Tutorial

Professor

Marcos de Melo

Sumário

Aula 1 – Fundamentos do NodeJS

Nesta primeira aula, vamos conhecer os fundamentos da plataforma Node.js. Veremos o que é essa tecnologia, como realizar sua instalação e entenderemos seu funcionamento na prática.

O que é o Node.js?

Com certeza! O Node.js costuma causar uma certa confusão no início porque ele não é uma linguagem de programação, mas sim um "ambiente" que permite que o JavaScript (que nasceu para rodar apenas no navegador) ganhe superpoderes no computador ou servidor.

- Ele é um ambiente que permite a execução de código JavaScript fora de um navegador.
- É construído sobre o motor **V8** do Google Chrome (o que o torna extremamente rápido).
- É utilizado para construir APIs (back-ends)
- É um **ambiente de execução** (runtime) de código aberto.
- Permite que você use JavaScript no **Backend** (servidores, bancos de dados, arquivos).

Analogia

Imagine que o **JavaScript** é um piloto de corrida. Por muito tempo, ele só podia correr em uma pista específica: o **Navegador** (Chrome, Firefox, Edge). O **Node.js** é como se alguém tivesse construído uma estrada de alta performance fora da pista, permitindo que esse mesmo piloto corra por cidades, desertos e montanhas.

Tecnicamente falando:

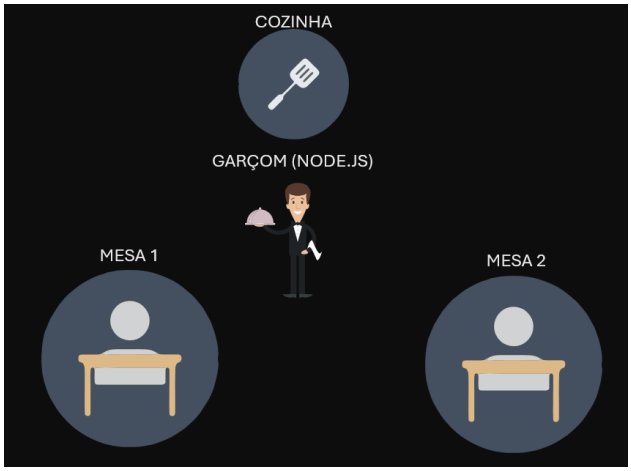
Características principais:

- **Single-threaded:** Ele executa uma tarefa por vez em um único "fio" de processamento.
- **I/O Não Bloqueante:** Diferente de outras linguagens, o Node não fica parado esperando um arquivo carregar ou uma busca no banco de dados terminar. Ele envia o pedido e continua trabalhando em outras tarefas até que a resposta chegue.

Como o NODE.JS funciona?

- Possui apenas um núcleo (single threaded)
- Suporta várias operações simultâneas (non-blocking)

Exemplo



Para que serve o Node.js?

O Node.js brilha em cenários onde a velocidade e a capacidade de lidar com muitas conexões simultâneas são essenciais. Ele é ideal para:

1. **APIs REST:** Criar a "ponte" que conecta o banco de dados ao seu aplicativo.
2. **Aplicações em Tempo Real:** Chats (como o WhatsApp Web), jogos online ou ferramentas colaborativas (como o Google Docs).
3. **Streaming:** Plataformas de vídeo ou áudio (como Netflix e Spotify utilizam partes em Node).
4. **Ferramentas de Automação:** Scripts para otimizar tarefas repetitivas no computador.
5. **Ferramentas de Linha de Comando:** Automações que rodam direto no seu terminal.

Por que aprender Node.js hoje?

- **Linguagem Única:** Você usa JavaScript no Front-end (com React/Vue) e no Back-end. É o famoso "Full Stack".
- **Comunidade Gigante:** O **NPM** (Node Package Manager) é o maior registro de bibliotecas do mundo. Precisa de algo? Provavelmente alguém já criou um pacote para o Node.
- **Site do NPM** (Node Package Manager): https://www.npmjs.com/
- **Mercado:** Empresas como Uber, LinkedIn, PayPal e NASA utilizam Node.js em seus sistemas críticos.

Por que usar Node.js e não outra tecnologia?

Para fechar a aula, é importante destacar os diferenciais competitivos no mercado:

- **Isomorfismo:** O dev usa a mesma linguagem (JS) no Front-end e no Back-end.
- **Velocidade de Escrita:** O que levaria 50 linhas em Java/C#, o Node resolve em 10.
- **Escalabilidade:** Grandes empresas (PayPal, Uber) migraram para o Node para aguentar milhões de acessos simultâneos com baixo consumo de memória.

Exemplo

```
const path = require('path');
const meuArquivo = require('./meu-arquivo.js');
```

Módulos no Node.JS

- Podemos criar nossos próprios módulos (nossos arquivos) e importá-los
- O Node.js vem módulos pré-instalados (patch, fs, http etc.)

NPM – Node Package Manager

- Podemos instalar módulos de terceiros utilizando o NPM
- Esses módulos são armazenados em uma pasta chamada "**node_modules**"
- Um arquivo chamado "**package.json**" lista todos os módulos que instalamos pelo NPM

Exemplo

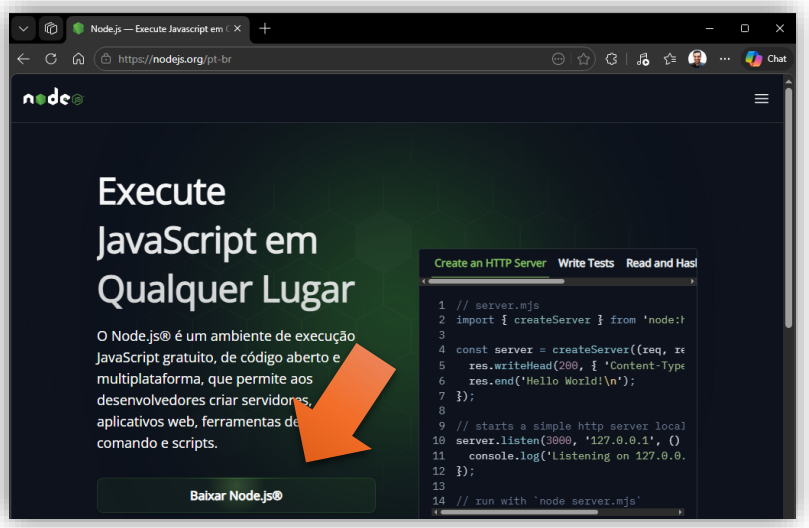
```
npm init // cria o package.json
npm -g install nodemon // instala um pacote globalmente
npm install express // instala um pacote localmente
```

Instalação do Node.JS

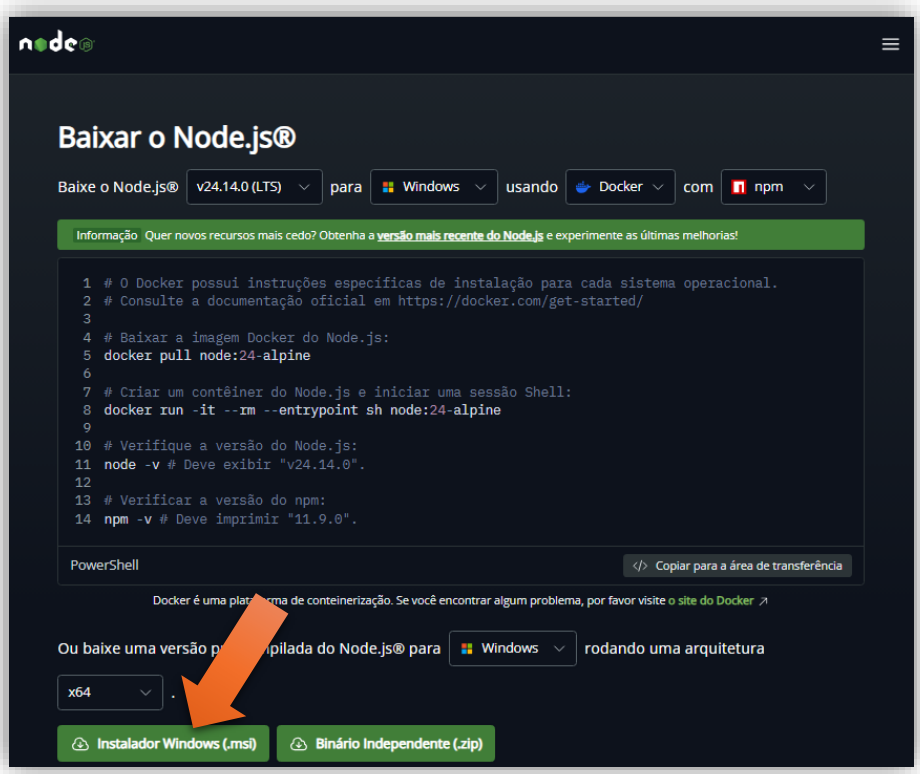
Se você quer ver o Node em ação agora, siga este roteiro de instalação:

Passo A: Instalação

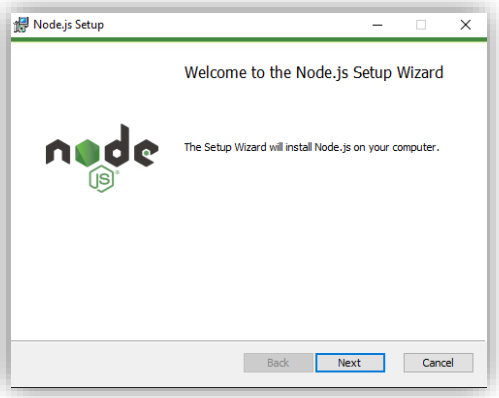
1. Vá ao site oficial nodejs.org.

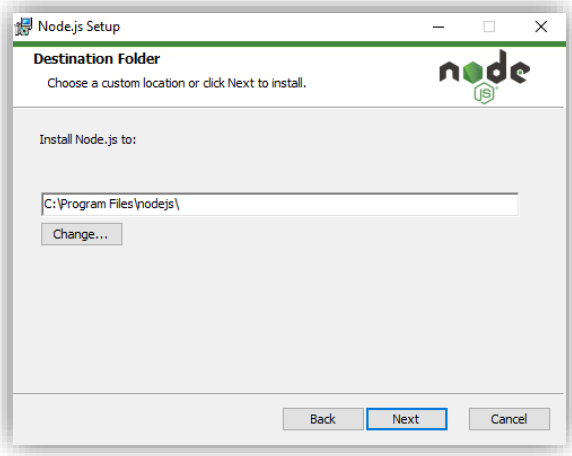


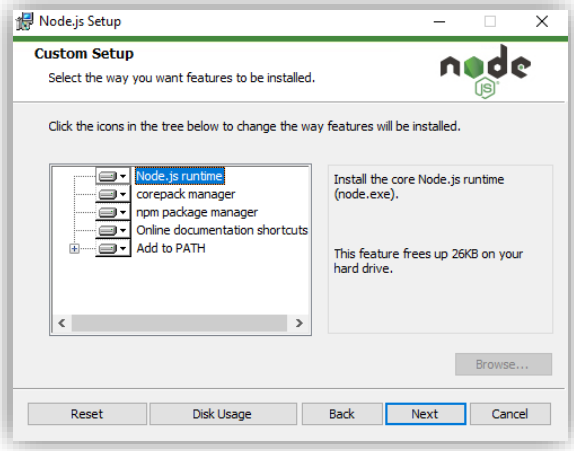
2. Baixe a versão **LTS** (Long Term Support), que é a mais estável para ensino e produção. Se estiver no sistema operacional Windows, baixar a versão pré-compilada lts (.msi) é a maneira mais fácil de instalar o node.

