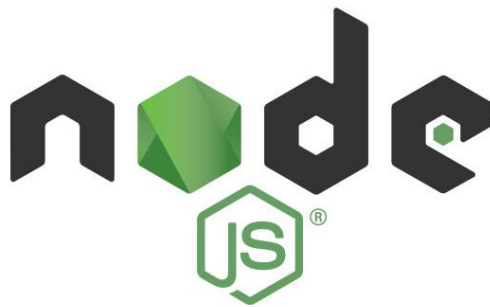


Tutorial



Professor

Marcos de Melo

Sumário

O que é o Node.js?	3
Analogia	3
Características principais:	3
Como o NODE.JS funciona?	3
Para que serve o Node.js?	4
Por que aprender Node.js hoje?	4
Por que usar Node.js e não outra tecnologia?	4
Módulos no Node.JS	5
NPM – Node Package Manager	5
Instalação do Node.JS	5
Passo A: Instalação	5
Passo B: Criando seu primeiro servidor http no node	8
Passo C: Executando	9
O Ecossistema NPM	9
Usando o NPM	10
Definindo o sistema de importação de módulos	12
Criando o servidor utilizando Framework	14
Por que usar Fastify em vez do Node Nativo?	14
Quando usar cada um?	15
Instalando o Fastify	15
Trocando o servidor node nativo para o Fastify	16
Criando rotas	16
Entendendo a rota Get	16

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:

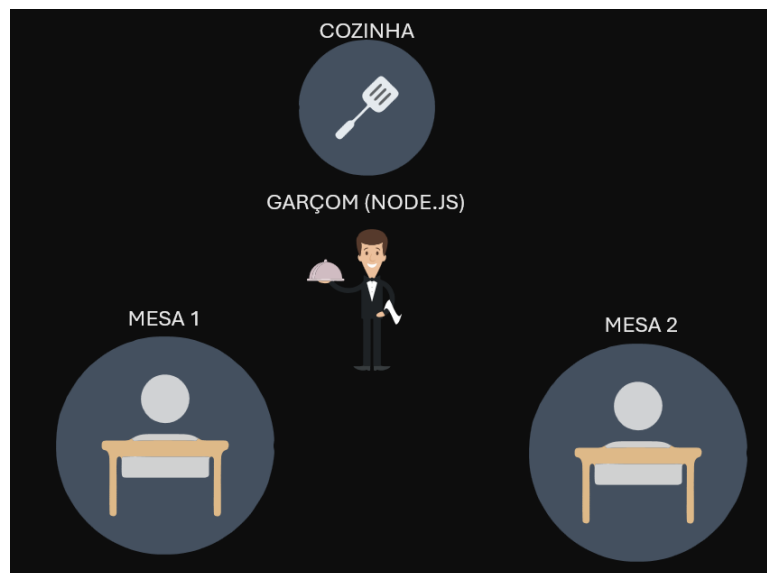
Características principais:

- **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 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

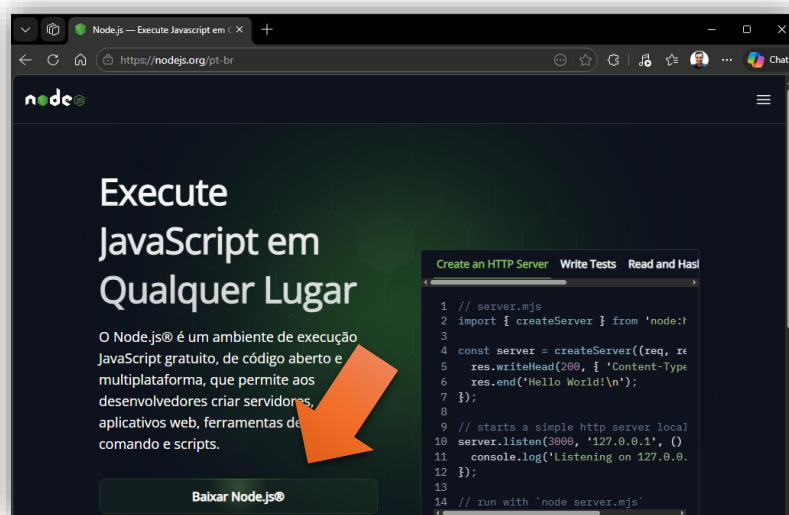
```
npm init // cria o package.json  
npm -g install nodemon // instala um pacote globalmente  
npm install express // instala um pacote localmente
```

