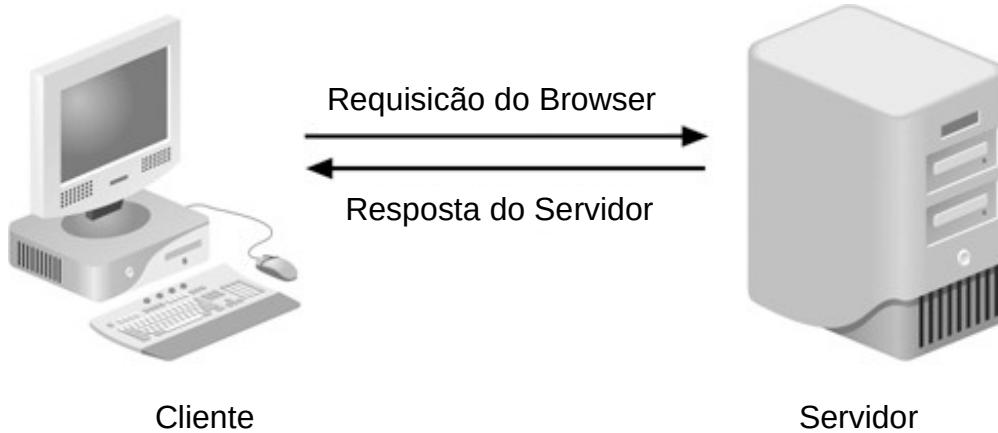




**Prof. David Fernandes de Oliveira
Instituto de Computação
UFAM**

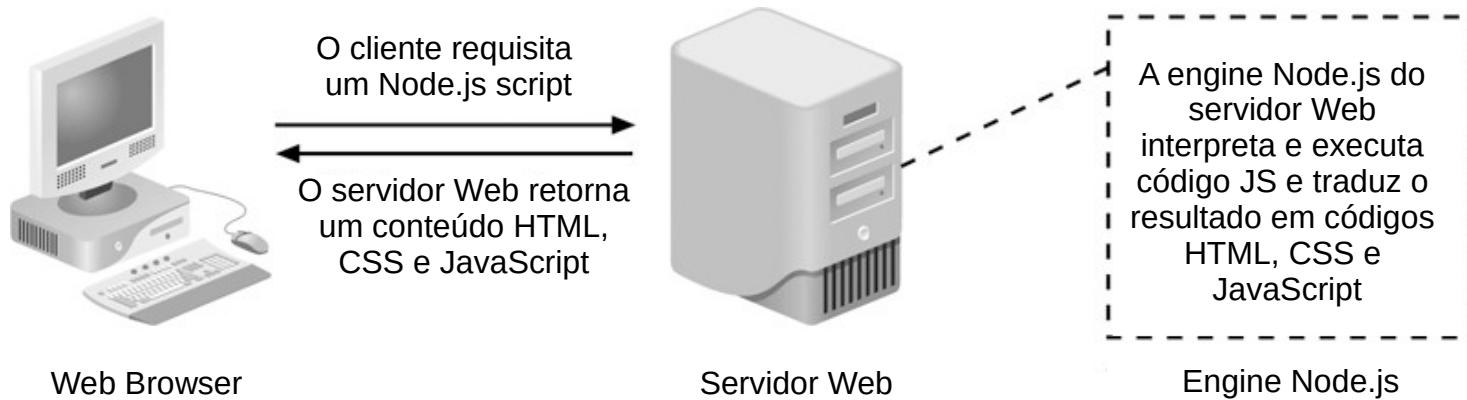
Arquitetura Cliente/Servidor

- Computador **Cliente** (frontend):
 - Interface com o usuário
 - Lê as requisições dos usuários, as submete para o servidor; recebe o conteúdo, e então apresenta este conteúdo para o usuário
- Computador **Servidor** (backend):
 - Processa as requisições dos usuários



Arquitetura Cliente/Servidor

- **Scripts do lado servidor** – projetados para executar do lado servidor, fornecendo a lógica principal da aplicação



Node.js: Javascript engine

- O **Node.js** é um ambiente de execução de código JavaScript baseado na engine **V8 da Google** e na biblioteca **libuv do C++**
- De código aberto e assíncrono, proporciona um poderoso conjunto de ferramentas para criação de scripts do lado servidor



libuv





Welcome to the libuv documentation

Overview

libuv is a multi-platform support library with a focus on asynchronous I/O. It was primarily developed for use by [Node.js](#), but it's also used by [Luvit](#), [Julia](#), [uvloop](#), and [others](#).