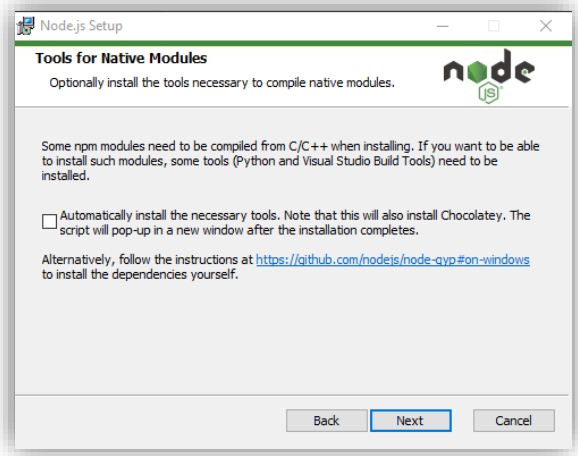


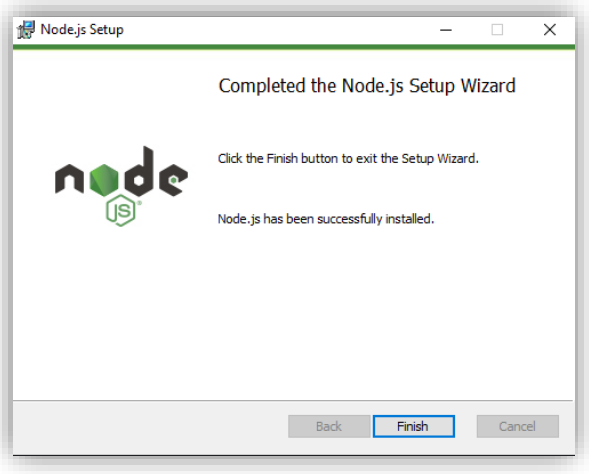
3. Após baixar o instalador do nodejs, execute-o para a instalação começar. A instalação é bem intuitiva e apesar de ser passo a passo decidindo o que instalar, geralmente é só avançar até concluir a instalação.




Node.js Setup
Destination Folder
Choose a custom location or click Next to install.
Install Node.js to:
C:\Program Files\nodejs\
Change...
Back
Next
Cancel


Node.js Setup
Custom Setup
Select the way you want features to be installed.
Click the icons in the tree below to change the way features will be installed.
Node.js runtime
corepack manager
npm package manager
Online documentation shortcuts
Add to PATH
Install the core Node.js runtime (node.exe).
This feature frees up 26KB on your hard drive.
Browse...
Reset
Disk Usage
Back
Next
Cancel


Node.js Setup
Tools for Native Modules
Optionally install the tools necessary to compile native modules.
Some npm modules need to be compiled from C/C++ when installing. If you want to be able to install such modules, some tools (Python and Visual Studio Build Tools) need to be installed.
Automatically install the necessary tools. Note that this will also install Chocolatey. The script will pop-up in a new window after the installation completes.
Alternatively, follow the instructions at https://github.com/nodejs/node-gyp#on-windows to install the dependencies yourself.
Back
Next
Cancel

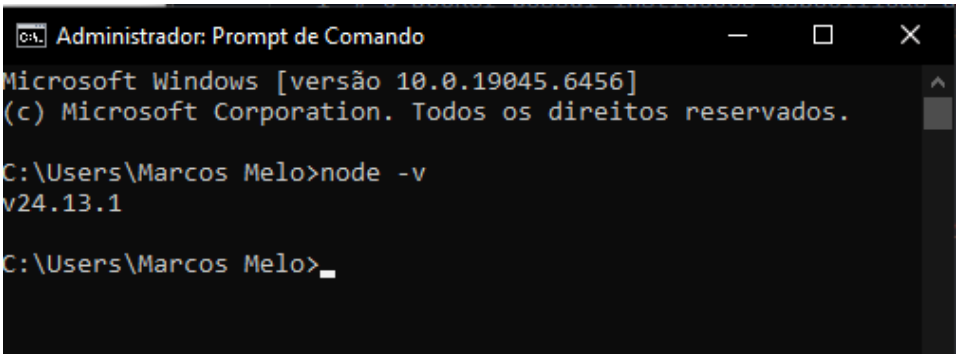


4. Após instalar, abra o seu terminal (CMD ou PowerShell) e digite:

Terminal

```
node -v
```

Se aparecer a versão, está tudo certo, seu nodejs está instalado.



Passo B: Criando seu primeiro servidor http no node

Para criar um servidor http não é necessário um servidor externo (como Apache). O Node **é** o servidor.

1. Crie uma pasta para o projeto, ex: **fundamentos-nodejs**. Se você esta no terminal, crie a pasta com o comando a **mkdir fundamentos-nodejs**;

```
C:\Users\Marcos Melo\Git>mkdir fundamentos-nodejs
```

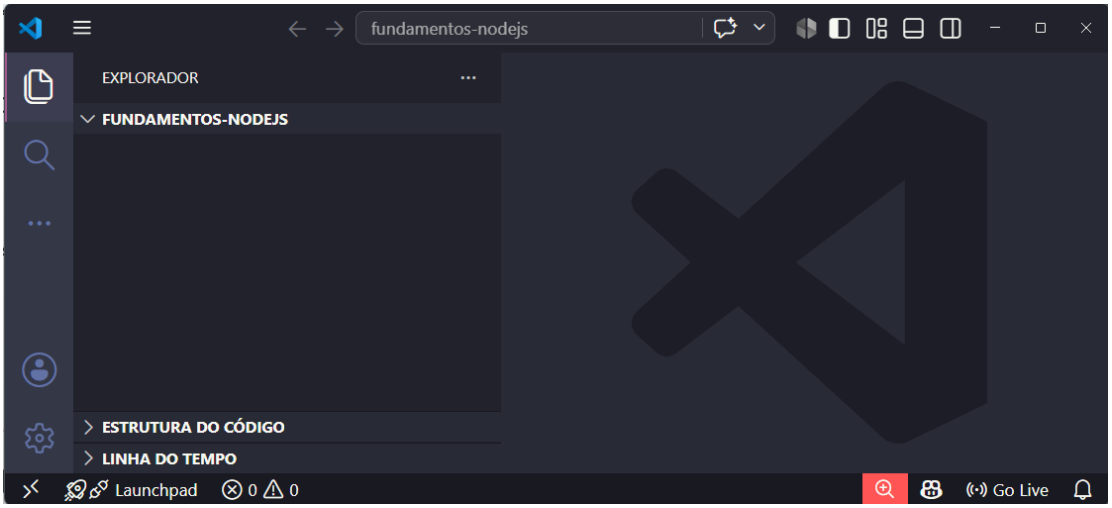
2. Entre dentro desta pasta com o comando **cd fundamentos-nodejs;**

```
C:\Users\Marcos Melo\Git>cd fundamentos-nodejs
```

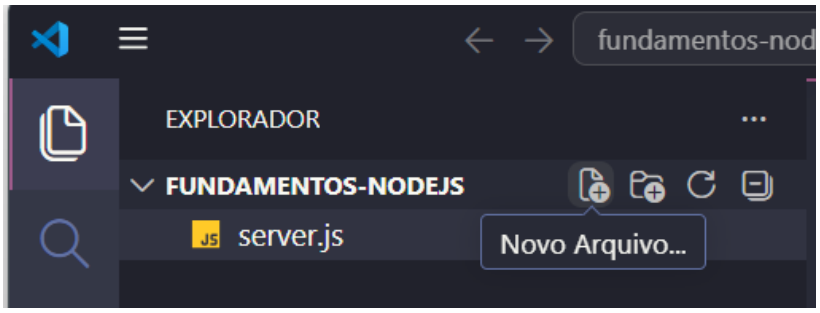
3. Agora que está dentro da pasta no terminal, abra esta pasta no VS CODE (Editor de Códigos) com o comando **code .**

```
C:\Users\Marcos Melo\Git\fundamentos-nodejs>code .
```

4. Se o VS Code abriu com a pasta fundamentos-nodejs já gerenciada por ele, você já pode fechar o terminal e ficar somente no VS Code.



5. Crie um arquivo chamado **server.js** na raiz da pasta.



6. Digite o seguinte código (exemplo simples de servidor HTTP):

```
const http = require('http');

const server = http.createServer((req, res)=>{
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plan');
    res.end('Ola! Este e o meu primeiro servidor HTTP');
});

server.listen(3000, 'localhost', ()=>{
    console.log('Servidor rodando em http://localhost:3000/');
});
```

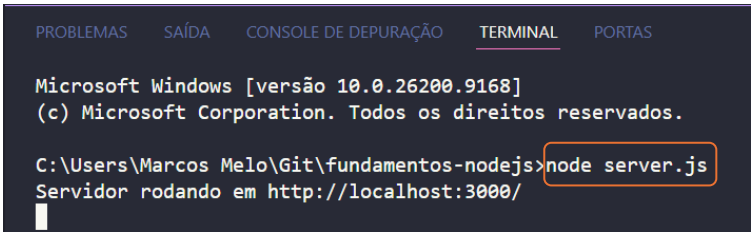
Passo C: Executando

Podemos acessar o terminal de comandos de texto o CMD dentro do VS CODE com o comando; **CTRL + J**.