Instalação do Node.JS

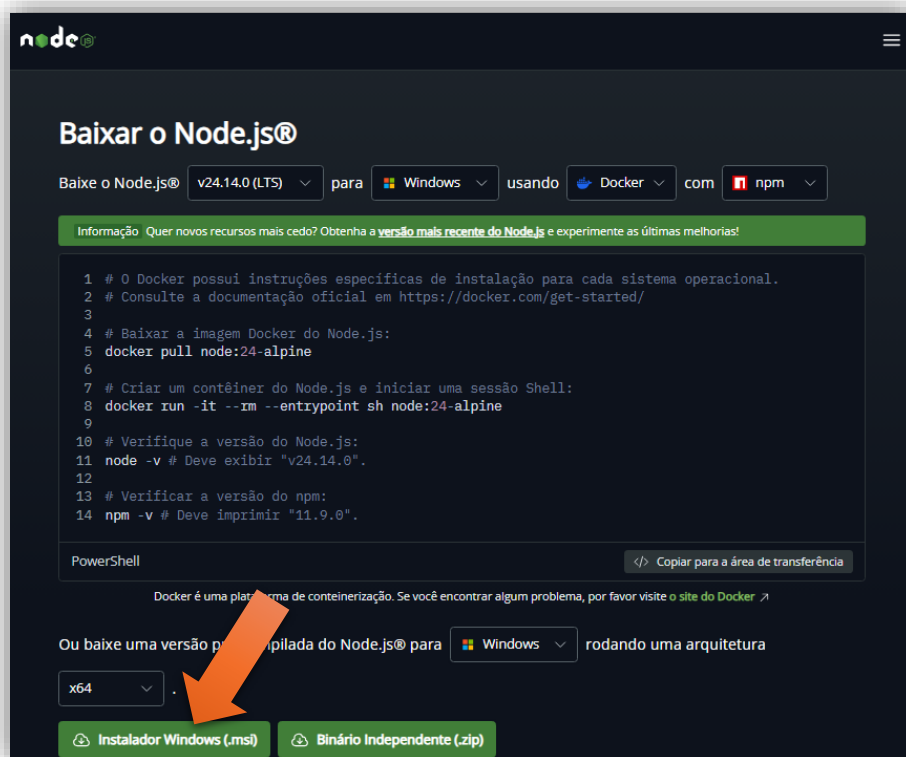
Se você quer ver o Node em ação agora, siga este roteiro de instalação:

Passo A: Instalação

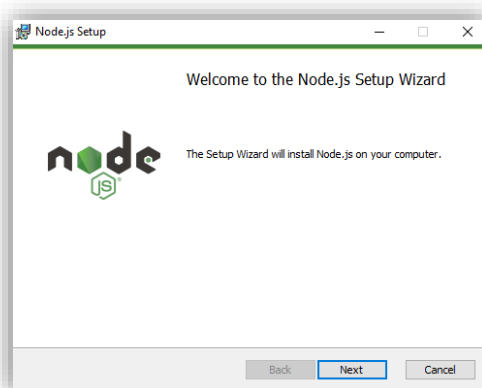
1. Vá ao site oficial nodejs.org.

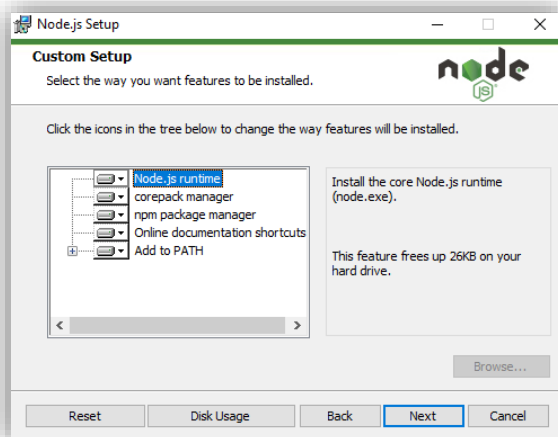
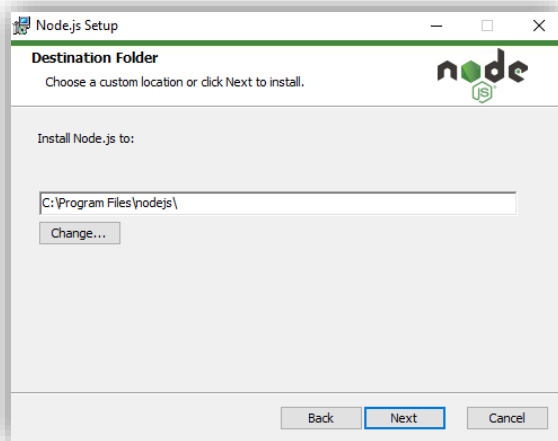
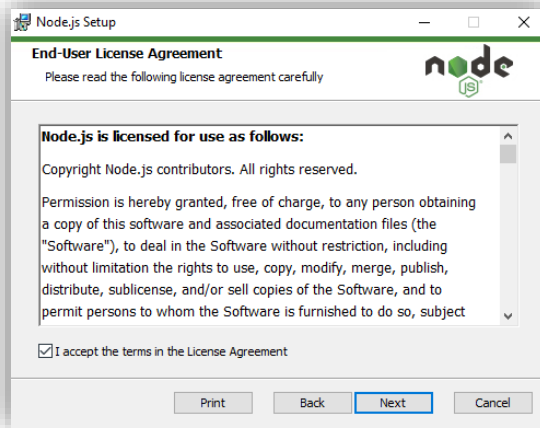


