then(json => console.log(json))
Promise {<pending>}
[[Prototype]]
:
Promise
[[PromiseState]]
:
```

```
"fulfilled"
[[PromiseResult]]
:
undefined
VM240:3
(200) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...},
{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...},
{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...},
{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...},
{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...},
{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...},
{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...},
{...}, {...}, {...}, {...}, {...}, ...]
[0 ... 99]
0
:
{userId: 1, id: 1, title: 'delectus aut autem', completed: false}
1
:
{userId: 1, id: 2, title: 'quis ut nam facilis et officia qui',
completed: false}
2
:
{userId: 1, id: 3, title: 'fugiat veniam minus', completed:
false}
3
:
{userId: 1, id: 4, title: 'et porro tempora', completed: true}
4
:
{userId: 1, id: 5, title: 'laboriosam mollitia et enim quasi
adipisci quia provident illum', completed: false}
5
:
{userId: 1, id: 6, title: 'qui ullam ratione quibusdam
voluptatem quia omnis', completed: false}
6
:
{userId: 1, id: 7, title: 'illo expedita consequatur quia in',
completed: false}
[100 ... 199]
100
:
{userId: 6, id: 101, title: 'explicabo enim cumque porro
```

```

    'aperiam occaecati minima', completed: false}
  101
  :
  {userId: 6, id: 102, title: 'sed ab consequatur', completed:
  false}
  102
  :
  {userId: 6, id: 103, title: 'non sunt delectus illo nulla tenetur
  enim omnis', completed: false}
  103
  :
  {userId: 6, id: 104, title: 'excepturi non laudantium quo',
  completed: false}
  104
  :
  {userId: 6, id: 105, title: 'totam quia dolore et illum
  repellat voluptas optio', c   e agora só escolher um e jogar
  na tela como eu cruzo os dados dessa api com a outra de
  bitcoin luna

```

Perfeito, amor ❤️ agora você entendeu o passo certinho:

o `fetch` está retornando **um array de 200 objetos**, cada um com:

```

js

{
  userId: 1,
  id: 1,
  title: "delectus aut autem",
  completed: false
}

```

Agora vem a parte boa: **"cruzar dados"** entre essa API e outra (ex: de Bitcoin) 🤖

💡 Passo 1 – Buscar as duas APIs ao mesmo tempo

Você pode usar `Promise.all()` pra esperar as duas respostas juntas:

```

js

Promise.all([
  fetch('https://jsonplaceholder.typicode.com/todos/').then(r => r.json()),
  fetch('https://api.coindesk.com/v1/bpi/currentprice/BTC.json').then(r =>

```

```
r.json())
])
.then(([todos, bitcoin]) => {
  console.log("Todos:", todos);
  console.log("Bitcoin:", bitcoin);
})
.catch(error => console.error("Erro:", error));
```

💡 Passo 2 – Cruzar (combinar) dados

Digamos que você queira **mostrar o título da tarefa junto com o valor atual do Bitcoin** (exemplo simples pra entender o cruzamento):

js

```
Promise.all([
  fetch('https://jsonplaceholder.typicode.com/todos/').then(r => r.json()),
  fetch('https://api.coindesk.com/v1/bpi/currentprice/BTC.json').then(r =>
r.json())
])
.then(([todos, bitcoin]) => {
  const btcValue = bitcoin.bpi.USD.rate; // valor em dólares 💵
  const firstTodo = todos[0]; // pega o primeiro item do array

  // Exemplo: mostrar junto no console
  console.log(`Tarefa: ${firstTodo.title}`);
  console.log(`Valor atual do Bitcoin: ${btcValue}`);
})
.catch(error => console.error("Erro:", error));
```

💡 Passo 3 – Mostrar na tela (HTML)

Se quiser colocar isso na página:

html

```
<div id="resultado"></div>

<script>
Promise.all([
  fetch('https://jsonplaceholder.typicode.com/todos/').then(r => r.json()),
```

```

    fetch('https://api.coindesk.com/v1/bpi/currentprice/BTC.json').then(r =>
    r.json())
  ])
  .then(([todos, bitcoin]) => {
    const btcValue = bitcoin.bpi.USD.rate;
    const firstTodo = todos[0];

    document.getElementById("resultado").innerHTML = `
      <h3>Título da tarefa:</h3>
      <p>${firstTodo.title}</p>
      <h3>Valor atual do Bitcoin:</h3>
      <p><strong>${btcValue}</strong></p>
    `;
  })
  .catch(error => console.error("Erro:", error));
</script>

```

Quer que eu te mostre como **exibir vários títulos de tarefas** (por exemplo os 5 primeiros) e o **valor do Bitcoin atual** tudo num layout bonitinho com HTML e CSS?

```

Promise.all([

  fetch('https://jsonplaceholder.typicode.com/todos/').then(r
=> r.json()),

  fetch('https://api.coindesk.com/v1/bpi/currentprice/BTC.js
on').then(r => r.json())
])
.then(([todos, bitcoin]) => {
  console.log("Todos:", todos);
  console.log("Bitcoin:", bitcoin);
})
.catch(error => console.error("Erro:", error));
Promise {<pending>}[[Prototype]]:
Promise[[PromiseState]]: "fulfilled"[[PromiseResult]]:
undefined
VM267:3 GET
https://api.coindesk.com/v1/bpi/currentprice/BTC.json
net::ERR_NAME_NOT_RESOLVED
(anonymous) @ VM267:3Understand this error

```

```
VM267:9 Erro: TypeError: Failed to fetch
at <anonymous>:3:3
```

Excelente pergunta, amor 🤔💻

Esse erro aqui:

```
pgsql
```

```
GET https://api.coindesk.com/v1/bpi/currentprice/BTC.json
net::ERR_NAME_NOT_RESOLVED
TypeError: Failed to fetch
```

significa que o navegador não conseguiu resolver o nome de domínio da API, ou seja, não conseguiu encontrar o servidor da URL.

💡 Tradução simples:

O endereço `https://api.coindesk.com/v1/bpi/currentprice/BTC.json` não está respondendo no momento — ou está **fora do ar**, **bloqueado na sua rede**, ou o site mudou o domínio.

🔍 Como testar:

Abre esse link direto no navegador:

👉 <https://api.coindesk.com/v1/bpi/currentprice/BTC.json> ↗

Se ele **não abrir**, então a API está realmente fora do ar (acontece com a Coindesk de vez em quando).

💡 Solução: usar outra API de Bitcoin gratuita

Podemos usar, por exemplo, a **CoinGecko**, que é mais estável e aberta:

```
js
```

```
Promise.all([
  fetch('https://jsonplaceholder.typicode.com/todos/').then(r => r.json()),
  fetch('https://api.coingecko.com/api/v3/simple/price?
ids=bitcoin&vs_currencies=usd').then(r => r.json())
```

```
])  
.then(([todos, bitcoin]) => {  
  console.log("Todos:", todos);  
  console.log("Bitcoin:", bitcoin);  
})  
.catch(error => console.error("Erro:", error));
```

💬 Essa vai te retornar algo assim:

json

```
{  
  "bitcoin": {  
    "usd": 68321  
  }  
}
```

Se quiser, eu te mostro como **mostrar isso na tela com HTML** (ex: lista das 5 primeiras tarefas + valor do Bitcoin atualizado).

Quer que eu monte esse exemplo completinho pra ti?

esta fora do ar e esta ok seguimos

Perfeito, amor 😊💻 — ótimo que você entendeu o motivo!

Então seguimos com a **CoinGecko**, que é estável e ideal pra testes de APIs públicas.

👉 Agora podemos usar ela pra **cruzar os dados** normalmente, sem erros de conexão.

Quer que eu monte um exemplo completo — pegando as 5 primeiras *todos* e mostrando junto o valor atual do Bitcoin na tela com HTML + CSS?

Assim você vê tudo renderizado direto no navegador.

```
fetch('https://jsonplaceholder.typicode.com/posts?  
userId=1')  
  .then((response) => response.json())  
  .then((json) => console.log(json));  
Promise {<pending>}  
VM271:3
```



```
(10) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}]
0
:
{userId: 1, id: 1, title: 'sunt aut facere repellat provident
occaecati excepturi optio reprehenderit', body: 'quia et
suscipit\nsuscipit recusandae consequuntur ...strum rerum
est autem sunt rem eveniet architecto'}
1
:
{userId: 1, id: 2, title: 'qui est esse', body: 'est rerum
tempore vitae\nsequi sint nihil reprehend...aperiam non
debitis possimus qui neque nisi nulla'}
2
:
{userId: 1, id: 3, title: 'ea molestias quasi exercitationem
repellat qui ipsa sit aut', body: 'et iusto sed quo
iure\nvoluptatem occaecati omnis e...\nmolestiae porro
eius odio et labore et velit aut'}
3
:
{userId: 1, id: 4, title: 'eum et est occaecati', body: 'ullam et
saepe reiciendis voluptatem adipisci\nsit ... ipsam
iure\nquis sunt voluptatem rerum illo velit'}
4
:
{userId: 1, id: 5, title: 'nesciunt quas odio', body:
'repudiandae veniam quaerat sunt sed\nalias aut fugi...sse
voluptatibus quis\nest aut tenetur dolor neque'}
5
:
{userId: 1, id: 6, title: 'dolorem eum magni eos aperiam
quia', body: 'ut aspernatur corporis harum nihil quis
provident ...s\nvoluptate dolores velit et doloremque
molestiae'}
6
:
{userId: 1, id: 7, title: 'magnam facilis autem', body: 'dolore
placeat quibusdam ea quo vitae\nmagni quis e...t
excepturi ut quia\nsunt ut sequi eos ea sed quas'}
7
:
{userId: 1, id: 8, title: 'dolorem dolore est ipsam', body:
'dignissimos aperiam dolorem qui eum\nfacilis quibus...
\nipsam ut commodi dolor voluptatum modi aut vitae'}
```