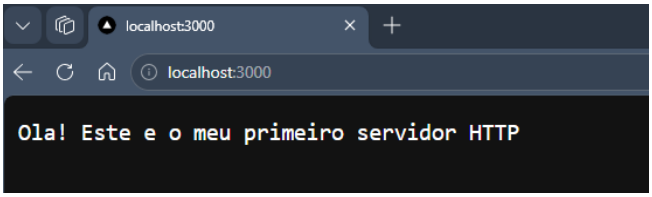
No terminal, dentro da pasta do arquivo, digite:

Terminal

```
node server.js
```

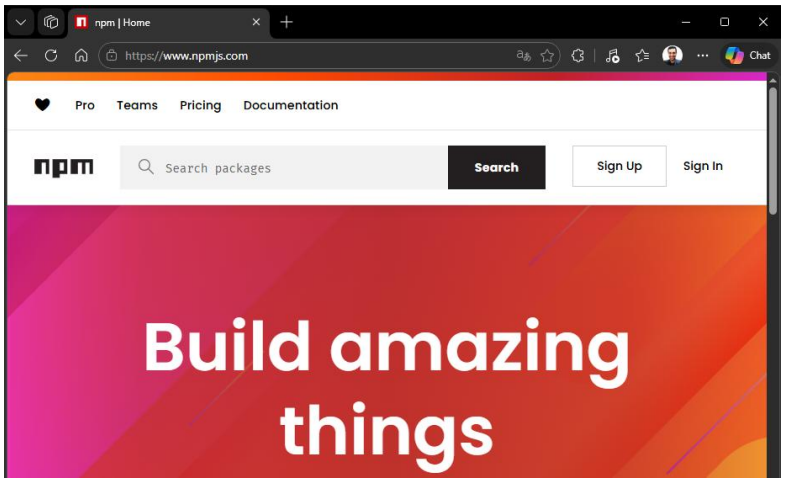


Agora abra o navegador e acesse **http://localhost:3000**. Você verá a mensagem enviada pelo seu servidor.



O Ecossistema NPM

Ao instalar o Node, você ganha o **NPM (Node Package Manager)**. Ele é a maior biblioteca de códigos do mundo. Precisa enviar e-mail? Existe um pacote. Precisa de um sistema de login? Existe um pacote. Isso acelera absurdamente o desenvolvimento.

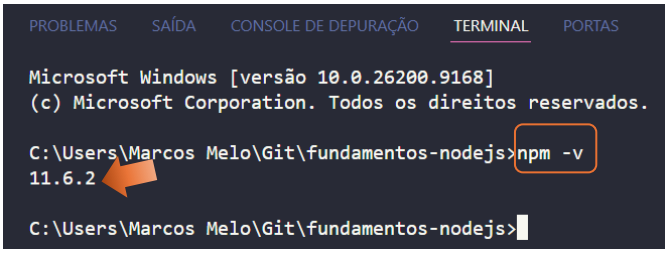


O **npm** já vem instalado junto com o node, e para verificar a versão do NPM digite no terminal:

Terminal

```
npm -v
```

se o **npm** estiver instalado deverá aparecer a versão instalada.



Usando o NPM

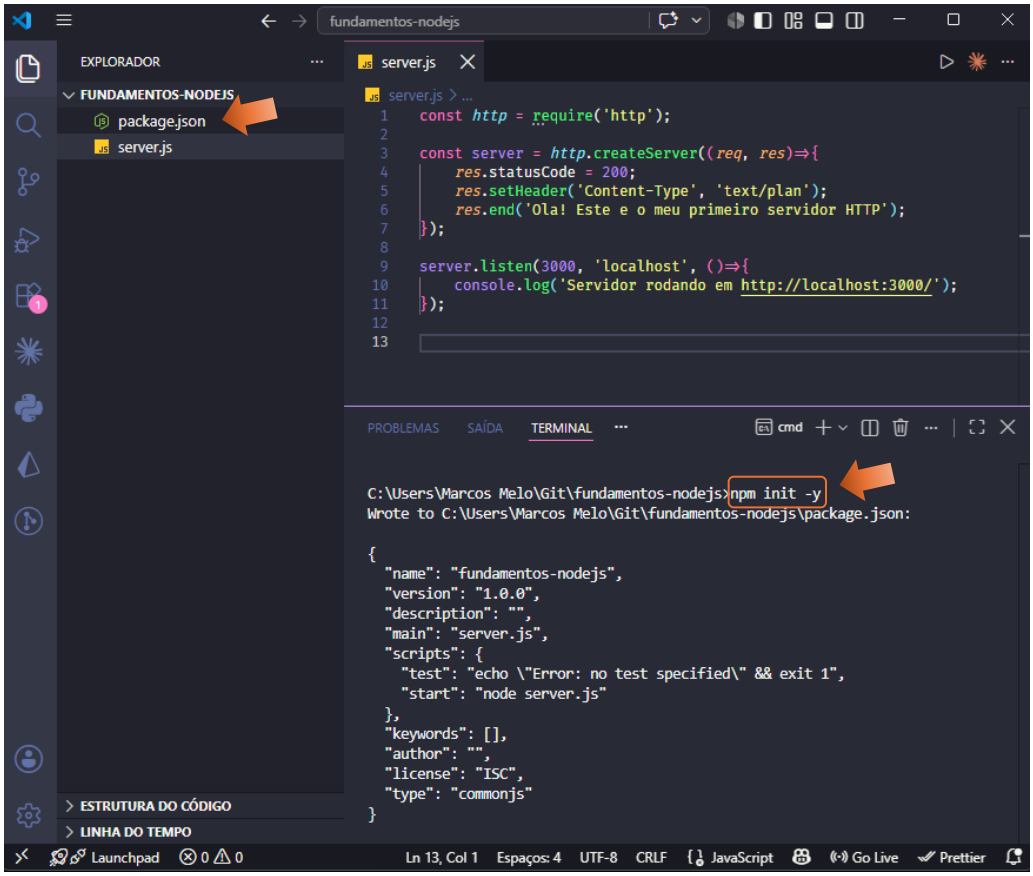
Para importar os vários pacotes de bibliotecas Javascript em um projeto servidor, api em node, é preciso inicializar o **npm** dentro da pasta do projeto.

Para iniciar o **npm** dentro da pasta do projeto digite no terminal estando dentro da pasta do projeto, o comando:

Terminal

```
npm init -y
```

isso criará o arquivo **package.json,** este arquivo irá controlar a gestão de pacotes instalados no projeto.



O arquivo **package.json** é o "coração" de qualquer projeto Node.js. Ele funciona como um **manifesto**, contendo os metadados necessários para gerenciar as dependências, scripts e informações gerais do projeto.

Aqui estão os pontos principais para entender sua função:

1. Metadados do Projeto

Ele define o que o seu projeto é. Sem esse arquivo, o Node e o gerenciador de pacotes (NPM ou Yarn) não sabem como lidar com o código.

- **Nome e Versão:** Essenciais se você pretende publicar o pacote.
- **Main:** Define o ponto de entrada da aplicação (geralmente index.js).

2. Gerenciamento de Dependências

Esta é a parte mais importante. O arquivo lista todas as bibliotecas externas que seu projeto precisa para rodar:

- **dependencies:** Pacotes necessários para a aplicação funcionar em produção (ex: Express, React).
- **devDependencies:** Ferramentas usadas apenas durante o desenvolvimento (ex: Jest para testes, Nodemon).

Por que isso é útil? Em vez de enviar a pasta node_modules (que é enorme) para o GitHub, você envia apenas o package.json. Quando outra pessoa baixar o projeto, ela só precisa rodar npm install para que todas as bibliotecas sejam baixadas automaticamente.

3. Scripts de Automação

O campo "scripts" permite criar atalhos para comandos complexos do terminal.

- Exemplo: Em vez de digitar **node server.js**, você pode configurar um script "start": "node server.js" e rodar apenas npm start.

Estrutura Visual do Arquivo

Na chave scripts crie os scripts dev e start como mostrado a seguir;

Após criar os scripts, digite no terminal o script dev para iniciar seu servidor como no exemplo a seguir;

Terminal

```
npm run dev
```

Definindo o sistema de importação de módulos

Essa é uma das dúvidas mais comuns hoje em dia, pois o ecossistema Node.js está em plena transição. Essa chave no package.json define como o Node deve interpretar os seus arquivos .js.

Basicamente, ela decide qual **sistema de módulos** será o padrão do seu projeto.

1. "type": "commonjs" (O Padrão Antigo)

Se você não colocar nada no package.json, o Node assume que o projeto é CommonJS. É o sistema que o Node usa desde o seu nascimento.

- **Sintaxe:** Usa require() para importar e module.exports para exportar.
- **Carregamento:** É **síncrono**. Os módulos são carregados um por um, na ordem em que aparecem.
- **Comportamento:** Permite importar arquivos sem extensão (ex: require('./auth')) e oferece variáveis globais automáticas como __dirname e __filename.

JavaScript

```
// Exemplo CommonJS
const express = require('express');
module.exports = { app };
```

2. "type": "module" (O Padrão Moderno / ESM)

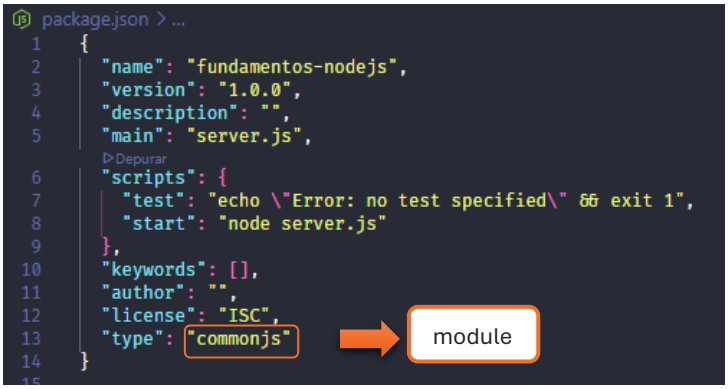
Ao definir isso, você diz ao Node para usar **ECMAScript Modules (ESM)**, que é o padrão oficial do JavaScript (o mesmo usado nos navegadores e em frameworks como React/Vue).

- **Sintaxe:** Usa import e export.
- **Carregamento:** É **assíncrono**, o que é mais eficiente para o desempenho em aplicações modernas.
- **Rigor:** Exige que você coloque a extensão do arquivo na importação
  ex: import {auth} from './auth.js'.
- **Limitação:** Não possui __dirname por padrão (você precisa usar import.meta.url para obter o caminho do arquivo).