 Note

In case you find errors in this documentation you can help by sending [pull requests!](#)

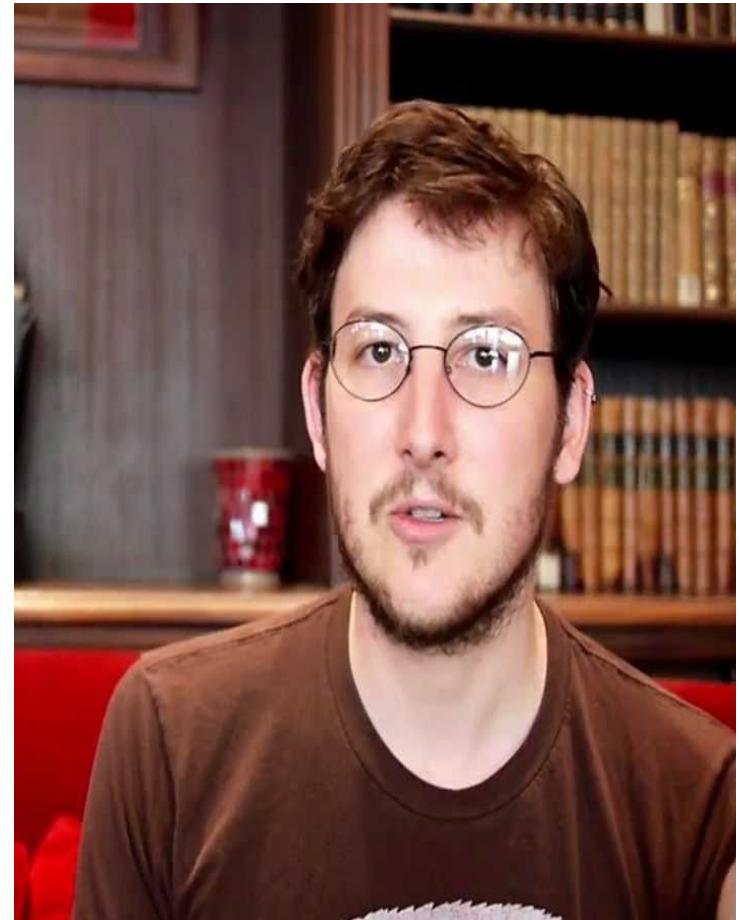
Features

- Full-featured event loop backed by epoll, kqueue, IOCP, event ports.
- Asynchronous TCP and UDP sockets
- Asynchronous DNS resolution
- Asynchronous file and file system operations
- File system events
- ANSI escape code controlled TTY
- IPC with socket sharing, using Unix domain sockets or named pipes (Windows)
- Child processes



Criação do Node.js

- Na JSConf 2009 Européia, um jovem programador chamado **Ryan Dahl** apresentou um ambiente de execução JavaScript para servidor baseada na engine V8 da Google
- Aproveitando o poder e a simplicidade do Javascript, essa engine facilitou o desenvolvimento de aplicações assíncronas
- Foi o ponto pé inicial para a criação do **Node.js**



- Na JSConf 2024, os organizadores apresentaram uma execução de um projeto baseado em Node.js.
- Aprovado por 99% das cidades que participaram, o projeto facilita a criação de aplicações web e mobile.
- Foi o projeto mais votado na votação do Node.js Foundation.

A MAIOR VOLTOU!