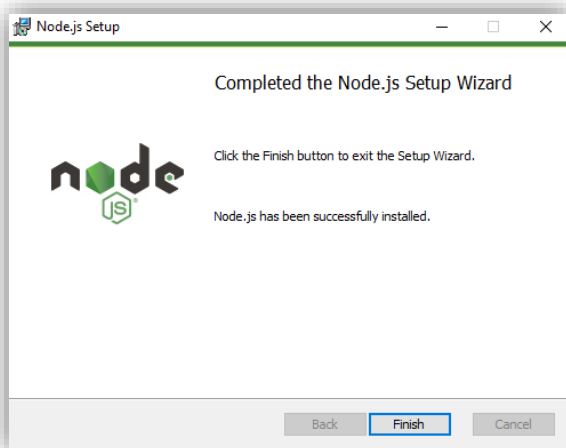
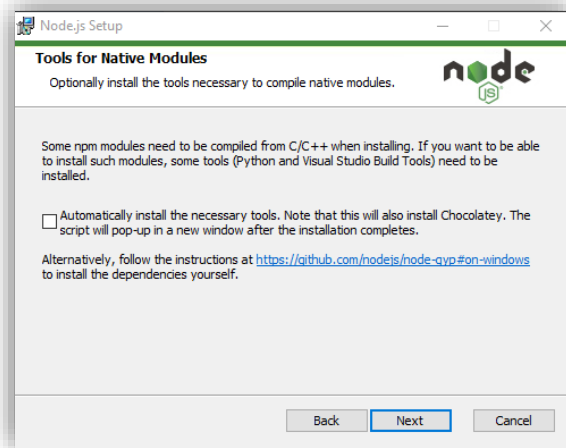
2. Baixe a versão **LTS** (Long Term Support), que é a mais estável para ensino e produção. Se estiver no sistema operacional Windows, baixar a versão pré-compilada lts (.msi) é a maneira mais fácil de instalar o node.



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.



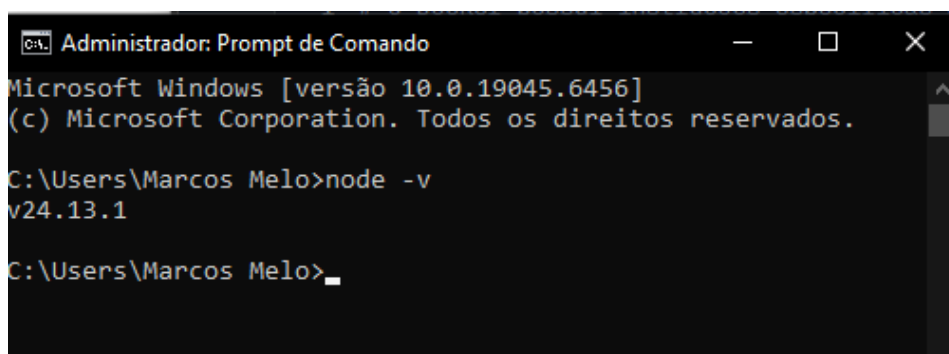




4. Após instalar, abra o seu terminal (CMD ou PowerShell) e digite:

```
node -v.
```

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



Passo B: Criando seu primeiro servidor http no node

Não é necessário um servidor externo (como Apache). O Node **é** o servidor.