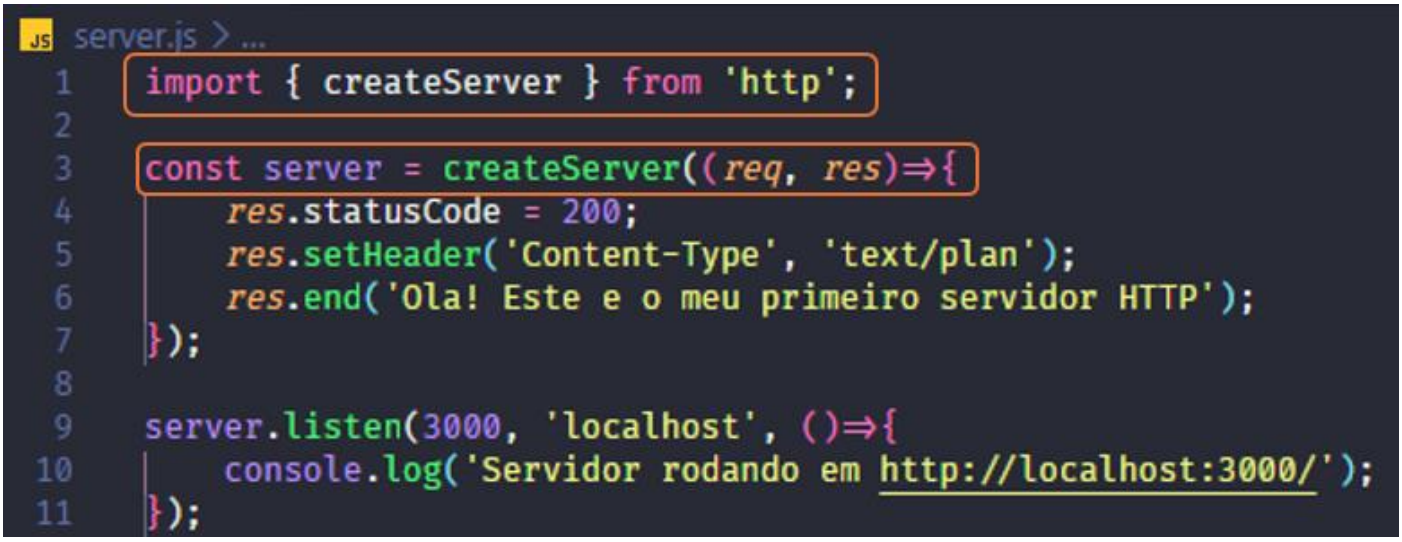
JavaScript

```
// Exemplo ESM
import express from 'express';
export const app = express();
```

Mudando o tipo **commonjs** para **module**



Ao fazer esta mudança seu projeto vai parar de funcionar devido a mudança no tipo de importação de módulos, então vamos mudar no arquivo, as linhas a seguir;



Após esta mudança o arquivo voltara a funcionar normalmente.

Quiz de Perguntas

De acordo com a aula, como habilitamos o uso das importações utilizando o ESModules?

- ☐ Alterando a extensão do arquivo para .cjs
- ☐ Adicionando ao arquivo *package.json* a propriedade "*type*" com o valor de "*module*"
- ☐ Definindo no arquivo jsconfig.son a propriedade "type" com o valor de "module"
- ☐ Nenhuma das anteriores

O prefixo node: na importação de um módulo serve para informar que esse módulo é interno do Node.js:

- ☐ Verdadeiro
- ☐ Falso

Para que serve o módulo http do Node.js?

- ☐ Unicamente para lidar com conexão do tipo WebSocket
- ☐ Para criar e lidar com as requisições em uma porta específica
- ☐ Realizar requisições HTTP para outros endpoints
- ☐ Nenhuma das anteriores

Sobre o parâmetro request é possível afirmar que: Através desse objeto é possível obter todas as informações presente na requisição, como dados enviados em JSON, parâmetros de rota nomeados e não nomeados, entre outros.

- ☐ Verdadeiro
- ☐ Falso

Aula 2 - Rotas

Bom, turma, agora que entendemos os conceitos iniciais, vamos conhecer as **rotas HTTP**, que funcionam como a espinha dorsal de qualquer API back-end, definindo como as requisições são processadas.

Rotas HTTP em Node.js

O que é uma rota?

Uma **rota** é a combinação de dois elementos que dizem ao servidor **o que fazer** quando uma requisição chega:

MÉTODO HTTP + CAMINHO (url=endpoint)

Exemplo:

GET /produtos

Isso significa: "quando alguém fizer uma requisição do tipo GET no caminho /produtos, execute essa função aqui".

Os métodos HTTP (verbos)

Cada rota tem um **método** que indica a *intenção* da requisição. É como um verbo de uma frase: "eu quero **buscar**", "eu quero **criar**", "eu quero **apagar**"...

Método	Intenção	Exemplo de uso
GET	Buscar/ler dados	Listar produtos
POST	Criar um novo recurso	Cadastrar um produto
PUT	Atualizar um recurso **por completo**	Editar todos os dados de um produto
PATCH	Atualizar um recurso **parcialmente**	Editar só o preço de um produto
DELETE	Remover um recurso	Excluir um produto

Pergunta clássica de prova: qual a diferença entre PUT e PATCH?
PUT → substitui o recurso inteiro (mesmo que você não envie um campo, ele pode ser apagado/zerado)
PATCH → altera só o que foi enviado, o resto permanece igual

No nosso exemplo, supondo que nosso projeto tenha duas rotas para cadastrar e listar usuários, se acrescentarmos as linhas de código em destaque, ficaria assim:

```
server.js > ...
1   import * as http from 'http';
2
3   const server = http.createServer((req, res) => {
4
5     // const url = req.url;
6     // const method = req.method;
7
8     const { url, method } = req;
9
10    console.log(method, url);
11
12    res.statusCode = 200;
13    res.setHeader('Content-Type', 'text/plain');
14
15    if (method === 'GET' && url === '/users') {
16      return res.end('Listagem de usuários\n');
17    }
18    if (method === 'POST' && url === '/users') {
19      return res.end('Cadastro de usuários\n');
20    }
21
22    return res.end('Ola! Este e um servidor rodando em Node');
23  });
24
25  server.listen(3000,'127.0.0.1', () => {
26    console.log('Servidor rodando em http://127.0.0.1:3000/');
27  });
```

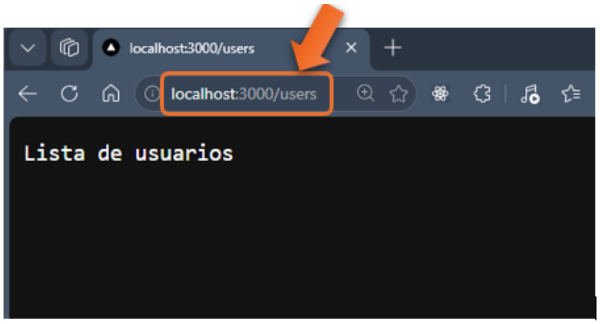
Repare que foi necessário utilizar uma estrutura condicional **IF** para verificar o **método** e a **url**, se no método for igual a **GET** e a **url** for igual a **/users** a **Listagem de usuários** será mostrada, e no próximo **IF** se o método for igual a **POST** e a **url** for igual a **/users** o **Cadastro de usuários** será mostrada

Fazendo requisição nas Rotas

Com o servidor em execução, as rotas http são acessadas por um cliente http como o Browser (Navegador) ou um APP Mobile entre outros.

A rota **get / users** pode ser acessada na barra de endereços URL no navegador, justamente porque rotas definidas como GET (Listar/ Pegar) iram retornar informações no navegador através da URL na barra de endereço.

Exemplo:



Nota: todo endereço url digitado na barra de endereço url do navegador é definido como uma rota GET, que é a combinação do método GET mais a url http://localhost:3000/users
As demais rotas precisam de implementações no front-end para serem acessadas Ex: POST, PUT, PATCH, DELETE e outras.

Arquivo de Rotas

As rotas criadas anteriormente foram feitas para serem acessadas por requisições http pelo navegador em um front-end (Navegador), porém, ainda não temos um sistema front-end, então vamos usar uma extensão do VSCODE para fazer as requisições http no lugar de um sistema front-end e então testar as rotas.

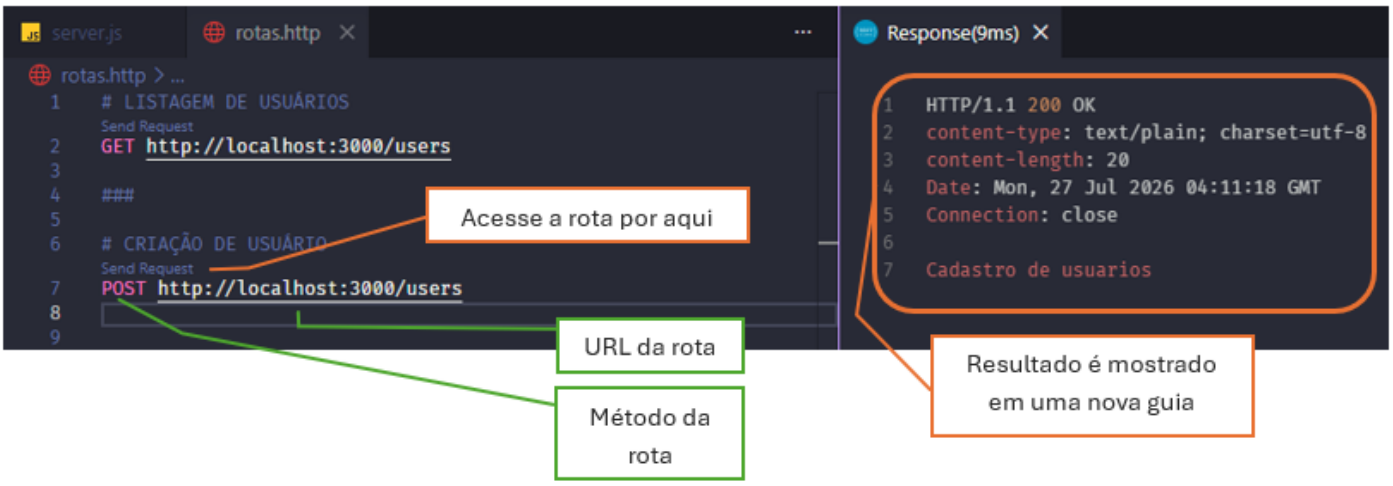
Certifique-se de ter instalado a extensão ***REST Client*** no VSCODE.

Com esta extensão instalada vamos criar um arquivo chamado **rotas.http,** neste arquivo vamos colocar todas as requisições http que criamos, implementando os métodos com url, para definir por qual rotas seguiremos.

Digite o código a seguir no arquivo rotas.http

```
# LISTAGEM DE USUÁRIOS
GET http://localhost:3000/users
###
# CRIAÇÃO DE USUÁRIO
POST http://localhost:3000/users
```

Após digitar o código acima, se foi instalado corretamente a extensão REST Client, acima de cada rota será incluído o link ***Send Request*** que ao ser clicado executara a rota no servidor e mostrara em uma guia a direita a resposta do servidor em formato JSON, texto ou até mesmo html dependendo do modo de exibição que foi configurado.