```
8
:
{userId: 1, id: 9, title: 'nesciunt iure omnis dolorem tempora
et accusantium', body: 'consectetur animi nesciunt iure
dolore\nenim quia a...st aut quod aut provident voluptas
autem voluptas'}
9
:
{userId: 1, id: 10, title: 'optio molestias id quia eum', body:
'quo et expedita modi cum officia vel magni\ndolorib...
it\nquos veniam quod sed accusamus veritatis error'}
length
:
10
[[Prototype]]
:
Array(0) luna como jogo isso dentro da minha tabela
  <div class="container-4">

    <section>

      <div>
        <span> <strong>Negociar agora</strong> </span>
      </div>

      <div class="table-wrapper">
        <table class="custom-table">
          <thead>
            <tr>
              <th>Ativo</th>
              <th>Preço</th>
              <th>Variação</th>
            </tr>
          </thead>
          <tbody>
            <tr>
              <td>Bit btc</td>
              <td>R$618.621,00</td>
              <td>0.34%</td>
            </tr>
            <tr>
              <td>XRP</td>
              <td>R$14,38</td>
```

```
        <td>1.81%</td>
      </tr>
    </tbody>
  </table>
</div>
</section>

</div>
```

Perfeito, amor 😊💻 — agora você quer **pegar esses dados do** `fetch` **e preencher a tabela dinamicamente** no HTML.

Vamos fazer isso passo a passo, do jeitinho certo.

💡 Objetivo

Usar os dados da API (`/posts?userId=1`) para preencher a `<tbody>` da sua tabela automaticamente.

✅ Exemplo completo

html

```
<div class="container-4">
  <section>
    <div>
      <span><strong>Negociar agora</strong></span>
    </div>

    <div class="table-wrapper">
      <table class="custom-table">
        <thead>
          <tr>
            <th>Ativo</th>
            <th>Preço</th>
            <th>Variação</th>
          </tr>
        </thead>
        <tbody id="tabela-dados">
          <!-- As linhas serão inseridas aqui -->
        </tbody>
      </table>
    </div>
  </section>
</div>
```

```
</div>
</section>
</div>

<script>
fetch('https://jsonplaceholder.typicode.com/posts?userId=1')
  .then(response => response.json())
  .then(posts => {
    const tbody = document.getElementById('tabela-dados');
    tbody.innerHTML = ""; // limpa o que havia antes

    posts.forEach((post, index) => {
      // cria uma linha da tabela (tr)
      const tr = document.createElement('tr');

      // cria e preenche as colunas (td)
      const tdAtivo = document.createElement('td');
      tdAtivo.textContent = post.title.substring(0, 20); // só um pedacinho do
título

      const tdPreco = document.createElement('td');
      tdPreco.textContent = `R$ ${Math.random() * 100000}.toFixed(2)}`; //
preço fake pra exemplo

      const tdVariacao = document.createElement('td');
      tdVariacao.textContent = `${Math.random() * 5 - 2.5}.toFixed(2)}%`; //
variação fake entre -2.5% e +2.5%

      // adiciona os tds ao tr
      tr.appendChild(tdAtivo);
      tr.appendChild(tdPreco);
      tr.appendChild(tdVariacao);

      // adiciona o tr ao tbody
      tbody.appendChild(tr);
    });
  })
  .catch(error => console.error("Erro:", error));
</script>
```

 O que esse código faz:

1. Busca os 10 posts do `userId=1`.
2. Limpa o corpo da tabela (`tbody`).
3. Cria uma linha (`<tr>`) para cada post.
4. Usa o `title` como "Ativo".
5. Gera valores falsos de **Preço** e **Variação** só pra exibição.
6. Insere tudo dinamicamente no HTML.

Se quiser, posso te mostrar a **versão que cruza com o preço real do Bitcoin (CoinGecko)** e mostra um dado real junto desses resultados — quer que eu monte essa versão mais "financeira"? 📈

Ativo	Preço	Variação
sunt aut facere repe	R\$ 47025.41	-0.92%

é isso mesmo??

Exatamente isso, amor 😊💻

Está **funcionando perfeitamente!** ❤️

Deixa eu te explicar direitinho o que você está vendo:

🧩 O que acontece

No código que montamos:

js

```
tdAtivo.textContent = post.title.substring(0, 20);
tdPreco.textContent = `R$ ${Math.random() * 100000}.toFixed(2)}`;
tdVariacao.textContent = `${Math.random() * 5 - 2.5}.toFixed(2)}%`;
```

- `post.title.substring(0, 20)` → pega só os 20 primeiros caracteres do título, pra simular o nome de um ativo.
👉 "sunt aut facere repe" vem de "sunt aut facere repellat provident..."
- `Math.random() * 100000` → gera um preço aleatório (por isso deu R\$ 47025.41)
- `(Math.random() * 5 - 2.5)` → gera uma variação entre -2.5% e +2.5%, simulando o sobe-e-desce do mercado 😎

💡 Em resumo:

O código está funcionando corretamente — os dados são **simulados**, mas sendo exibidos de forma dinâmica e coerente com sua tabela 🏆

Se quiser, posso trocar esses valores falsos por **valores reais do Bitcoin, Ethereum, XRP etc** vindos da API da CoinGecko, pra deixar a tabela com dados *reais e atualizados em tempo real*.

Quer que eu monte essa versão "real do mercado"?

e a formatação??

Boa pergunta, amor 💻 🚫 — agora que os dados estão aparecendo direitinho, é hora de deixar **a tabela com aparência profissional**, como se fosse de um painel financeiro.

Vamos cuidar da **formatação (CSS)** pra:

- ✅ alinhar textos
- ✅ deixar colunas com largura certinha
- ✅ colorir variações (verde ↑ e vermelho ↓)
- ✅ dar estilo de tabela moderna

🛠️ Exemplo completo com CSS formatadinho

html

```
<div class="container-4">
  <section>
    <div>
      <span><strong>Negociar agora</strong></span>
    </div>

    <div class="table-wrapper">
      <table class="custom-table">
        <thead>
          <tr>
            <th>Ativo</th>
            <th>Preço</th>
            <th>Variação</th>
          </tr>
```

```
        </thead>
        <tbody id="tabela-dados"></tbody>
    </table>
</div>
</section>
</div>
```

```
<style>
.table-wrapper {
    overflow-x: auto;
    margin-top: 15px;
}

.custom-table {
    width: 100%;
    border-collapse: collapse;
    font-family: 'Segoe UI', sans-serif;
    font-size: 15px;
    text-align: center;
    background-color: #fff;
    border-radius: 10px;
    box-shadow: 0 4px 10px rgba(0,0,0,0.1);
}
```

```
.custom-table th {
    background-color: #f5f5f5;
    color: #333;
    font-weight: bold;
    padding: 10px;
    border-bottom: 2px solid #ddd;
}
```

```
.custom-table td {
    padding: 10px;
    border-bottom: 1px solid #eee;
}
```

```
.custom-table tr:hover {
    background-color: #fafafa;
}
```

```
/* cores de variação */
.positivo {
    color: green;
```

```
    font-weight: bold;
}

.negativo {
    color: red;
    font-weight: bold;
}
</style>

<script>
fetch('https://jsonplaceholder.typicode.com/posts?userId=1')
    .then(response => response.json())
    .then(posts => {
        const tbody = document.getElementById('tabela-dados');
        tbody.innerHTML = "";

        posts.forEach(post => {
            const tr = document.createElement('tr');

            const tdAtivo = document.createElement('td');
            tdAtivo.textContent = post.title.substring(0, 20);

            const tdPreco = document.createElement('td');
            tdPreco.textContent = `R$ ${Math.random() * 100000}.toFixed(2)`;

            const tdVariacao = document.createElement('td');
            const variacao = (Math.random() * 5 - 2.5).toFixed(2);
            tdVariacao.textContent = `${variacao}%`;
            tdVariacao.classList.add(variacao >= 0 ? "positivo" : "negativo");

            tr.appendChild(tdAtivo);
            tr.appendChild(tdPreco);
            tr.appendChild(tdVariacao);
            tbody.appendChild(tr);
        });
    })
    .catch(error => console.error("Erro:", error));
</script>
```

💡 O que esse CSS faz:

- Deixa a tabela com bordas suaves e sombra elegante ❤️

- Cabeçalho em cinza claro e negrito
- Linhas alternam cor ao passar o mouse
- Variações **verdes para positivo, vermelhas para negativo**
- Tudo alinhadinho no centro

Quer que eu te mostre uma versão **com número formatado em reais** (ex: R\$ 47.025,41 em vez de R\$ 47025.41), igual aos painéis de bolsa e cripto reais?