BRAZILJS



[CONF 24]

25 E 26 DE
ABRIL DE 2024



ARAÚJO VIANNA, PORTO ALEGRE

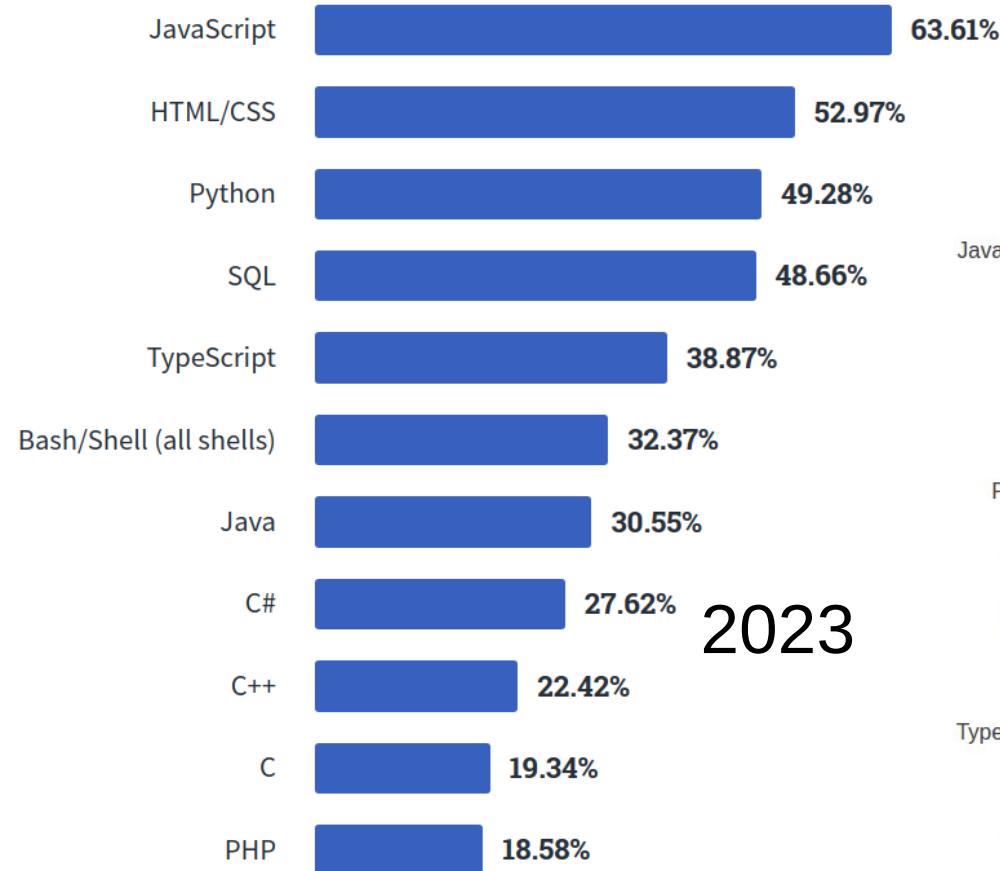


Porque usar Node.js?