Aula 3 – Montando Servidor HTTP

Fala, pessoal! Nesta aula, veremos como integrar bibliotecas JavaScript de terceiros aos nossos projetos, aproveitando funcionalidades já desenvolvidas pela comunidade.

Criando o servidor utilizando Framework

Acabamos de criar um servidor nativo usando somente o Node, que é bem limitado, na verdade os projetos de servidor em node são criados utilizando frameworks por possuírem mais recursos e melhor desempenho.

Usar o Node.js "puro" (nativo) para criar um servidor é como construir um carro comprando peça por peça: você tem controle total, mas gasta um tempo enorme montando coisas básicas que todo carro precisa, como o chassi ou o painel.

Um framework como o **Fastify** (ou Express) já te entrega o "carro montado", permitindo que você foque apenas em dirigir (a lógica do seu negócio).

Por que usar Fastify em vez do Node Nativo?

1. Abstração e Facilidade (Menos Código)

No Node nativo, você precisa lidar manualmente com fluxos de dados (streams), analisar cabeçalhos de requisição e gerenciar rotas com if/else complexos.

- **Nativo:** Você escreve 20 linhas para ler um JSON básico.
- **Fastify:** Você escreve 1 linha; o framework já entende o JSON e o entrega pronto.

2. Gerenciamento de Rotas

O Node nativo não sabe distinguir automaticamente entre um GET /usuarios e um POST /usuarios. Você precisa criar essa lógica do zero. O Fastify oferece um sistema de roteamento intuitivo e extremamente rápido.

3. Performance e Validação

O Fastify é famoso por ser um dos frameworks mais rápidos do ecossistema.

- **JSON Schema:** Ele usa esquemas para validar os dados que chegam. Se o usuário enviar um texto onde deveria ser um número, o Fastify barra a requisição antes mesmo de processá-la, poupando recursos.
- **Serialização:** Ele transforma objetos em texto (JSON) de forma muito mais veloz que o padrão JSON.stringify().

4. Ecossistema de Plugins

Precisa conectar ao banco de dados? Autenticação JWT? CORS? No nativo, você teria que configurar cada detalhe. No Fastify, você simplesmente instala um plugin oficial (@fastify/postgres, @fastify/jwt) e ele se integra ao ciclo de vida da aplicação de forma segura.

Quando usar cada um?

- **Node Nativo:** Use apenas para aprendizado (entender como o protocolo HTTP funciona por baixo dos panos) ou se estiver criando uma ferramenta extremamente minimalista onde cada byte de memória conta.
- **Fastify:** Use para quase tudo na vida real. É focado em **performance extrema** e baixo overhead (sobrecarga), sendo ideal para microsserviços modernos.

Instalando o Fastify

Para instalar a dependência do Framework Fastify digite no terminal o comando a seguir;

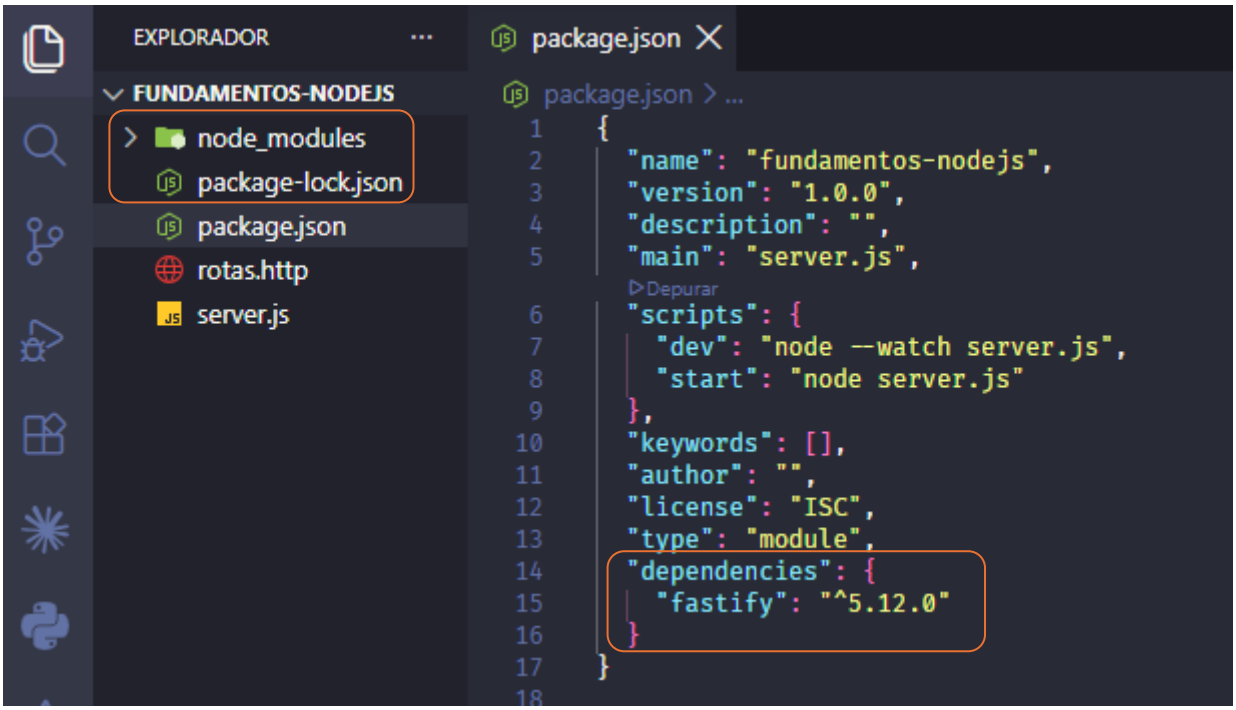
Terminal

```
npm install fastify
```

Ao instalar o Fastify, repare que foi criado a pasta **node_modules**, esta pasta contém todas as dependências necessárias para o Fastify e qualquer outra biblioteca javascript funcionar.

Repare também que no arquivo package.json agora possui uma chave chamada "***dependencies***" com o fastify instalado e especificando a versão que foi instalada.

Dica Extra: Você notará que, ao instalar um pacote, também surge o arquivo package-lock.json. Ele serve para registrar as versões exatas de cada sub-dependência, garantindo que o projeto rode da mesma forma em qualquer máquina.



Trocando o servidor node nativo para o Fastify

Vamos lá, para que seja perceptível o quanto o fastify é mais fácil de ser implementado e usado, vamos apagar o código anterior e aplicar o código a seguir que usa o fastify. Você vai perceber que a aplicação irá fazer a mesma coisa, mas de maneira muito mais fácil e eficiente.

server.js

```
import { fastify } from 'fastify';

const server = fastify();

server.get('/', (req, res) => {
  return 'Home - Cadastro de usuarios';
});

server.get('/users', (req, res) => {
  return 'Listagem de usuarios';
});
server.post('/users', (req, res) => {
  return 'Cadastro de usuarios';
});


server.listen({port: 3000}, (err, address) => {
  if (err) {
    console.error(err);
  }
  console.log(`Servidor rodando em ${address}`);
});
```

Pare a execução do servidor http que ainda está em execução, acessando o terminal e pressionando o atalho no teclado ***CTRL + C*** *que fara o servidor parar*.

Execute novamente o servidor digitando no terminal o comando a seguir, para vê-lo em funcionamento novamente, mas desta vez usando o Fastify.

Terminal

```
npm run dev
```

Acesse as rotas novamente no arquivo rotas.http, observe que, o fastify já possui os métodos get e post para combinar com as urls e então, criar a rota desejada, evitando ter que criar estruturas condicionais usando IF.

Aula 4 – Parâmetros de rotas

Nesta aula, veremos como definir e utilizar parâmetros dinâmicos em rotas, permitindo a captura de valores diretamente pela URL.

Route Params (Parâmetros de rotas)

São valores **dinâmicos** dentro do caminho da URL, indicados com "**:**" (dois pontos)

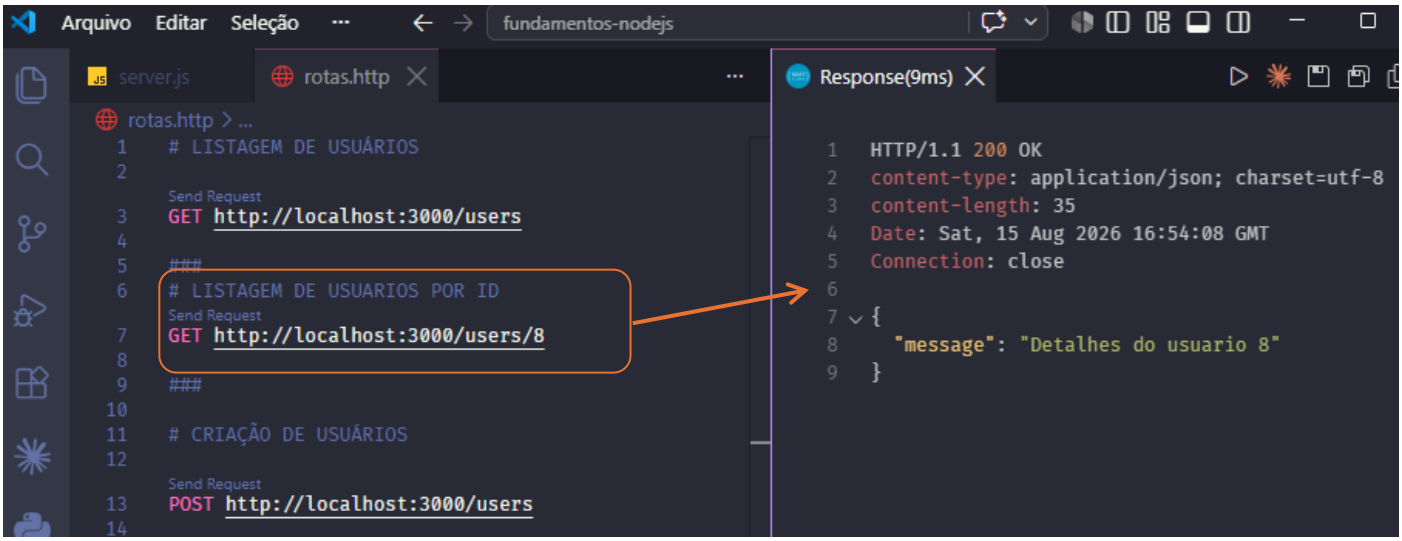
```
server.get('/users/:id', async (req, res)⇒{
    const { id } = req.params;
    return { 'message': `Detalhes do usuario ${id}` };
});
```

Adicione o código a seguir no arquivo rotas.http, notamos que;

```
# LISTAGEM DE USUÁRIO POR ID
Send Request
GET http://localhost:3000/users/8
```

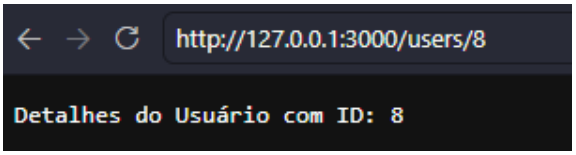
Se o usuário acessar a rota:

GET /users/8



Então **req.params.id** será igual a **"8"**.