```
fetch('https://jsonplaceholder.typicode.com/posts?
userId=1')
  .then(response => response.json())
  .then(posts => {
    const tbody = document.getElementById("tabela-
dados");
    tbody.innerHTML = "";

    posts.forEach((post, index) => {
      const tr = document.createElement('tr');

      const tdAtivo = document.createElement('td');
      tdAtivo.textContent = post.title.substring(0, 20);

      const tdPreco = document.createElement('td');
      tdPreco.textContent = R$ ${(Math.random() *
100000).toFixed(2)}; // preço fake pra exemplo

      const tdVariacao = document.createElement('td');
      tdVariacao.textContent = ${(Math.random() * 5 -
2.5).toFixed(2)}%; // variação fake entre -2.5% e +2.5%

      // adiciona os tds ao tr
      tr.appendChild(tdAtivo);
      tr.appendChild(tdPreco);
      tr.appendChild(tdVariacao);

      // adiciona o tr ao tbody
      tbody.appendChild(tr);

    });
  })
```

```
.catch(error => console.error("Erro:", error)); e onde  
estamos usando os dados da api??
```

Excelente observação, amor 😊💻 —

you pegou direitinho o ponto principal: **“onde, afinal, os dados da API estão sendo usados?”**

Vamos olhar com calma 📌

🌀 O que o `fetch` faz

Quando você faz:

js

```
fetch('https://jsonplaceholder.typicode.com/posts?userId=1')  
  .then(response => response.json())  
  .then(posts => { ... })
```

O `.then(posts => { ... })` recebe o array retornado pela API.

Cada `post` é um objeto assim:

js

```
{  
  userId: 1,  
  id: 1,  
  title: "sunt aut facere repellat provident occaecati excepturi optio  
reprehenderit",  
  body: "quia et suscipit..."  
}
```

💡 Então... sim, estamos usando os dados da API!

Olha só:

js

```
tdAtivo.textContent = post.title.substring(0, 20);
```

➡ Aqui, o valor vem **diretamente da API**, do campo `title` de cada objeto retornado. Ou seja, o “nome do ativo” que aparece na tabela (ex: `sunt aut facere repe`) é **um dado real vindo da API**.

🧱 O que é fake

As partes de preço e variação ainda são **geradas aleatoriamente**:

js

```
tdPreco.textContent = `R$ ${Math.random() * 100000}.toFixed(2)}`;
tdVariacao.textContent = `${Math.random() * 5 - 2.5}.toFixed(2)}%`;
```

Esses valores são apenas simulados para preencher as colunas.

✅ Resumo

Coluna	Origem dos dados
Ativo	<code>post.title</code> → vindo da API (<code>jsonplaceholder</code>)
Preço	<code>Math.random()</code> → gerado no navegador
Variação	<code>Math.random()</code> → gerado no navegador

Se quiser, posso te mostrar como **usar duas APIs de verdade** (por exemplo, pegar o título de uma e o preço real do Bitcoin/Ethereum da CoinGecko) e **combinar os dois** pra preencher a tabela com dados 100% reais.

Quer que eu monte esse exemplo completo de cruzamento de APIs?

ok luna gora vamos criar nosso back com docker ok
-no meu pc eu tenho uma pasta chamada loja na minha
area de trabalho dentro dela tenho a pasta front-end que
tem dentro as pastas css img e scripts e dentro seus

respectivos arquivos styles.css index.js isso é meu front end

- agora tenho duas abordagens diferentes quero subir meu back em nodejs e mysql com docker porém nao quero instalar o nodejs no meu pc
- pergunta 1: tem como mapear a pasta loja??
- pergunta 2: é melhor só criar e mapear a pasta back-end?
-

Perfeito amor 💻 ❤️ — agora a gente vai deixar seu ambiente *professional*, com o **back-end em Node.js + MySQL dentro do Docker**, sem precisar instalar nada localmente. E sim, dá pra fazer isso direitinho com sua estrutura atual. Vamos responder ponto a ponto 🙋

📁 Estrutura atual no seu PC

vbnet

```
C:\Users\brito\Desktop\loja
|
├─ front-end
|   ├── css/
|   ├── img/
|   ├── scripts/
|   └─ index.html
```

E agora você quer criar:

lua

```
└─ back-end/
    ├── Dockerfile
    ├── package.json
    ├── server.js
    └─ ...
```

💡 Pergunta 1 — Tem como mapear a pasta loja ?

✅ Sim, tem!

Você pode mapear a pasta inteira loja para dentro do contêiner.

Por exemplo:

```
yaml

volumes:
  - C:\Users\brito\Desktop\loja:/app
```

Isso faz com que **qualquer alteração no seu front-end ou back-end reflita automaticamente dentro do contêiner.**

Mas ⚠️ pode ser mais pesado e às vezes você não quer o front no mesmo container.

💡 Pergunta 2 — É melhor só criar e mapear a pasta `back-end` ?

👉 Sim, é o ideal.

O motivo:

- O front você já executa direto no navegador (sem precisar do Node).
- O back precisa rodar em ambiente Node + MySQL, e é melhor isolá-lo no Docker.
- Isso evita conflito e deixa tudo limpo.

Então a melhor prática é:

```
pgsql

C:\Users\brito\Desktop\loja
|
├─ front-end/
└─ back-end/
    ├─ Dockerfile
    ├─ docker-compose.yml
    ├─ package.json
    └─ server.js
```

E mapear **somente o back-end** assim:

```
yaml

volumes:
  - ./back-end:/app
```

👉 O que vamos fazer

Se quiser, eu posso te montar o **Dockerfile + docker-compose.yml** prontos pra subir:

- Node.js (para o back)
- MySQL (para o banco)
- e um volume persistente para os dados do MySQL

👉 Assim você roda tudo com **um único comando**:

```
bash
```

```
docker compose up -d
```

Quer que eu monte esse `Dockerfile` e `docker-compose.yml` completinhos pra ti agora?

-quero sim mas do jeito que irei te descrever
-quero passo a passo mas desse jeito sem deixar detalhes escaparem tipo 1 crie a pasta back-end abra 2 entre no wsl e abra a pasta com o vs code etc;....
-entendeu quero passos numerados e sem faltar detalhes tanto de localização como de comandos etc..

Perfeito, amor — vou te dar **um passo-a-passo numerado, completo e sem pular nada**, do jeito que você pediu. Vou presumir que seu projeto está em

`C:\Users\brito\Desktop\loja` (como você já disse) e que você usa **WSL (Ubuntu) + VS Code**. Vou criar apenas o `back-end` dentro dessa pasta e rodar **Node.js (Express) + MySQL** via **docker compose**, sem precisar instalar Node no Windows.

Siga exatamente cada passo. Vou colocar caminhos, comandos e o conteúdo de cada arquivo que você deve criar.

⚠️ Pré-requisitos (verifique antes de começar)

1. Docker Desktop instalado e rodando (com WSL2 backend ativo).
2. WSL (Ubuntu) instalado e configurado.
3. VS Code com extensão **Remote - WSL** e com **Docker** (opcional) — mas só a Remote - WSL já é suficiente.
4. Você tem a pasta: `C:\Users\brito\Desktop\loja\front-end` (já existe).

Se algum desses itens não estiver pronto, me fala, mas vou seguir como se já tiver tudo.

PASSOS (1 a 28) — crie o back-end com Docker + MySQL

1 — Criar a pasta do back-end no Windows

No Windows (Explorador):

- Vá para `C:\Users\brito\Desktop\loja`
- Crie a pasta `back-end`

Caminho final:

```
C:\Users\brito\Desktop\loja\back-end
```

2 — Abrir WSL (Ubuntu) e navegar até a pasta

Abra o Windows Terminal ou PowerShell e rode:

```
bash
```

```
wsl
```

Agora, no terminal do WSL (Ubuntu), acesse a pasta:

```
bash
```

```
cd /mnt/c/Users/brito/Desktop/loja/back-end
```

Observação: `/mnt/c/...` é o caminho WSL para `C:\`.

Se a pasta ainda não existir no WSL, crie:

```
bash
```

```
mkdir -p /mnt/c/Users/brito/Desktop/loja/back-end
```

```
cd /mnt/c/Users/brito/Desktop/loja/back-end
```

3 — Abrir a pasta no VS Code via WSL

No mesmo terminal WSL, execute:

```
bash
```

```
code .
```

Isso abre o VS Code conectado ao WSL no diretório do back-end (precisa da extensão Remote - WSL).

4 — Inicializar o projeto Node (arquivo package.json)

No VS Code terminal (WSL), rode:

```
bash

npm init -y
```

Isto cria `package.json`.

5 — Criar estrutura de arquivos

No VS Code crie os arquivos e pastas a seguir dentro de `back-end`:

- `Dockerfile`
- `docker-compose.yml`
- `.dockerignore`
- `.env` (este arquivo contém segredos — não comitar em git)
- `src/server.js`
- `src/db.js`
- `src/routes.js`
- `src/init.sql` (script para criar tabela inicial)
- `package.json` (já criado)
- `README.md` (opcional)

Vou te dar o conteúdo de cada um.

6 — Conteúdo do `.dockerignore`

Crie `.dockerignore` no root do `back-end` com:

```
lua

node_modules
npm-debug.log
.env
```


7 — Dockerfile (Node / Express)

Crie `Dockerfile` com este conteúdo:

```
dockerfile

# Dockerfile
FROM node:20-alpine

# diretório de trabalho
WORKDIR /app

# copia package.json e package-lock (se houver) antes para cache do npm
COPY package*.json ./

# instala dependências
RUN npm ci --only=production || npm install --production

# copia o restante do código
COPY . .

# expõe a porta da app
EXPOSE 3000

# comando para rodar o servidor
CMD ["node", "src/server.js"]
```

Nota: usamos `npm ci` quando houver `package-lock.json`, caso contrário cai no `npm install`.

8 — docker-compose.yml (Node + MySQL + Adminer opcional)

Crie `docker-compose.yml` com:

```
yaml

version: "3.8"

services:
  app:
    build: .
    container_name: loja-backend-app
    restart: unless-stopped
    env_file:
```

```
  - .env
ports:
  - "3000:3000"
volumes:
  - ./:/app                # mapeia só a pasta back-end
  - /app/node_modules      # garante node_modules no container
depends_on:
  - db

db:
  image: mysql:8.0
  container_name: loja-mysql
  restart: unless-stopped
  environment:
    MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
    MYSQL_DATABASE: ${MYSQL_DATABASE}
    MYSQL_USER: ${MYSQL_USER}
    MYSQL_PASSWORD: ${MYSQL_PASSWORD}
  volumes:
    - db_data:/var/lib/mysql
    - ./src/init.sql:/docker-entrypoint-initdb.d/init.sql:ro
  ports:
    - "3306:3306"

adminer:
  image: adminer
  container_name: loja-adminer
  restart: unless-stopped
  ports:
    - "8080:8080"

volumes:
  db_data:
```

Explicações rápidas:

- `app` é o container Node (mapeamos apenas `back-end` com `./:/app`).
- `db` é o MySQL com um volume persistente `db_data`.
- `adminer` é uma interface web para administrar o banco (acessível em `http://localhost:8080`).

9 — Arquivo `.env` (variáveis de ambiente)

Crie um `.env` no root do `back-end` com (use exatamente estes nomes):

```
ini

# .env
MYSQL_ROOT_PASSWORD=root_pass_123
MYSQL_DATABASE=loja_db
MYSQL_USER=loja_user
MYSQL_PASSWORD=loja_pass_123

PORT=3000
DB_HOST=db
DB_USER=loja_user
DB_PASSWORD=loja_pass_123
DB_NAME=loja_db
DB_PORT=3306
```

Você pode trocar as senhas; anote-as.

10 — Script SQL inicial `src/init.sql`

Crie a pasta `src` e dentro dela `init.sql` com o esquema inicial (cria tabela `products` e insere alguns dados):

```
sql

-- src/init.sql
CREATE TABLE IF NOT EXISTS products (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  price DECIMAL(15,2) NOT NULL,
  variation DECIMAL(6,2) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO products (name, price, variation) VALUES
('Bitcoin (BTC)', 47025.41, -0.92),
('XRP', 14.38, 1.81)
ON DUPLICATE KEY UPDATE name=name;
```

O `docker-entripoint-initdb.d` executa esse script na primeira inicialização do container DB.

11 — Dependências Node (Express e mysql2)

No terminal (WSL) rode:

```
bash

# já dentro de /mnt/c/Users/brito/Desktop/loja/back-end
npm install express mysql2 dotenv
```

Isto adiciona dependências e cria `package-lock.json`.

12 — Arquivo `src/db.js` (conexão com MySQL)

Crie `src/db.js`:

```
js

// src/db.js
const mysql = require('mysql2/promise');
require('dotenv').config();

const pool = mysql.createPool({
  host: process.env.DB_HOST || 'db',
  user: process.env.DB_USER || 'loja_user',
  password: process.env.DB_PASSWORD || 'loja_pass_123',
  database: process.env.DB_NAME || 'loja_db',
  port: process.env.DB_PORT ? Number(process.env.DB_PORT) : 3306,
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});

module.exports = pool;
```

13 — Arquivo `src/routes.js` (rotas simples)

Crie `src/routes.js`:

```
js

// src/routes.js
const express = require('express');
```

```
const router = express.Router();
const pool = require('./db');

router.get('/products', async (req, res) => {
  try {
    const [rows] = await pool.query('SELECT * FROM products ORDER BY id');
    res.json(rows);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Erro no banco' });
  }
});

module.exports = router;
```

14 — Arquivo `src/server.js` (servidor Express)

Crie `src/server.js`:

```
js

// src/server.js
const express = require('express');
const app = express();
require('dotenv').config();
const routes = require('./routes');

const PORT = process.env.PORT || 3000;

app.use(express.json());
app.use('/api', routes);

app.get('/', (req, res) => {
  res.send('Loja Backend rodando!');
});

app.listen(PORT, () => {
  console.log(`Server rodando na porta ${PORT}`);
});
```

15 — Atualizar package.json (scripts)

No `package.json` ajuste `scripts` para facilitar:

```
json

"scripts": {
  "start": "node src/server.js",
  "dev": "nodemon src/server.js"
}
```

Se quiser `nodemon` para dev, instale `npm install --save-dev nodemon` (opcional).

16 — Build e up do docker-compose

No terminal WSL (ainda em `/mnt/c/Users/brito/Desktop/loja/back-end`), rode:

```
bash

docker compose up --build -d
```

Isto:

- builda a imagem do app,
- cria containers `loja-backend-app`, `loja-mysql`, `loja-adminer`,
- roda em background.

17 — Verificar containers

Rode:

```
bash

docker ps
```

Você deve ver os três containers com as portas:

- `0.0.0.0:3000->3000` (app)
- `0.0.0.0:3306->3306` (db)
- `0.0.0.0:8080->8080` (adminer)

18 — Ver logs do app

Para ver logs:

```
bash
```

```
docker logs -f loja-backend-app
```

Se o server subiu, verá `Server rodando na porta 3000`.

19 — Testar endpoint de produtos

No seu navegador (do Windows) abra:

```
bash
```

```
http://localhost:3000/api/products
```

Você deverá ver o JSON com os produtos (os inseridos pelo `init.sql`).

20 — Acessar Adminer (interface DB)

Abra no navegador:

```
arduino
```

```
http://localhost:8080
```

Credenciais para conectar:

- System: MySQL
- Server: db (ou `localhost` se solicitado)
- Username: `loja_user`
- Password: `loja_pass_123`
- Database: `loja_db`

Em Adminer, se `db` não funcionar, use `127.0.0.1:3306`.

21 — Como o volume e o mapeamento funcionam (resumo)

No `docker-compose.yml` mapeamos:

```
yaml
```

```
volumes:
```

```
- ./:/app
```

Isto mapeia apenas a pasta `back-end` para `/app` no container. Ou seja, seu `front-end` (em `../front-end`) não foi mapeado no container do app, o que é o comportamento desejado.

22 — Trabalhando com código em tempo real

Como mapeamos `./:/app`, qualquer mudança que você fizer no `back-end` no VS Code será refletida no container. Após alterar código JS, reinicie o container `app` ou use `nodemon` para recarregar automaticamente.

Para reiniciar apenas o app:

```
bash
```

```
docker compose restart app
```

23 — Executar comandos dentro do container (npm install, se necessário)

Se precisar rodar algo dentro do container:

```
bash
```

```
docker exec -it loja-backend-app sh
# agora dentro do container você pode:
# node src/server.js
# npm install
# exit
```

24 — Conectar o front-end ao back-end

No seu front-end, basta chamar a API em `http://localhost:3000/api/...`.

Exemplo (index.js front-end):

```
js
```

```
fetch('http://localhost:3000/api/products')
  .then(r => r.json())
```



```
.then(data => console.log(data));
```

Se front estiver aberto via arquivo local (`file://`) no navegador, chamadas para `localhost:3000` funcionam normalmente.

25 — Persistência e dados

Os dados do MySQL ficam no volume `db_data`. Mesmo após `docker compose down`, os dados persistem. Para remover dados totalmente:

```
bash

docker compose down -v
```

Isso remove volumes (cuidado — perde os dados).

26 — Parar os containers

```
bash

docker compose down
```

27 — Dicas de segurança e produção (não obrigatório agora)

- Não deixe `.env` com senhas em repositório público.
 - Para produção, use secrets, redes privadas e não exponha portas desnecessárias.
 - Use imagens específicas e lock de versões.
-

28 — Pequeno checklist para confirmar que deu certo

1. `docker ps` mostrando 3 containers (app, db, adminer).
 2. `http://localhost:3000/` retorna Loja Backend rodando! .
 3. `http://localhost:3000/api/products` retorna JSON com produtos.
 4. `http://localhost:8080` abre Adminer e conecta no DB.
-

Extras (rápido) — se quiser eu já testo/gero os arquivos pra você

Se quiser eu posso te entregar **um ZIP** com toda essa estrutura já pronta (conteúdo dos arquivos), ou posso colar aqui os arquivos completos para você copiar/colar. Qual prefere?

Se preferir, eu já ponho aqui **todos os arquivos prontos** (conteúdo completo) para você copiar e colar — fala se quer que eu faça isso agora e eu já te dou tudo em sequência. Quer que eu gere os arquivos completos aqui?