1. Crie uma pasta para o projeto, ex: **node-do-zero**.
2. Abra esta pasta no VS CODE (Editor de Códigos)
3. Crie um arquivo chamado **server.js**.

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

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

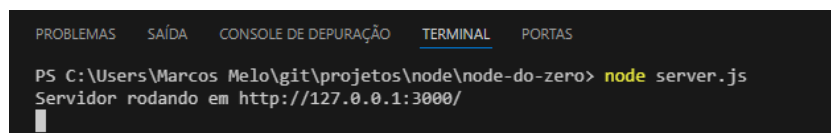
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Ola! Este e um servidor rodando em Node.js');
});

server.listen(3000, '127.0.0.1', () => {
  console.log('Servidor rodando em http://127.0.0.1:3000/');
});
```

Passo C: Executando

No terminal, dentro da pasta do arquivo, digite:

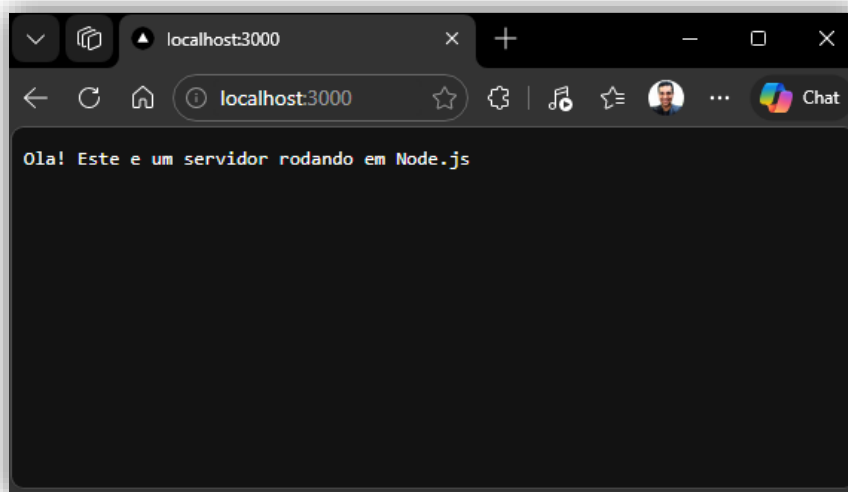
```
node server.js
```



```
PROBLEMAS  SAÍDA  CONSOLE DE DEPURACÃO  TERMINAL  PORTAS

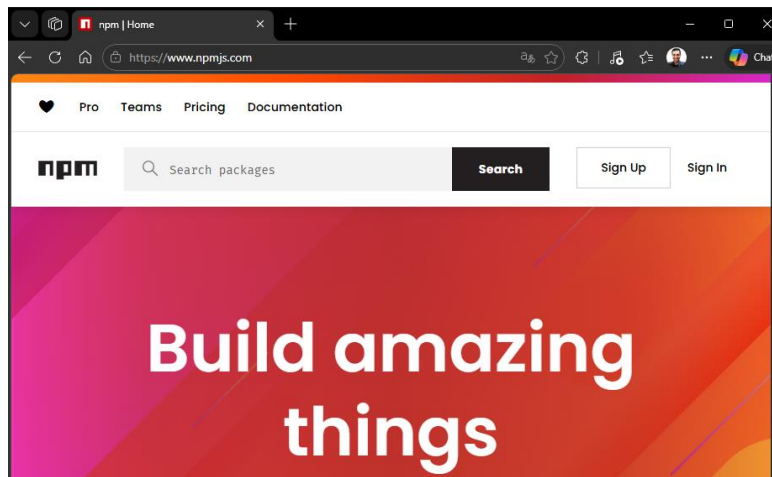
PS C:\Users\Marcos Melo\git\projetos\node\node-do-zero> node server.js
Servidor rodando em http://127.0.0.1:3000/
```

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:

```
npm -v
```

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

```
Administrador: Prompt de Comando
C:\Users\Marcos Melo\git\projetos\node>npm -v
11.9.0
C:\Users\Marcos Melo\git\projetos\node>
```

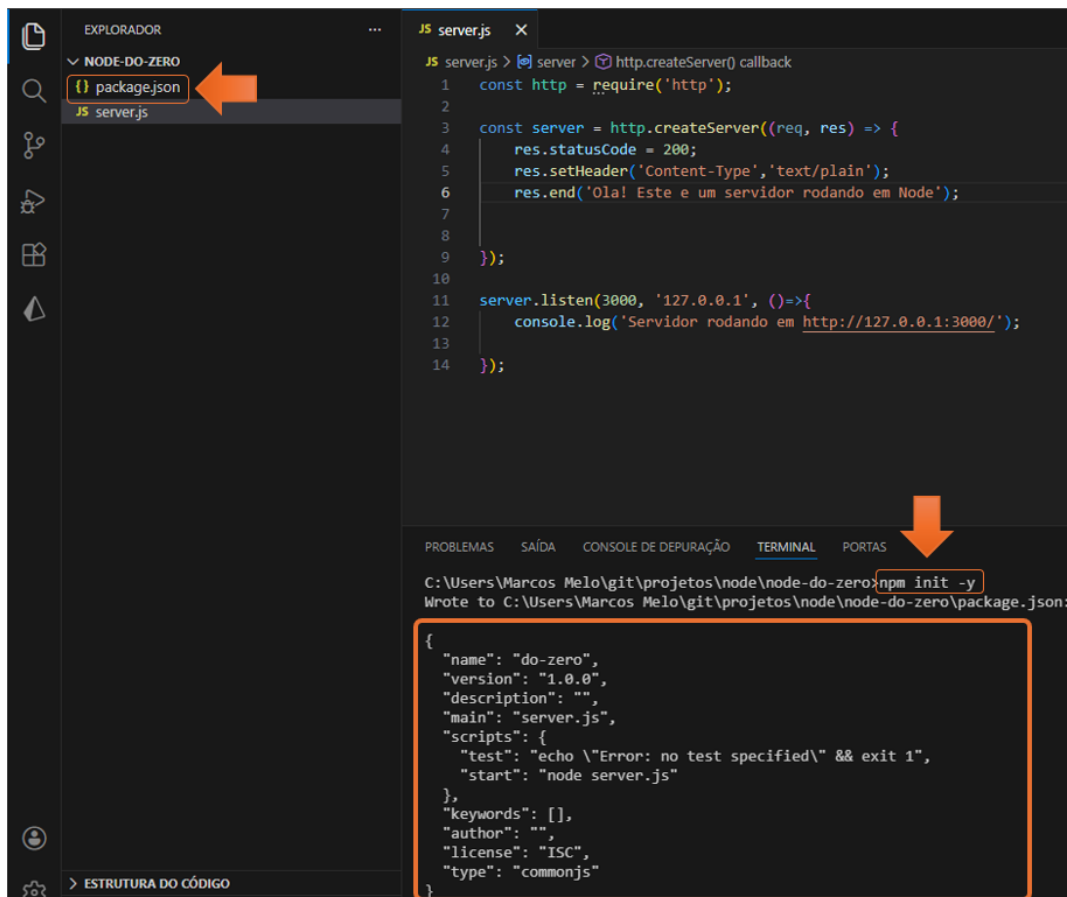
Usando o NPM

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:

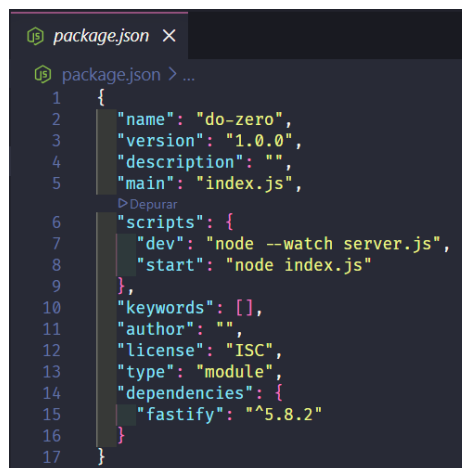
```
npm init -y
```

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;



```
package.json X
package.json > ...
1 {
2   "name": "do-zero",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "dev": "node --watch server.js",
8     "start": "node server.js"
9   },
10  "keywords": [],
11  "author": "",
12  "license": "ISC",
13  "type": "module",
14  "dependencies": {
15    "fastify": "^5.8.2"
16  }
17 }
```

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

```
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**

```
{} package.json > ...
1  {
2    "name": "do-zero",
3    "version": "1.0.0",
4    "description": "",
5    "main": "server.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1",
8      "start": "node server.js"
9    },
10   "keywords": [],
11   "author": "",
12   "license": "ISC",
13   "type": "commonjs"  ➡ module
14 }
```

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;

```
JS server.js > ...
1  import {createServer} from 'node:http';
2
3  const server = createServer((req, res) => {
4    res.statusCode = 200;
5    res.setHeader('Content-Type', 'text/plain');
6    res.end('Ola! Este e um servidor rodando em Node');
7
8  });
9
10
11 server.listen(3000, '127.0.0.1', ()=>{
12   console.log('Servidor rodando em http://127.0.0.1:3000/');
13
14 });
```

Após esta mudança o arquivo voltara a funcionar normalmente.