Visualização no navegador



Podemos ter mais de um parâmetro como mostra o exemplo a seguir:

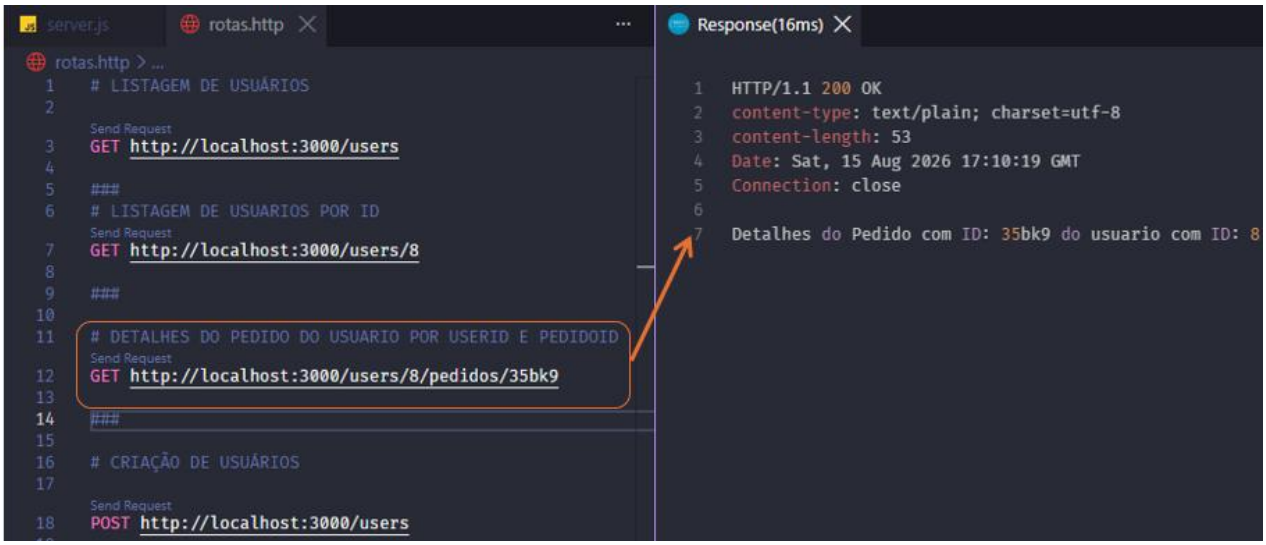
```
server.get('/users/:userId/pedidos/:pedidoId', (req, res) ⇒ {
  return `Detalhes do Pedido com ID: ${req.params.pedidoId} do Usuário com ID: ${req.params.userId}`;
});
```

```
GET /users/8/pedidos/35bk9
// usuarioId = "8", pedidoId = "35bk9"
```

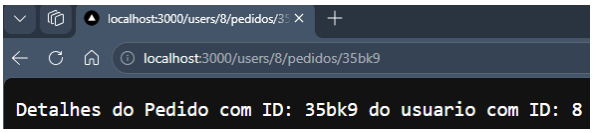
Adicione o código a seguir no arquivo rotas.http

```
# DETALHES DO PEDIDO DO USUARIO POR USERID E PEDIDOID
Send Request
GET http://localhost:3000/users/8/pedidos/35bk9
```

O resultado a direita;



Visualização no navegador



Query Params (parâmetros de consulta)

São aqueles parâmetros depois do **"?"** na URL, usados geralmente para **filtros, busca, paginação e ordenação**.

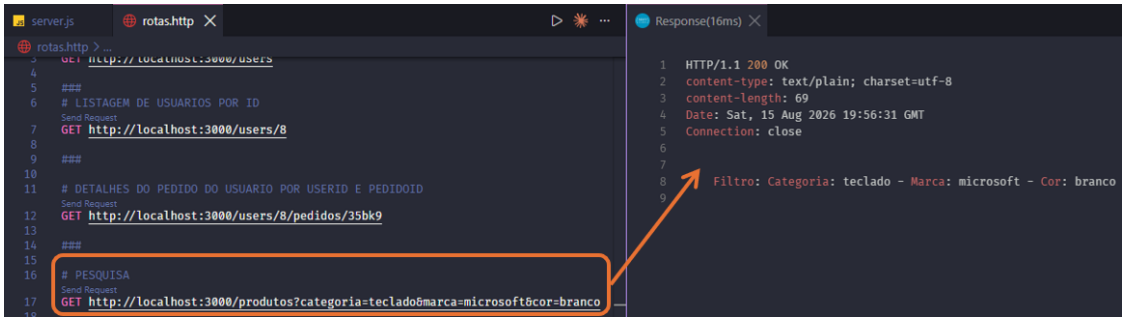
```
server.get('/produtos',(req, res)=>{
    const { categoria, marca, cor } = req.query;
    return `
    Filtro: Categoria: ${categoria} - Marca: ${marca} - Cor: ${cor}
    `
});
```

Digite e acesse no arquivo de rotas como é mostrado a seguir:

GET http://localhost:3000/produtos?categoria=teclado&marca=microsoft&cor=branco

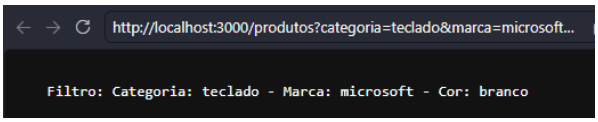
```
# PESQUISA
Send Request
GET http://localhost:3000/produtos?categoria=teclado&marca=microsoft&cor=branco

###
```

Será mostrado assim:



Nota
Os exemplos listados são todos pela rota get, portanto, podem ser testados pelo navegador digitando na barra de endereços url ou pelo arquivo de rota

No navegador será mostrado assim, bastando digitar na barra de endereço a url:



Nota: cada parâmetro da url é separado um do outro pelo caracter "&".
http://localhost:3000/produtos**?categoria=teclado&marca=microsoft&cor=branco**

Diferença prática entre Route Params e Query Params

	Route Params	Query Params
Sintaxe	/produtos/:id	/produtos?id=1
Uso comum	Identificar **um recurso específico**	Filtrar, buscar, paginar
Obrigatório?	Geralmente sim	Geralmente opcional
Exemplo	/usuarios/5	/usuarios?ativo=true&pagina=2

Body (corpo da requisição)

Usado em POST, PUT e PATCH para enviar dados **maiores/estruturados**, geralmente em JSON.

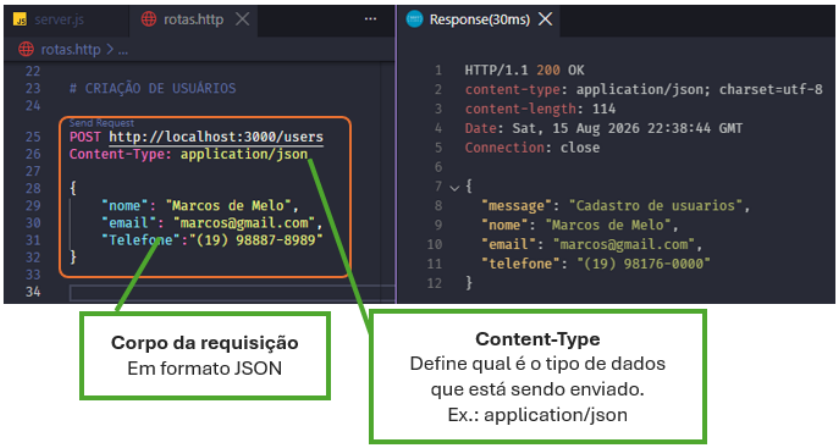
Já compreendemos que a rota POST é usada para criar / inserir um novo recurso (dado). Agora vamos compreender como mandar os dados por ele. Para enviar os vários dados de um registro, enviamos estas informações em um corpo da requisição, dentro da rota definimos o que será enviado na requisição criando variáveis que serão recuperadas pela requisição no método **.body**.

Vamos editar a rota post para ter um corpo com dados a serem enviados, veja no exemplo como definir o corpo da ***query params***;

```
server.post('/users', async (req, res)=>{
    const {nome, email, telefone } = req.body;

    return { 'message': 'Cadastro de usuarios',
            nome,
            email,
            telefone
    };
});
```

Quando acessarmos a rota acima sabemos que devemos enviar os dados ***{ nome, email, telefone}***

No arquivo de rotas ***POST /users*** vamos acrescentar o corpo da requisição com os dados que queremos adicionar.



Observações: Ainda não estamos adicionando nenhum registro em um banco de dados, no exemplo acima, estamos demonstrando como enviar os dados pela rota e como recuperá-los, para então, depois, implementar no código a parte que insere no banco de dados.

Resumo visual da rota completa

```
GET     /produtos/:id?ordenar=preco
 |          |      |        |
método   caminho  route    query
         fixo     param    param
```

```javascript
app.get('/produtos/:id', (req, res) => {
  const { id } = req.params;      // parâmetro da rota
  const { ordenar } = req.query;  // parâmetro de consulta
  // req.body aqui não teria uso, pois GET não costuma ter corpo
});
```

Resumo para guardar

Conceito	O que é	Onde aparece
Método HTTP	Intenção da ação	GET, POST, PUT, PATCH, DELETE
Route Params	Identifica um recurso específico	/produtos/:id
Query Params	Filtros e opções	/produtos?categoria=x
Body	Dados enviados pelo cliente	req.body
Router	Organização de rotas	express.Router()

Quiz de Perguntas

Questionário — Aula 4: Parâmetros de Rotas

Instruções: Leia cada questão com atenção e marque a alternativa correta (apenas uma por questão).

Questão 01.

No Node.js com Fastify, o que são Route Params (parâmetros de rota)?

a) Parâmetros enviados no corpo da requisição em formato JSON.

b) Parâmetros opcionais que aparecem após o símbolo "?" na URL.

c) Valores dinâmicos inseridos diretamente no caminho da URL, indicados com ":" (dois pontos).

d) Cabeçalhos HTTP adicionados pelo cliente para identificar o recurso.

e) Parâmetros utilizados exclusivamente em requisições do tipo POST.

Questão 02.

Qual das URLs abaixo representa corretamente o uso de um Route Param para buscar o usuário de ID 8?

a) GET /users?id=8

b) GET /users&id=8

c) GET /users#8

d) GET /users/8

e) GET /users[8]

Questão 03.