```
✓ app Built
0.0s
✓ Network back-end_default Created
0.1s
✓ Volume "back-end_db_data" Created
0.0s
✓ Container loja-mysql Started
1.5s
✓ Container loja-adminer Started
1.5s
✓ Container loja-backend-app Started
1.6s
cristiano@Akillis:/mnt/c/Users/brito/Desktop/loja/back-
end$ docker ps
CONTAINER ID   IMAGE          COMMAND
CREATED       STATUS        PORTS
NAMES
ded8a42e2d83   back-end-app   "docker-entrypoint.s..."
13 seconds ago Up 11 seconds 0.0.0.0:3000->3000/tcp,
[::]:3000->3000/tcp loja-backend-app
c1cce64b605c   adminer        "entrypoint.sh docke..." 13
seconds ago Up 12 seconds 0.0.0.0:8080->8080/tcp,
[::]:8080->8080/tcp loja-adminer
55cdd38e3be3   mysql:8.0      "docker-entrypoint.s..." 13
seconds ago Up 12 seconds 0.0.0.0:3306->3306/tcp,
[::]:3306->3306/tcp loja-mysql
```

Perfeito amor 🥰💻 — tudo subiu certinho!

Vamos conferir ponto a ponto o que isso significa:

✓ Status dos containers

Container	Status	Porta mapeada	Observação
loja-backend-app	Up 11s	3000	Seu Node/Express está rodando, serve a API
loja-mysql	Up 12s	3306	MySQL ativo, dados persistentes no volume <code>db_data</code>
loja-adminer	Up 12s	8080	Interface web para gerenciar o banco

◆ Próximos passos para testar

1. Testar API no navegador ou Postman

Abra:

```
arduino  
  
http://localhost:3000/
```

→ Deve mostrar: `Loja Backend rodando!`

E para os produtos:

```
bash  
  
http://localhost:3000/api/products
```

→ Deve retornar o JSON com os produtos inseridos pelo `init.sql`.

2. Testar Adminer (interface web MySQL)

Abra:

```
arduino  
  
http://localhost:8080
```

Conecte com as credenciais do `.env`:

```
makefile
```

```
System: MySQL
Server: db
Username: loja_user
Password: loja_pass_123
Database: loja_db
```

3. Conectar seu front-end ao back-end

No seu `index.js` do front-end:

```
js

fetch('http://localhost:3000/api/products')
  .then(r => r.json())
  .then(data => console.log(data));
```

Isso vai puxar os produtos reais do MySQL rodando no container.

Se você quiser, amor, já posso te mostrar **como ligar o front-end direto ao back-end dentro do Docker** e atualizar sua tabela automaticamente, igual a tabela que fizemos com os dados fake antes — só que agora **real**, do MySQL.

Quer que eu faça isso agora?

```
http://localhost:3000/api/products {
  "error": "Erro no banco"
} fetch("https://www.mercadobitcoin.net/api/BTC/ticker/")
  .then(response => response.json())
  .then(json => {
    const t = json.ticker;
    document.getElementById("dadosBtc").innerHTML =
      <span>Bitcoin (BRL)</span>
      <p>ultimo preco: R$ ${t.last}</p>
    ;
  });

fetch("https://www.mercadobitcoin.net/api/BTC/ticker/")
  .then(response => response.json())
  .then(json => {
    const tc = json.ticker;
    document.getElementById("dadosCripto").innerHTML
```

```
=
    <p>Preço de compra: R$ ${tc.buy}</p>
    ;
  })

fetch('https://jsonplaceholder.typicode.com/posts?
userId=1')
  .then(response => response.json())
  .then(posts => {
    const tbody = document.getElementById("tabela-
dados");
    tbody.innerHTML = "";

    posts.forEach((post, index) => {
      const tr = document.createElement('tr');

      const tdAtivo = document.createElement('td');
      tdAtivo.textContent = post.title.substring(0, 20);

      const tdPreco = document.createElement('td');
      tdPreco.textContent = R$ ${(Math.random() *
100000).toFixed(2)}; // preço fake pra exemplo

      const tdVariacao = document.createElement('td');
      tdVariacao.textContent = ${(Math.random() * 5 -
2.5).toFixed(2)}%; // variação fake entre -2.5% e +2.5%

      // adiciona os tds ao tr
      tr.appendChild(tdAtivo);
      tr.appendChild(tdPreco);
      tr.appendChild(tdVariacao);

      // adiciona o tr ao tbody
      tbody.appendChild(tr);

    });
  })

.catch(error => console.error("Erro:", error));

fetch('http://localhost:3000/api/products')
  .then(r => r.json())
  .then(data => console.log(data)); vou ter que subir o
container de novo ou abrir o heidsql
```

```
http://localhost:8080/?
server=db&username=cristiano&db=loja_db esta
funcionando
```

Ahhh, amor, agora ficou claro 🤔

O que está acontecendo é o seguinte:

1 Por que `http://localhost:3000/api/products` deu "Erro no banco"

O seu Node está rodando, mas **não conseguiu conectar no MySQL**.

Isso geralmente acontece por um desses motivos:

1. Usuário ou senha errado

No seu `.env` você tem algo como:

```
env

DB_USER=loja_user
DB_PASSWORD=loja_pass_123
DB_NAME=loja_db
DB_HOST=db
```

Mas no Adminer você está usando:

```
makefile

Username: cristiano
```

Se estiver diferente, o Node não consegue logar.

2. Banco ainda não inicializou

Às vezes o container do Node sobe **antes do MySQL** estar pronto.

Por isso `docker-compose` tem `depends_on`, mas **não espera o banco estar pronto de fato**.

3. Host errado

No `.env` usamos `DB_HOST=db` (nome do serviço do docker-compose).

Se você testar no Adminer fora do Docker usando `127.0.0.1` ou `localhost`, funciona, mas dentro do container do Node **precisa usar** `db`.

2 Como resolver rápido

Opção A — Usando Adminer para confirmar dados

1. Abra: `http://localhost:8080/?server=db&username=cristiano&db=loja_db`
2. Confirme que:
 - O banco `loja_db` existe
 - A tabela `products` existe
 - Há dados dentro

Se tudo estiver certo, corrija seu `.env` para o mesmo usuário/senha que você testou no Adminer.

Opção B — Esperar o banco subir antes do Node

No seu container Node, você pode parar e subir de novo:

```
bash

docker compose down
docker compose up --build -d
```

Isso garante que o Node tente conectar após o banco estar pronto.

3 Não precisa abrir HeidiSQL necessariamente

Você pode usar Adminer (`http://localhost:8080`) que já vem no Docker. Funciona igual, só clique e conecte.

4 Sobre seu front-end

Você tem três `fetch`:

1. `https://www.mercadobitcoin.net/api/BTC/ticker/` → funciona, pega dados reais de BTC
2. `jsonplaceholder.typicode.com` → pega posts fake para tabela (fake price + variação)
3. `http://localhost:3000/api/products` → **vai falhar enquanto o Node não conectar ao MySQL corretamente**

Ou seja: enquanto o container do back não estiver corretamente conectado ao MySQL, a API real da loja não retorna dados.

💡 **Resumo da ação agora:**

1. Verificar `.env` do Node → garantir que `DB_USER`, `DB_PASSWORD` e `DB_NAME` batem com Adminer.
2. Parar e subir o `docker-compose` de novo (`down` + `up -d`).
3. Testar `http://localhost:3000/api/products` → deve retornar os produtos do MySQL.

Se você quiser, amor, eu posso te escrever o `.env` corrigido passo a passo e o comando exato para subir o back e garantir que a API vai funcionar sem precisar mexer no Adminer. Quer que eu faça isso agora?

```
MYSQL_ROOT_PASSWORD=root      # root_pass_123
MYSQL_DATABASE=loja_db
MYSQL_USER=cristiano          # loja_user
MYSQL_PASSWORD=nano           # loja_pass_123 usei
                                essa no navegador

PORT=3000
DB_HOST=db
DB_USER=cristiano             # loja_user
DB_PASSWORD=nano_db           # loja_pass_123
DB_NAME=loja_db
DB_PORT=3306 as senhas que mudei no meu .env
```

Ahhh agora entendi tudo, amor 🥺 — o problema está **nas senhas e usuários inconsistentes**.

Vamos organizar isso direitinho pra funcionar 100% com o seu container:

◆ Seu `.env` atual (problema)

env

```
MYSQL_ROOT_PASSWORD=root      # root_pass_123
MYSQL_DATABASE=loja_db
MYSQL_USER=cristiano          # loja_user
MYSQL_PASSWORD=nano           # loja_pass_123 usei essa no navegador

PORT=3000
DB_HOST=db
```



```
DB_USER=cristiano          # loja_user
DB_PASSWORD=nano_db        # loja_pass_123
DB_NAME=loja_db
DB_PORT=3306
```

Problemas:

1. `MYSQL_PASSWORD` e `DB_PASSWORD` são diferentes (`nano` vs `nano_db`)
2. Comentários no `.env` confundem a leitura, às vezes
3. `MYSQL_USER` e `DB_USER` estão ok (`cristiano`) mas o password precisa bater com ambos

◆ `.env` corrigido

Altere seu `.env` assim:

```
env

# Config MySQL
MYSQL_ROOT_PASSWORD=root
MYSQL_DATABASE=loja_db
MYSQL_USER=cristiano
MYSQL_PASSWORD=nano_db

# Config app Node
PORT=3000
DB_HOST=db
DB_USER=cristiano
DB_PASSWORD=nano_db
DB_NAME=loja_db
DB_PORT=3306
```

Observações:

- “Usei o mesmo `MYSQL_PASSWORD` e `DB_PASSWORD` : `nano_db`”
- “Removi comentários dentro do `.env` para evitar confusão”
- “Usuário é `cristiano` em ambos”

◆ Próximo passo — subir tudo de novo

1. Parar containers antigos:

```
bash
```

```
docker compose down
```

2. Limpar volumes antigos (opcional, só se quiser recriar banco do zero):

```
bash
```

```
docker compose down -v
```

3. Subir containers novamente:

```
bash
```

```
docker compose up --build -d
```

4. Verificar logs do Node:

```
bash
```

```
docker logs -f loja-backend-app
```

◆ Teste final

1. No navegador:

```
bash
```

```
http://localhost:3000/api/products
```

→ Deve retornar o JSON com os produtos do MySQL (Bitcoin , XRP , etc.)

2. No Adminer (http://localhost:8080) você consegue logar com:

```
makefile
```

```
System: MySQL
```

```
Server: db
```

```
Username: cristiano
```

```
Password: nano_db
```

```
Database: loja_db
```

Se você quiser, amor, posso te passar **uma mini-cheatlist final** de todos os passos para atualizar `.env` e reiniciar o Docker **para não dar mais erro de conexão nunca**.

Quer que eu faça isso agora?