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

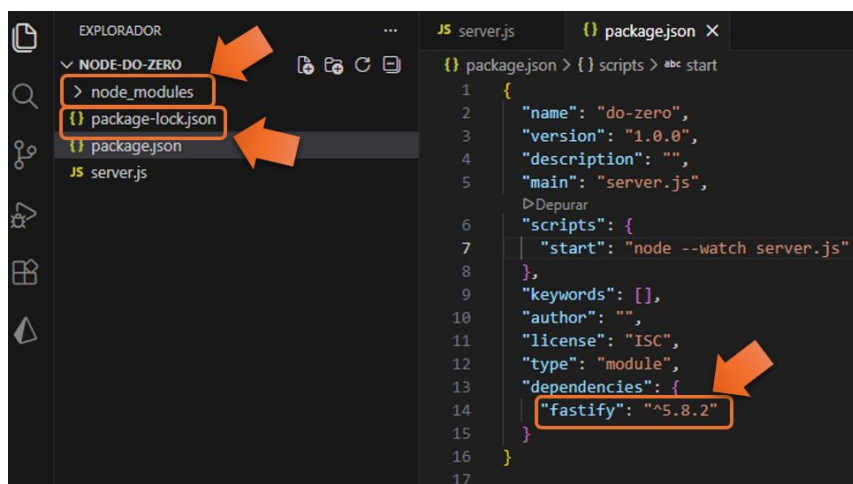
Para instalar a dependência do Framework Fastify digite no terminal o comando a seguir;

```
npm install fastify
```

Ao instalar o Fastify, repare que foi criado a pasta **node_modules**, dentro desta pasta esta todas as dependências necessárias para o Fastify 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.

```
server.js
1 import { fastify } from "fastify";
2
3
4 const server = fastify();
5
6 server.get("/", () => {
7   return "Olá Mundo!!";
8 });
9
10 server.listen({port:3000}, (err, address) => {
11   if (err) {
12     console.error(err);
13     process.exit(1);
14   }
15   console.log(`Servidor rodando em ${address}`);
16 });
```

Criando rotas

Após criar um servidor http e perceber o quão é simples cria-lo com o fastify, vamos compreender como funciona o sistema de rotas implementado pelo fastify.

Em um servidor **Node.js** usando **Fastify**, as **rotas** são os caminhos (URLs) que definem como o servidor deve responder a diferentes requisições HTTP (como GET, POST, PUT, DELETE).

Em resumo:

- Uma rota é composta por **método HTTP + caminho**.
- Cada rota tem um **handler** (função) que recebe a requisição e retorna uma resposta.
- Elas permitem organizar a lógica da aplicação, separando o que acontece em cada endpoint.
- **Endpoint:** é a combinação da rota com o método HTTP (GET, POST, PUT, DELETE etc.) que define uma ação específica.
- Exemplo:
 - GET /usuarios → retorna lista de usuários.
 - POST /usuarios → cria um novo usuário.
 - GET /usuarios/:id → retorna um usuário específico.
 - PUT /usuarios/:id → atualiza um usuário específico.
 - DELETE /usuarios/:id → deleta um usuário específico.

Entendendo a rota Get


```
serverjs  X
serverjs > ...
1  import { fastify } from "fastify";
2
3
4  const server = fastify();
5
6  server.get("/", () => {
7    return "Olá Mundo!!";
8  });
9
10 server.get("/sobre", () => {
11   return {
12     nome: "Do Zero",
13     versao: "1.0.0";
14   });
15
16 server.get("/contato/:email/:telefone", (req, reply) => {
17   return {
18     email: req.params.email,
19     telefone: req.params.telefone
20   };
21 });
22
23 server.listen({port:3000}, (err, address) => {
24   if (err) {
25     console.error(err);
26     process.exit(1);
27   }
28   console.log(`Servidor rodando em ${address}`);
29 });
```