Observe a rota a seguir:

GET /users/8/pedidos/35bk9. Ao acessar essa rota, qual será o valor da variável pedidoId?

a) "users"

b) "8"

c) "pedidos"

d) "35bk9"

e) undefined, pois a rota possui dois parâmetros e isso gera conflito.

Questão 04.

Para que servem os Query Params em uma rota HTTP?

a) Para identificar um recurso único e obrigatório na URL, como um ID.

b) Para enviar senhas e tokens de forma segura no cabeçalho da requisição.

c) Para substituir o corpo (body) em requisições do tipo POST.

d) Para filtrar, buscar, paginar e ordenar resultados de forma opcional.

e) Para definir o método HTTP que será utilizado na requisição.

Questão 05.

Dada a URL: http://localhost:3000/produtos?categoria=teclado&marca=microsoft&cor=branco, quantos Query Params ela contém e qual caractere os separa entre si?

a) Dois parâmetros, separados pelo caractere "?".

b) Três parâmetros, separados pelo caractere "&".

c) Três parâmetros, separados pelo caractere "/".

d) Quatro parâmetros, separados pelo caractere "=".

e) Dois parâmetros, separados pelo caractere ":".

Questão 06.

Qual é a principal diferença de uso entre Route Params e Query Params?

a) Route Params são usados apenas em requisições DELETE, e Query Params em requisições GET.

b) Route Params identificam um recurso específico (geralmente obrigatório), enquanto Query Params são usados para filtros e opções (geralmente opcionais).

c) Route Params aparecem após o "?" na URL, enquanto Query Params aparecem antes. d) Route Params são enviados no corpo da requisição, enquanto Query Params são enviados no cabeçalho.

e) Não há diferença prática: ambos têm o mesmo propósito e sintaxe.

Questão 07.

No código de uma rota, por qual propriedade do objeto de requisição acessamos os dados enviados no Route Param?

a) req.query

b) req.body

c) req.headers

d) req.params

e) req.url

Questão 08.

O Body (corpo da requisição) é utilizado para enviar dados ao servidor. Em quais métodos HTTP o seu uso é mais comum?

a) GET e DELETE.

b) GET e OPTIONS.

c) POST, PUT e PATCH.

d) DELETE e HEAD.

e) GET, POST e DELETE.

Questão 09.

Analise a tabela de resumo vista na Aula 4 e assinale a alternativa que associa corretamente o conceito ao seu propósito:

a) Query Params → Identifica um recurso específico na URL, como /produtos/:id.

b) Route Params → Utilizado para filtros e opções, como /produtos?categoria=x.

c) Body → Dados enviados pelo cliente, acessados via req.body.

d) Método HTTP → Organiza e agrupa rotas da aplicação.

e) Router → Define a intenção da ação (GET, POST, PUT, DELETE).

Questão 10.

Considere a seguinte situação: um desenvolvedor precisa criar uma rota que liste todos os usuários ativos e ainda permita paginar os resultados. Qual abordagem é a mais adequada segundo os conceitos da Aula 4?

a) GET /usuarios/:ativo/:pagina — usando dois Route Params.

b) POST /usuarios com { ativo: true, pagina: 2 } no body.

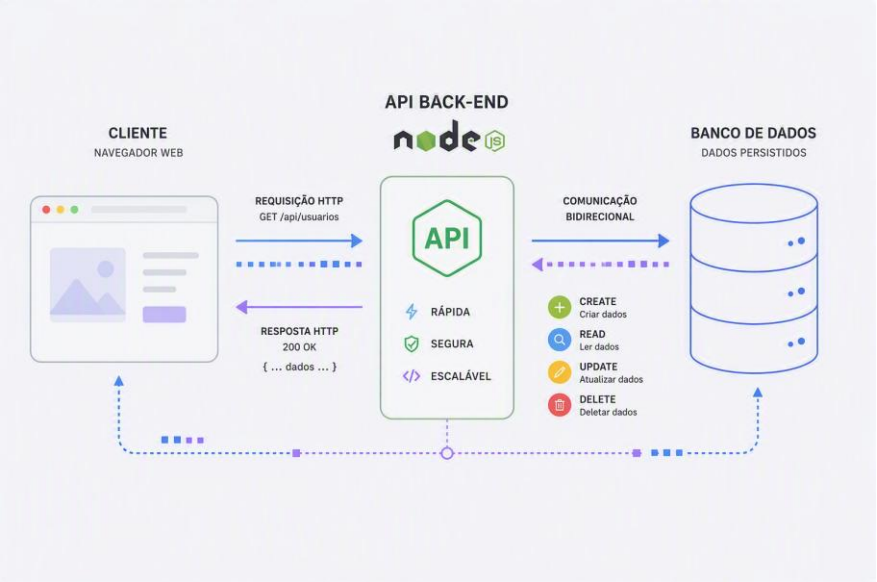
c) GET /usuarios?ativo=true&pagina=2 — usando Query Params.

d) DELETE /usuarios/:ativo — usando Route Param com o status do usuário.

e) GET /usuarios#ativo=true — usando o fragmento de URL.

Aula 5 – CRUD – Persistência de Dados

Chegou o momento de aprender um dos conceitos mais importantes do desenvolvimento back-end: a persistência de dados. Nesta aula, veremos como conectar nossa aplicação ao banco de dados e manipular as informações de forma segura e eficiente.



O banco de dados MySQL

Até agora nossas rotas apenas simulavam o envio e o recebimento de dados. Para que os dados sobrevivam ao desligamento do servidor, precisamos de um banco de dados.

O MySQL é um Sistema Gerenciador de Banco de Dados (SGBD) relacional, ou seja, ele guarda os dados organizados em tabelas, formadas por linhas (registros) e colunas (campos), muito parecido com uma planilha do Excel, porém muito mais robusto, rápido e preparado para lidar com milhares de acessos simultâneos.

É um dos bancos de dados mais usados do mundo, gratuito, open source e amplamente utilizado em aplicações Node.js.

Pré-requisito:

Para acompanhar esta aula você precisa ter o MySQL instalado e em execução na sua máquina (pode ser através do MySQL Server + MySQL Workbench, ou de um pacote como XAMPP/Laragon, que já instala o MySQL junto). Certifique-se também de ter criado um usuário e uma senha de acesso ao banco.

Arquivo ENV

O arquivo **.env** é um arquivo de texto simples usado em aplicações Node.js para armazenar **variáveis de ambiente**. Ele permite separar as configurações e dados sensíveis da aplicação do código-fonte propriamente dito.

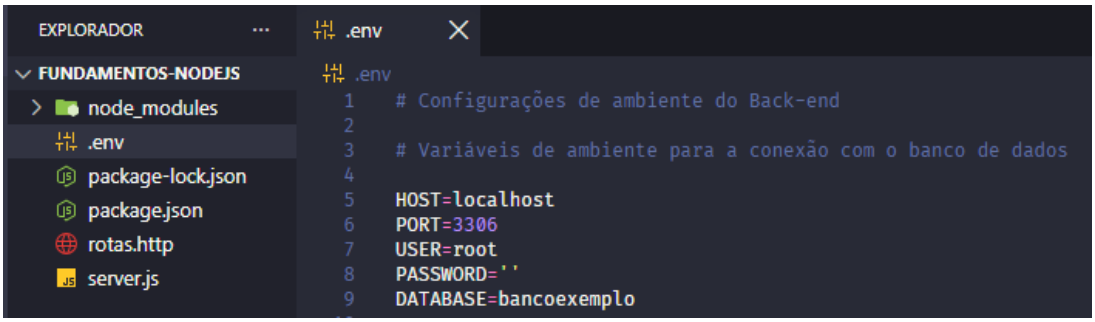
Para que serve?

- **Segurança:** Protege informações sensíveis, como senhas de banco de dados, chaves de API e segredos de tokens, evitando que sejam expostos no código-fonte.

- **Configuração por ambiente:** Facilita o uso de diferentes configurações para desenvolvimento, teste e produção sem precisar alterar o código.
- **Centralização:** Armazena todas as configurações em um único local no formato CHAVE=valor.

Como usar no Node.js

Crie na pasta **raiz** do projeto o arquivo "**.env**" e digite o código a seguir nele.



Para acessar esses valores, a aplicação lê o arquivo e popula o objeto global process.env.

A forma que iremos acessar as informações deste arquivo é através da **Biblioteca dotenv:** É a solução mais comum para versões anteriores do Node.js.

Digite no terminal o comando a seguir para instalar a biblioteca dotenv.

Terminal

```
npm install dotenv -D
```

Dica importante: Nunca envie seu arquivo **.env** para o controle de versão (como o Git). Adicione-o ao seu arquivo **.gitignore** para garantir que seus segredos não sejam compartilhados publicamente.

Conectando o Node.js ao MySQL

Para que o nosso back-end em Fastify consiga conversar com o banco de dados MySQL, precisamos instalar um driver, uma biblioteca que sabe como abrir uma conexão, enviar comandos SQL e devolver o resultado para dentro do JavaScript. Vamos usar o pacote mysql2, um dos mais usados do ecossistema Node.js.

Terminal

```
npm install mysql2
```

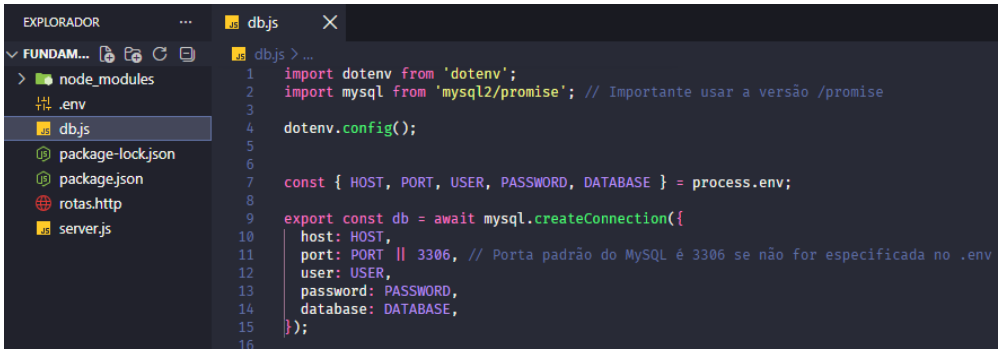
Criando o banco de dados e a tabela

Abra o MySQL Workbench (ou o cliente de sua preferência) e crie um banco de dados novo para o nosso projeto, chamado ***bancoexemplo***, e dentro dele a tabela **users**, que vai guardar os usuários da nossa aplicação.

```
CREATE DATABASE bancoexemplo;
USE bancoexemplo;
CREATE TABLE users (
  id VARCHAR(255) PRIMARY KEY,
  nome VARCHAR(100) NOT NULL,
  email VARCHAR(100) NOT NULL,
  telefone VARCHAR(20)
);
```

Criando o arquivo de conexão

Com a dependência instalada, vamos criar um arquivo separado só para cuidar da conexão com o banco, chamado ***db.js***, na raiz do projeto. Separar esse código em um arquivo próprio evita repetir a configuração da conexão em cada rota.