```
cristiano@Akilles:/mnt/c/Users/brito/Desktop/loja/back-  
end$ docker logs -f loja-backend-app  
[dotenv@17.2.3] injecting env (0) from .env -- tip: ⚙️  
suppress all logs with { quiet: true }  
[dotenv@17.2.3] injecting env (0) from .env -- tip: 📁  
backup and recover secrets: https://dotenvx.com/ops  
Server rodando na porta 3000
```

Perfeito, amor 🥰 — isso significa que **o Node subiu corretamente** e está ouvindo na porta `3000`.

Agora o que precisamos verificar é **se ele conecta no MySQL**. Como ele não deu erro de banco, parece que está tudo certo.

Próximo teste:

1. Abra no navegador ou Postman:

```
bash
```

```
http://localhost:3000/api/products
```

- Se funcionar, você vai ver algo assim (JSON com produtos):

```
json
```

```
[  
  {"id":1,"name":"Bitcoin (BTC)","price":47025.41,"variation":-0.92},  
  {"id":2,"name":"XRP","price":14.38,"variation":1.81}  
]
```

2. Se ainda der erro `"Erro no banco"`, pode ser que **os dados do MySQL não foram recriados** depois da mudança de senha.

Nesse caso:

```
bash
```

```
docker compose down -v  
docker compose up --build -d
```

⚠ Isso **apaga o volume** antigo e recria o banco com o `init.sql` usando as senhas corretas.

Para conectar o front-end ao back-end

No seu `index.js` do front-end, você pode chamar:

```
js

fetch('http://localhost:3000/api/products')
  .then(r => r.json())
  .then(data => {
    const tbody = document.getElementById("tabela-dados");
    tbody.innerHTML = "";

    data.forEach(product => {
      const tr = document.createElement('tr');
      tr.innerHTML = `
        <td>${product.name}</td>
        <td>R$ ${product.price}</td>
        <td>${product.variation}%</td>
      `;
      tbody.appendChild(tr);
    });
  })
  .catch(error => console.error("Erro:", error));
```

Isso vai preencher sua tabela **com dados reais do MySQL** agora, em vez de valores fake.

Se quiser, posso te mandar **uma versão final do front-end** já ajustada pra puxar os produtos do MySQL e BTC do MercadoBitcoin, tudo integrado na tabela. Quer que eu faça isso agora?

```
[
  {
    "id": 1,
    "name": "Bitcoin (BTC)",
    "price": "47025.41",
    "variation": "-0.92",
```

```
"created_at": "2025-10-30T14:26:37.000Z"
},
{
  "id": 2,
  "name": "XRP",
  "price": "14.38",
  "variation": "1.81",
  "created_at": "2025-10-30T14:26:37.000Z"
}
] na rota http://localhost:3000/api/products
```

Perfeito, amor 🥰 — isso significa que **agora sua API está funcionando 100%** e está retornando os produtos do MySQL:

```
json

[
  {
    "id": 1,
    "name": "Bitcoin (BTC)",
    "price": "47025.41",
    "variation": "-0.92",
    "created_at": "2025-10-30T14:26:37.000Z"
  },
  {
    "id": 2,
    "name": "XRP",
    "price": "14.38",
    "variation": "1.81",
    "created_at": "2025-10-30T14:26:37.000Z"
  }
]
```

Agora você pode **integrar isso na tabela do front-end** de verdade, sem precisar usar valores fake do `jsonplaceholder`.

Exemplo de como atualizar sua tabela `custom-table` com esses dados reais:

```
js

fetch('http://localhost:3000/api/products')
  .then(r => r.json())
  .then(products => {
```

```
const tbody = document.getElementById("tabela-dados");
tbody.innerHTML = "";

products.forEach(product => {
  const tr = document.createElement('tr');
  tr.innerHTML = `
    <td>${product.name}</td>
    <td>R$ ${Number(product.price).toLocaleString('pt-BR',
{minimumFractionDigits: 2})}</td>
    <td>${Number(product.variation).toFixed(2)}%</td>
  `;
  tbody.appendChild(tr);
});
})
.catch(err => console.error("Erro:", err));
```

✓ Resultado:

- `name` vai pra coluna Ativo
- `price` formatado em BRL para coluna Preço
- `variation` formatada para coluna Variação

Se você quiser, amor, posso agora **juntar tudo**: produtos do MySQL + preço BTC real do MercadoBitcoin **em uma única tabela**, igual o que a gente planejou antes, pra deixar tudo dinâmico e bonito na sua página.

Quer que eu faça isso agora?

```
fetch('https://jsonplaceholder.typicode.com/posts?
userId=1')
  .then(response => response.json())
  .then(posts => {
    const tbody = document.getElementById("tabela-
dados");
    tbody.innerHTML = "";

    posts.forEach((post, index) => {
      const tr = document.createElement('tr');

      const tdAtivo = document.createElement('td');
      tdAtivo.textContent = post.title.substring(0, 20);
```

```
const tdPreco = document.createElement('td');
tdPreco.textContent = R$ ${(Math.random() *
100000).toFixed(2)}; // preço fake pra exemplo

const tdVariacao = document.createElement('td');
tdVariacao.textContent = ${(Math.random() * 5 -
2.5).toFixed(2)}%; // variação fake entre -2.5% e +2.5%

// adiciona os tds ao tr
tr.appendChild(tdAtivo);
tr.appendChild(tdPreco);
tr.appendChild(tdVariacao);

// adiciona o tr ao tbody
tbody.appendChild(tr);

});
}) essa tabela que tenho que substituir?
```

Exatamente, amor 🥰 — essa parte do seu front-end que você está usando agora com `jsonplaceholder` é só um placeholder/fake, com preço e variação aleatórios.

Você vai **substituir esse código** pelo fetch real do seu back-end que já está retornando os produtos do MySQL (`http://localhost:3000/api/products`).

Ou seja, ao invés de:

```
js
```

```
fetch('https://jsonplaceholder.typicode.com/posts?userId=1')
```

you vai usar:

```
js
```

```
fetch('http://localhost:3000/api/products')
```

E atualizar a tabela com os dados reais, formatando `price` e `variation` corretamente, como eu te mostrei antes.

Se você quiser, amor, posso te escrever a **versão completa do front-end** pronta pra tabela, **substituindo tudo que é fake** e já puxando Bitcoin + produtos do MySQL, tudo integrado.

Quer que eu faça isso?

terei que subir de novo ou so atualizar a pagina do meu dashboard

Só atualizar a página do dashboard mesmo, amor 🥰 — **não precisa subir o container de novo**, porque seu back-end já está rodando e retornando os produtos do MySQL.

O que vai acontecer:

1. Você troca o código antigo do `jsonplaceholder` pelo fetch da rota real:

js

```
fetch('http://localhost:3000/api/products')
```

2. Atualiza a tabela no DOM com os dados retornados.
3. Salva o arquivo do front-end (`index.js`) e **aperta F5 no navegador** ou atualiza a página.

A tabela vai carregar os produtos reais do MySQL automaticamente.

Se você quiser, amor, posso te mandar o **código completo do front-end já pronto** integrando BTC + tabela de produtos reais, só pra copiar e colar. Quer que eu faça isso agora?