Criando o arquivo de persistência de dados

Crie o arquivo "**database-mysql.js**", vamos codificar uma classe com o nome DatabaseMYSQL e dentro vamos definir os métodos para persistir os dados no banco de dados.

database-mysql.js

```
import { randomUUID } from "node:crypto"
import { db } from './db.js';

export class DatabaseMYSQL {


    // Cria novo usuario
    async createUser(user){
        const userId = randomUUID();
        const { nome, email, telefone } = user;

        // No mysql2, passamos os valores em um array com segundo argumento
        await db.execute(
            'INSERT INTO users (id, nome, email, telefone) values (?, ?, ?, ?)',
            [userId, nome, email, telefone]
        )
    }
    // Lista os usuarios cadastrados
    async readUser(search){
        let users;
        if(search){
            [users] = await db.execute(
                'SELECT * FROM users WHERE nome LIKE ?',
                [`%${search}%`]
            )
        }else{
            [users] = await db.execute('SELECT * FROM users');
        }
        return users;
    }
    // Lista um usuario especifico pelo id
    async readUserDetails(id){
        let users;

            [users] = await db.execute('SELECT * FROM users WHERE id = ?', [id]);

        return users;
    }

    // Atualiza um usuario especifico
    async updateUser(id, user){
         const { nome, email, telefone } = user;
         await db.execute(
            'UPDATE users SET nome = ?, email = ?, telefone = ? WHERE id = ?',
            [nome, email, telefone, id]
         )
    }
    // Excluir um usuario especifico
    async deleteUser(id){
        await db.execute(
            'DELETE FROM users WHERE id = ?',[id]
        );
    }

}
```

Modificando as rotas

Vamos Editar o arquivo server.js para implementar nas rotas a persistência de dados. O código completo está apresentado em partes para ser explicado.

Na linha 2 importamos o arquivo do banco de dados DatabaseMYSQL

Na linha 6 instanciamos o objeto **database** com a class DatabaseMYSQL.

server.js

server.js > server.listen() callback

```
1  import { fastify } from 'fastify'
2  import { DatabaseMYSQL } from './database-mysql.js';
3
4  const server = fastify();
5
6  const database = new DatabaseMYSQL();
```

Nas linhas 8 a 10 continua a rota principal.

```
7
8  server.get('/', async (req, res)=>{
9      return { 'message': 'Home - Cadastro de Usuarios' };
10 });
```

Nas linhas 12 a 16, definição da rota Read que lista todos os usuários ou filtra por pesquisa somente alguns.

Na linha 13, recuperamos a query params search e na linha 14 atribuimos na constante users o conteúdo da execução do metodo readUser(search) no banco de dados.

```
11 // READ - Lista todos os usuários
12 server.get('/users', async (req)=>{
13     const search = req.query.search;
14     const users = await database.readUser(search);
15     return users;
16 });
```

Nas linhas 19 a 23, definição da rota Read que retorna somente as informações de um usuário especifico.

```
17
18 // READ - Lista detalhes do usuario pelo id
19 server.get('/users/:id', async (req, res)=>{
20     const id   = req.params.id;
21     const user = await database.readUserDetails(id)
22     return user;
23 });
24
```

Nas linhas 26 a 36, definição da rota CREATE – cadastro de novo usuário.

Na linha 28, desestruturamos os dados recebidos no corpo da rata.

Nas linhas 30 a 34 executamos o metodo createUser que insere os dados no banco de dados

```
25  // CREATE - Cadastra um novo usuário
26  server.post('/users', async (req, res)=>{
27
28      const {nome, email, telefone } = req.body;
29
30      await database.createUser({
31          nome,
32          email,
33          telefone
34      });
35      return res.status(201).send();
36  });
```

Nas linhas 38 a 47, definimos a rota Update

```
37  // Update - Atualiza um usuario informando o id
38  server.put('/users/:id', async (req, res)=>{
39      const id = req.params.id;
40      const {nome, email, telefone } = req.body;
41
42      await database.updateUser(id, {
43          nome,
44          email,
45          telefone
46      })
47  });
```

Nas linhas 49 a 53, definição da rota Delete

```
48  // Delete - Exclui um suario informando o id
49  server.delete('/users/:id',async (req, res)=>{
50      const id = req.params.id;
51      await database.deleteUser(id);
52      return res.status(204).send();
53  })
54
```

Nas linhas 55 a 62, execução do servidor http

```
55  server.listen({ port: 3000},(err, address) => {
56      if(err){
57          console.error(err);
58          process.exit(1);
59      }
60      console.log(`Servidor rodando em ${address}`);
61  }
62  );
```

Editando o arquivo de Rotas

Edite o arquivo ***rotas.http*** para testar a persistência de dados no banco de dados.

rotas.http

```
# LISTAGEM DE USUÁRIOS TOTAL OU FILTRADO

Send Request
GET http://localhost:3000/users

###
# LISTAGEM DETALHES DO USUARIOS POR ID
Send Request
GET http://localhost:3000/users/f196705e-bb14-4e3f-b044-40ebc48036be

###


# CRIAÇÃO DE USUÁRIOS

Send Request
POST http://localhost:3000/users
Content-Type: application/json

{
    "nome": "Juliana Melo",
    "email": "ju@gmail.com",
    "telefone":"(19) 98887-8989"
}

###

# Atualização de usuário
Send Request
PUT http://localhost:3000/users/9113c34c-9d1d-471f-8f32-0a9d4af2129e
Content-Type: application/json

{
    "nome": "Juliana Almeida Melo",
    "email": "juliana@gmail.com",
    "telefone":"(19) 98887-3333"
}

####
# Deletar usuário
Send Request
DELETE http://localhost:3000/users/9113c34c-9d1d-471f-8f32-0a9d4af2129e

```

Aula 6 – Códigos de Erros

Vamos entender uma parte fundamental do desenvolvimento back-end: **os status codes** (códigos de status) das respostas HTTP.

Oque são Status Codes HTTP?

Todo request HTTP feito ao nosso servidor recebe uma **resposta**, e essa resposta sempre carrega um código numérico de 3 dígitos que informa ao cliente (navegador, app mobile, outra API) **o que aconteceu** com aquela requisição.

Pensem assim: é como se o servidor estivesse respondendo "deu certo", "você errou algo" ou "eu quebrei aqui" — e cada situação tem um número padronizado.

Esses códigos são divididos em **5 categorias**, identificadas pelo primeiro dígito:

Faixa	Categoria	Significado
1xx	Informational	Requisição recebida, processando
2xx	Success	Deu tudo certo
3xx	Redirection	Precisa de mais uma ação (redirecionamento)
4xx	Client Error	O cliente (quem fez o request) errou algo
5xx	Server Error	O servidor quebrou/falhou

Vamos focar nas três categorias que vocês mais vão usar no dia a dia: **2xx, 4xx e 5xx**.

2xx — Sucesso

200 OK

Uso mais comum. A requisição foi bem-sucedida.

```javascript
app.get('/produtos', (req, res) => {
  const produtos = [{ id: 1, nome: 'Teclado' }];
  res.status(200).json(produtos);
});
```

201 Created

Usado quando um novo recurso foi **criado** com sucesso (ex: cadastro de usuário).

```javascript
app.post('/usuarios', (req, res) => {
  const novoUsuario = { id: 10, nome: req.body.nome };
  // salvar no banco...
  res.status(201).json(novoUsuario);
});
```

204 No Content

Sucesso, mas **sem corpo de resposta**. Muito usado em DELETE.

```javascript
app.delete('/usuarios/:id', (req, res) => {
  // deletar usuário do banco...
  res.status(204).send();
});
```

4xx — Erro do Cliente

400 Bad Request

O cliente enviou dados **inválidos ou incompletos**.

```javascript
app.post('/usuarios', (req, res) => {
  if (!req.body.nome) {
    return res.status(400).json({ erro: 'O campo "nome" é obrigatório' });
  }
  // continua o cadastro...
});
```

401 Unauthorized

O cliente **não está autenticado** (não enviou token, ou token inválido).

```javascript
app.get('/perfil', (req, res) => {
  const token = req.headers.authorization;
  if (!token) {
    return res.status(401).json({ erro: 'Token não fornecido' });
  }
  // continua...
});
```

403 Forbidden

O cliente **está autenticado, mas não tem permissão** para aquela ação.

```javascript
app.delete('/admin/usuarios/:id', (req, res) => {
  if (req.usuario.tipo !== 'admin') {
    return res.status(403).json({ erro: 'Acesso negado. Somente administradores.' });
  }
  // continua...
});
```

404 Not Found

O recurso solicitado **não existe**.

```javascript
app.get('/produtos/:id', (req, res) => {
  const produto = produtos.find(p => p.id === Number(req.params.id));
  if (!produto) {
    return res.status(404).json({ erro: 'Produto não encontrado' });
  }
  res.status(200).json(produto);
});
```

409 Conflict

Conflito com o estado atual do recurso (ex: e-mail já cadastrado).

```javascript
app.post('/usuarios', (req, res) => {
  const existe = usuarios.find(u => u.email === req.body.email);
  if (existe) {
    return res.status(409).json({ erro: 'E-mail já cadastrado' });
  }
  // continua...
});
```

5xx — Erro do Servidor

500 Internal Server Error

Algo deu errado **no nosso código**, geralmente uma exceção não tratada.

```javascript
app.get('/produtos', async (req, res) => {
  try {
    const produtos = await buscarProdutosNoBanco();
    res.status(200).json(produtos);
  } catch (erro) {
    console.error(erro);
    res.status(500).json({ erro: 'Erro interno no servidor' });
  }
});
```

💡 Dica prática

Um erro comum de quem está começando: **esquecer de definir o status code**. Se vocês não especificarem, o Fastify por padrão envia 200, mesmo quando é um erro! Por isso é essencial ser explícito:

```javascript
// ❌ Ruim: mesmo dando erro, retorna 200 por padrão
res.json({ erro: 'Não encontrado' });

// ✅ Correto: status code condizente com a situação
res.status(404).json({ erro: 'Não encontrado' });
```

Resumo para guardar

Código	Quando usar
200	Sucesso geral (GET, PUT)
201	Recurso criado (POST)
204	Sucesso sem retorno (DELETE)
400	Dados inválidos enviados
401	Não autenticado
403	Autenticado, mas sem permissão
404	Recurso não existe
409	Conflito de dados
500	Erro interno do servidor