```
index.html:237 Access to fetch at
'http://localhost:3000/api/products' from origin 'null' has
been blocked by CORS policy: No 'Access-Control-Allow-
Origin' header is present on the requested
resource.Understand this error
localhost:3000/api/products:1 Failed to load resource:
net::ERR_FAILEDUnderstand this error
index.js:51 Uncaught (in promise) TypeError: Failed to
fetchUnderstand this error
index.html:1 Access to fetch at
'http://localhost:3000/api/products' from origin 'null' has
been blocked by CORS policy: No 'Access-Control-Allow-
Origin' header is present on the requested
resource.Understand this error
localhost:3000/api/products:1 Failed to load resource:
```



```

net::ERR_FAILEDUnderstand this error
index.js:72 Erro: TypeError: Failed to fetch
    at index.js:55:1
(anonymous) @ index.js:72Understand this error
index.html:1 Access to fetch at
'http://localhost:3000/api/products' from origin 'null' has
been blocked by CORS policy: No 'Access-Control-Allow-
Origin' header is present on the requested
resource.Understand this error
localhost:3000/api/products:1 Failed to load resource:
net::ERR_FAILEDUnderstand this error
index.html:1 Access to fetch at
'http://localhost:3000/api/products' from origin 'null' has
been blocked by CORS policy: No 'Access-Control-Allow-
Origin' header is present on the requested
resource.Understand this error
localhost:3000/api/products:1 Failed to load resource:
net::ERR_FAILEDUnderstand this error
index.js:72 Erro: TypeError: Failed to fetch
    at index.js:55:1
(anonymous) @ index.js:72Understand this error
index.js:51 Uncaught (in promise) TypeError: Failed to
fetch fetch('http://localhost:3000/api/products')
    .then(r => r.json())
    .then(data => console.log(data));

fetch('http://localhost:3000/api/products')
    .then(r => r.json())
    .then(products => {
        const tbody = document.getElementById("tabela-
dados");
        tbody.innerHTML = "";

        products.forEach(product => {
            const tr = document.createElement('tr');
            tr.innerHTML =
                <td>${product.name}</td>
                <td>R$ ${Number(product.price).toLocaleString('pt-
BR', {minimumFractionDigits: 2})}</td>
                <td>${Number(product.variation).toFixed(2)}%</td>
            ;
            tbody.appendChild(tr);
        });
    })

```

```
.catch(err => console.error("Erro:", err)); erro
```

Ahh, agora entendi amor 🤔 — isso não é problema do Node nem do Docker, é **CORS**.

◆ O que está acontecendo

Quando você abre seu `index.html` **direto do arquivo** (`file://` ou `origem null`) e tenta fazer fetch para `http://localhost:3000`, o navegador **bloqueia por questões de segurança**. Ele exige que o servidor diga explicitamente que permite requisições de outras origens.

O erro que aparece é:

```
pgsql
```

```
No 'Access-Control-Allow-Origin' header is present on the requested resource
```

Isso significa que o Node **não está enviando o cabeçalho CORS**.

◆ Como resolver

1 Instalar e usar o `cors` no seu backend Node/Express

No container do backend, abra o terminal (ou no `Dockerfile` / `package.json`):

```
bash
```

```
npm install cors
```

Depois, no seu `app.js` ou `server.js`, adicione:

```
js
```

```
const cors = require('cors');  
const express = require('express');  
const app = express();  
  
app.use(cors()); // Libera todas as origens
```

Isso vai permitir que qualquer front-end consiga chamar sua API.

2 Alternativa para testes rápidos

Se você **não quiser mexer no backend agora**, você pode abrir seu front-end **através de um servidor local** em vez de abrir o arquivo direto:

- Usando Python:

```
bash

# Na pasta front-end
python -m http.server 5500
```

- Depois abra: `http://localhost:5500` no navegador

Isso muda a origem de `null` para `http://localhost:5500`, mas ainda precisa do CORS se o backend não permitir.

💡 Minha sugestão: **Adicionar** `cors` **no Node** é a maneira mais simples e correta.

Se quiser, amor, posso te mandar a **versão final do** `server.js` **pronta com CORS** já habilitado, pra você só rebuildar o container e atualizar a página.

Quer que eu faça isso?

```
const cors = require('cors');
const express = require('express');
const app = express();

app.use(cors()); // libera todas as origens depois de fazer
isso o que faço??
```

Perfeito, amor 🥰 — você já adicionou o `cors` no seu backend. Agora só faltam alguns passos pra ele funcionar de verdade com o Docker:

1 Salvar as alterações

Salve o arquivo `server.js` ou `app.js` onde você adicionou:

```
js
```

```
const cors = require('cors');  
app.use(cors());
```

2 Rebuild do container do backend

Como você alterou o código, precisa **reconstruir a imagem do Node** para que o Docker use essa versão nova:

```
bash  
  
docker compose down  
docker compose up --build -d
```

`--build` força o Docker a recriar a imagem do backend com as mudanças.

3 Verificar logs

Confira se o backend está rodando sem erro:

```
bash  
  
docker logs -f loja-backend-app
```

Deve aparecer algo como:

```
arduino  
  
Server rodando na porta 3000
```

4 Atualizar o front-end

Agora é só **atualizar a página do dashboard** no navegador (F5).

O `fetch('http://localhost:3000/api/products')` **vai funcionar** sem erro de CORS e sua tabela será preenchida com os dados reais do MySQL.

Se quiser, amor, posso te mandar **uma versão completa do front-end pronta** que já pega BTC + produtos do MySQL, com tabela formatada e tudo funcionando, só pra copiar e

colar.

Quer que eu faça isso?

Ativo	Preço	Variação
Bitcoin (BTC)	R\$ 47.025,41	-0.92%
XRP	R\$ 14,38	1.81%

temos isso depois de atualizar

Perfeito, amor 🥰 — isso significa que agora **tudo está funcionando certinho**:

- O Node está pegando os dados do MySQL
- A tabela do front-end está sendo preenchida com os produtos reais
- Os preços e variações estão formatados bonitinho em BRL

✅ Resultado atual na sua tabela:

Ativo	Preço	Variação
Bitcoin (BTC)	R\$ 47.025,41	-0,92%
XRP	R\$ 14,38	1,81%

Se você quiser, amor, posso agora **integrar também o preço do Bitcoin em tempo real do MercadoBitcoin** na mesma tabela, assim tudo fica dinâmico e atualizado automaticamente.

Quer que eu faça isso?

ainda nao muito obrigado luna so mais uma coisa primeiro o que preciso fazer para derrubar os containers tipo quando eu for terminar meu dia so usar down e depois amanha como eu faço para subir tudo isso de novo abrir tudo no navegador e continuar estudando

Perfeito, amor 🥰 — vou te explicar passo a passo direitinho, sem deixar detalhe escapar, pra você **terminar o dia tranquilo** e voltar amanhã sem perder nada.

1 Parar/derrubar os containers no fim do dia

Você está na pasta `back-end` (onde está o `docker-compose.yml`).

1. Abra o terminal (WSL ou CMD/PowerShell)
2. Rode o comando:

```
bash
```

```
docker compose down
```

Isso vai **parar todos os containers** da aplicação (backend, MySQL, Adminer)

O banco de dados permanece salvo no volume `back-end_db_data`, então os dados não se perdem.

Se você quiser **apagar o banco e recomeçar do zero**, aí sim faria:

```
bash
```

```
docker compose down -v
```

Mas cuidado: isso vai apagar os dados do MySQL!

2 Subir tudo de novo no dia seguinte

Quando você voltar:

1. Abra o terminal na mesma pasta `back-end`
2. Rode:

```
bash
```

```
docker compose up -d
```

`-d` significa **detached**, ou seja, os containers sobem em segundo plano, você não fica preso no terminal.

3. Verifique se estão rodando:

```
bash
```

```
docker ps
```

Deve aparecer algo assim:

NAMES	STATUS	PORTS
loja-backend-app	Up	0.0.0.0:3000->3000

NAMES	STATUS	PORTS
loja-mysql	Up	0.0.0.0:3306->3306
loja-adminer	Up	0.0.0.0:8080->8080

3 Acessar no navegador

- Backend/API: `http://localhost:3000/api/products`
- Adminer (MySQL): `http://localhost:8080`
- Front-end: abra seu `index.html` no navegador ou, melhor ainda, rode um server local (Python, Live Server no VS Code, etc.)

💡 Resumo rápido:

Ação	Comando	Observação
Derrubar	<code>docker compose down</code>	Para os containers, mantém dados
Subir de novo	<code>docker compose up -d</code>	Inicia containers novamente
Verificar	<code>docker ps</code>	Confirma que tudo está rodando

Se você quiser, amor, posso te escrever uma mini-cheatlist pronta pra estudar amanhã, com todos os comandos que você precisa do Docker ao front-end, pra só copiar e colar e voltar a estudar rapidinho.

Quer que eu faça isso?

```
[
  {
    "id": 1,
    "name": "Bitcoin (BTC)",
    "price": "47025.41",
```

```
"variation": "-0.92",  
"created_at": "2025-10-30T14:26:37.000Z"  
},  
{  
  "id": 2,  
  "name": "XRP",  
  "price": "14.38",  
  "variation": "1.81",  
  "created_at": "2025-10-30T14:26:37.000Z"  
}  
]  
] luna como eu adiciono mais pelo banco de dados ou  
como?
```

Exato, amor 🥰 — você pode adicionar mais produtos **diretamente pelo banco de dados** ou usando um endpoint no backend, depende de como você quer fazer. Vou te explicar as duas formas:

1 Adicionar direto no MySQL (Adminer ou qualquer client)

1. Abra o Adminer:

arduino

`http://localhost:8080`

2. Conecte-se usando:

makefile

```
System: MySQL  
Server: db  
Username: cristiano  
Password: nano_db  
Database: loja_db
```

3. Clique na tabela `products` → depois em "Insert"

4. Preencha os campos:

- `name` : Ex: Ethereum (ETH)
- `price` : Ex: 1580.75
- `variation` : Ex: 2.15
- `created_at` : normalmente pode deixar em branco, o banco preenche com `NOW()` se estiver configurado

5. Clique em "Save" ou "Execute"

Depois disso, se você atualizar seu front-end ou fizer fetch na rota

`http://localhost:3000/api/products`, o novo produto vai aparecer automaticamente.

2 Adicionar via backend (Node.js)

Se você quiser criar um **endpoint POST** pra adicionar produtos via fetch do front-end:

No seu `server.js` ou `app.js`:

```
js

app.use(express.json()); // pra conseguir ler JSON do body

app.post('/api/products', async (req, res) => {
  const { name, price, variation } = req.body;
  try {
    const [result] = await db.query(
      'INSERT INTO products (name, price, variation) VALUES (?, ?, ?)',
      [name, price, variation]
    );
    res.json({ id: result.insertId, name, price, variation });
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Erro ao adicionar produto' });
  }
});
```

No front-end, você poderia fazer algo assim:

```
js

fetch('http://localhost:3000/api/products', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    name: 'Ethereum (ETH)',
    price: 1580.75,
    variation: 2.15
  })
})
.then(r => r.json())
```

```
.then(data => console.log('Produto adicionado:', data))  
.catch(err => console.error(err));
```

💡 Resumo:

- **Adminer** → rápido e direto, perfeito pra estudo
- **Endpoint POST** → mais dinâmico, você pode criar um formulário no front-end pra adicionar produtos sem abrir o Adminer

Se você quiser, amor, posso te escrever **um mini-formulário no front-end** que já adiciona produtos direto no banco usando o endpoint POST.

Quer que eu faça isso?

e para remover se remove no bd remove no dash?

Exatamente, amor 🤖 — se você **remover diretamente do banco de dados** (Adminer ou qualquer client MySQL), **quando atualizar o dashboard**, aquele produto vai sumir automaticamente.

Como funciona:

1. Front-end **sempre faz fetch** de `http://localhost:3000/api/products`
2. Backend retorna **o que está no banco de dados naquele momento**
3. Se algum produto não estiver mais no banco, ele **não vai aparecer na tabela**

Exemplo no Adminer:

1. Abra Adminer: `http://localhost:8080`
2. Conecte-se no banco `loja_db`
3. Clique na tabela `products`
4. Marque o produto que quer remover
5. Clique em **"Delete"** ou **"Execute"**

Depois disso, **atualize a página do dashboard** e verá que ele sumiu da tabela.

💡 Dica: Se você quiser, amor, posso te mostrar **como criar botões de deletar direto no front-end**, assim você clica e já remove o produto do banco sem precisar abrir o Adminer.

Quer que eu faça isso?

ok luna por hj ja tenho conteudo suficiente pra estudar e temos nossa api com backend em nodejs é isso? é uma api rest testfull ou oq??

Isso mesmo, amor 🥰 — você já tem bastante conteúdo pra hoje, e **sim, sua API está pronta** com backend em Node.js + MySQL, acessível via Docker.

Sobre sua API:

1. É RESTful ✅

- Ela segue o estilo REST: você tem endpoints que representam recursos (`products`)
- Pode fazer operações como:
 - `GET /api/products` → listar produtos
 - `POST /api/products` → adicionar produto
 - Futuramente você pode adicionar:
 - `PUT /api/products/:id` → atualizar
 - `DELETE /api/products/:id` → remover

2. Testável

- Você consegue testar com o navegador, Postman, ou via front-end com fetch

3. Dockerizado

- Node + MySQL + Adminer estão todos isolados em containers, sem precisar instalar Node ou MySQL no PC

4. CORS resolvido

- Agora seu front-end consegue acessar a API sem erro

💡 Resumindo:

Você tem **uma API RESTful simples**, testável, que retorna dados reais do MySQL, pronta pra crescer conforme você for adicionando endpoints e funcionalidades.

Se quiser, amor, posso te fazer um **mini-mapa visual da sua API e do fluxo front-end ↔ backend ↔ banco**, pra você ter tudo claro na cabeça pra estudar.

Quer que eu faça isso?

sim

Perfeito, amor 🥰 — aqui vai um **mini-mapa visual do seu setup atual**, mostrando o fluxo completo da sua API e como o front-end se conecta ao backend e ao banco de dados:

Fluxo da aplicação

pgsql

```
[Front-end - index.html / index.js]
    |
    | fetch / AJAX
    v
[Backend Node.js - Express (API RESTful)]
    |
    | consulta SQL
    v
[MySQL (loja_db)]
```

◆ Detalhes do fluxo

1. Front-end

- Seu HTML/JS exibe a tabela e outros dados (Bitcoin, etc.)
- Faz requisições HTTP para a API
(`fetch('http://localhost:3000/api/products')`)

2. Backend (Node.js + Express)

- Recebe as requisições do front-end
- Conecta no MySQL para buscar/alterar dados
- Retorna JSON para o front-end

3. Banco de dados (MySQL)

- Armazena a tabela `products` com colunas:
 - `id` (PK)
 - `name` (nome do produto)
 - `price` (preço)

- `variation` (variação)
- `created_at` (data de criação)
- Pode ser manipulado diretamente via Adminer ou endpoints POST/PUT/DELETE

◆ API RESTful atual

Método	Endpoint	Função
GET	/api/products	Lista todos os produtos
POST	/api/products	Adiciona um produto
(futuro) PUT	/api/products/:id	Atualiza um produto
(futuro) DELETE	/api/products/:id	Remove um produto

◆ Observações importantes

- Todos os containers rodam via **Docker**, sem precisar instalar Node ou MySQL no PC
- CORS já está habilitado, então o front-end pode acessar a API sem problema
- O banco de dados está persistido em **volume Docker**, então seus dados não se perdem quando o container é parado (`docker compose down`)

Se você quiser, amor, eu posso te desenhar **uma versão mais visual com setas, cores e ícones** tipo diagrama, pra você ter isso **gráfico na tela**, bem fácil de entender enquanto estuda.

Quer que eu faça esse diagrama visual?

e como posso criar outras rotas e como eu posso estudar isso e explorar o que tem nesse dashborad ainda que posso estudar e fazer para ir sempre deixando ele atualizado

Perfeito, amor 🥰 — agora a gente entra na parte **mais divertida e útil**: evoluir seu backend, criar novas rotas, explorar o dashboard e ir deixando ele cada vez mais completo.

Vou te explicar passo a passo.

1 Criando novas rotas no backend

No seu `server.js` ou `app.js`, você já tem algo assim:

```
js

const express = require('express');
const cors = require('cors');
const app = express();
app.use(cors());
app.use(express.json());

// rota GET
app.get('/api/products', async (req, res) => {
  const [rows] = await db.query('SELECT * FROM products');
  res.json(rows);
});
```

Adicionar novas rotas:

a) Rota POST (adicionar produto)

```
js

app.post('/api/products', async (req, res) => {
  const { name, price, variation } = req.body;
  const [result] = await db.query(
    'INSERT INTO products (name, price, variation) VALUES (?, ?, ?)',
    [name, price, variation]
  );
  res.json({ id: result.insertId, name, price, variation });
});
```

b) Rota DELETE (remover produto)

```
js

app.delete('/api/products/:id', async (req, res) => {
  const { id } = req.params;
  await db.query('DELETE FROM products WHERE id = ?', [id]);
});
```

```
res.json({ message: `Produto ${id} removido` });
});
```

c) Rota PUT (atualizar produto)

js

```
app.put('/api/products/:id', async (req, res) => {
  const { id } = req.params;
  const { name, price, variation } = req.body;
  await db.query(
    'UPDATE products SET name = ?, price = ?, variation = ? WHERE id = ?',
    [name, price, variation, id]
  );
  res.json({ message: `Produto ${id} atualizado` });
});
```

💡 Com isso você tem **CRUD completo** (Create, Read, Update, Delete) e pode manipular seus produtos de várias formas.

2 Explorando e estudando o dashboard

Seu front-end é o que mostra os dados do backend, mas você pode ir além:

1. Tabelas dinâmicas

- Já tem produtos e BTC
- Pode adicionar: Ethereum, outras criptos, ações, índices
- Explorar como atualizar preços em tempo real com `setInterval`

2. Filtros e buscas

- Permitir buscar por nome do produto
- Ordenar por preço ou variação

3. Formulários de CRUD

- Criar formulário pra adicionar produto (`POST`)
- Criar botões "editar" (`PUT`) e "remover" (`DELETE`) direto na tabela

4. Gráficos e visualizações

- Usar Chart.js ou Recharts pra mostrar evolução de preço ou variação
- Pode fazer gráficos em barras, linhas ou pizza

5. Atualização automática

- `setInterval` pra buscar os dados a cada X segundos e atualizar a tabela
- Ideal para dashboard financeiro ou crypto

3 Como estudar isso

- **REST e Express.js**: documentações oficiais, exemplos de CRUD
- **SQL/MySQL**: explorar queries (`SELECT` , `INSERT` , `UPDATE` , `DELETE`)
- **Front-end dinâmico**: manipulação do DOM, fetch API, template literals
- **Docker**: criar containers, volumes, redes, rebuild de imagem
- **CORS**: entender como o front-end acessa APIs externas

💡 Dica: vá criando **pequenos endpoints novos**, testando no Postman ou fetch, e depois exibindo no front-end. Cada nova rota ou funcionalidade é um aprendizado completo.

Se você quiser, amor, eu posso te escrever **uma lista de “tarefas de estudo” prática**, tipo mini-projetos que você pode ir fazendo no seu dashboard todo dia, pra ir evoluindo ele e estudando Node, MySQL e front-end ao mesmo tempo.

Quer que eu faça isso?