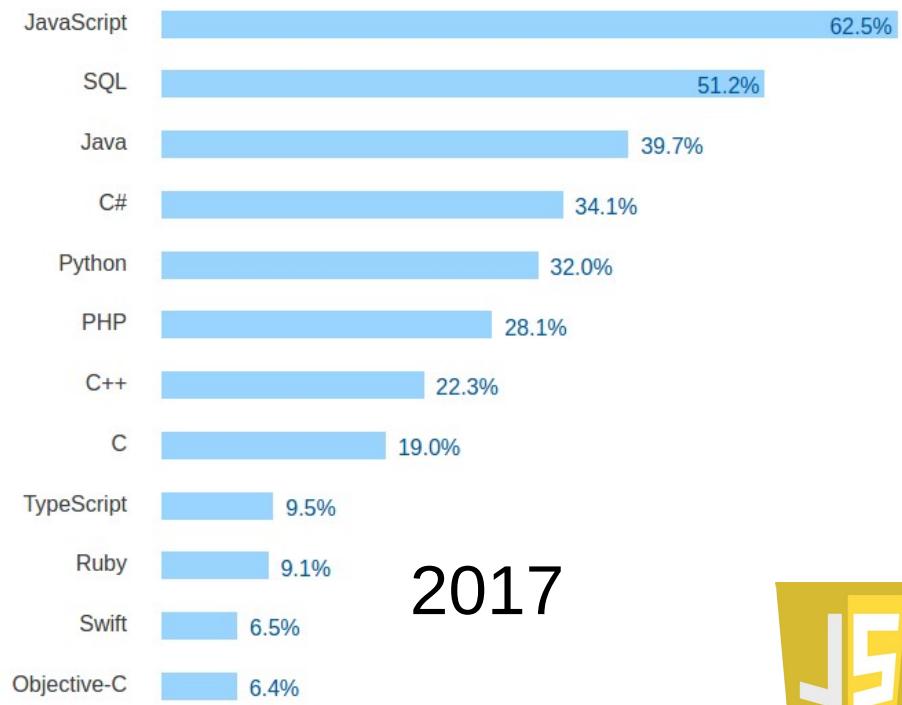
- **Comunidade Ativa** – O NPM (Node Package Manager) é o gerenciador de pacotes do Node.js e também é o maior repositório de softwares do mundo
- **Ele é rápido** – O V8 compila o JavaScript e executa usando código de máquina (compilação just-in-time – JIT). Execução single thread com I/O não bloqueante
- **Mesma linguagem no frontend e backend** – Não é preciso aprender uma nova linguagem, e nem trocar de contexto ao sair de um código do cliente e ir para um do servidor
- **Ambiente de inovação** – O Node.js é a opção da grande maioria das startups



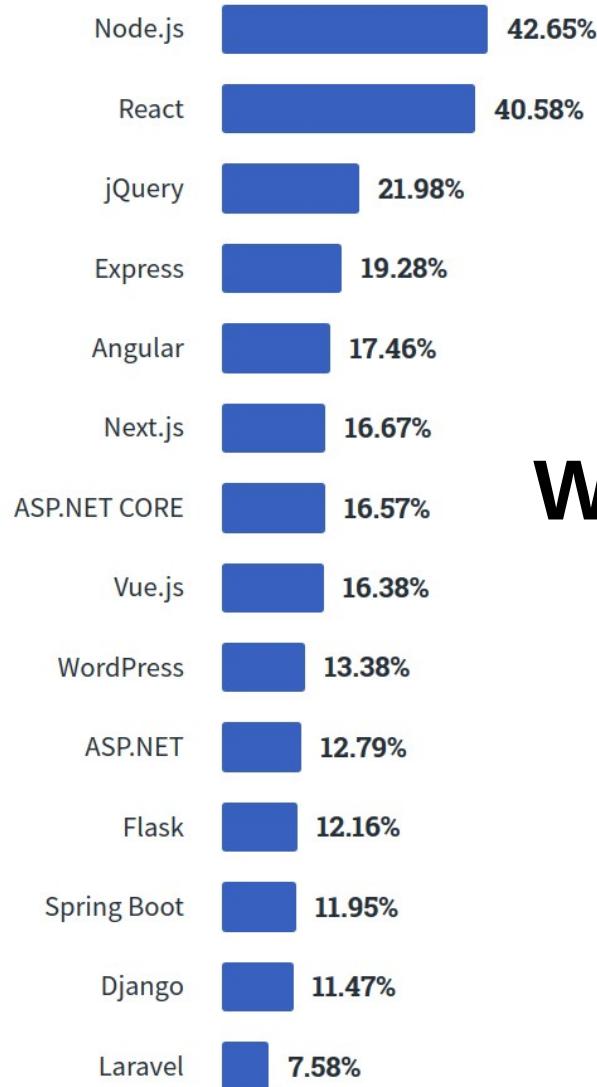
StackOverflow Surveys



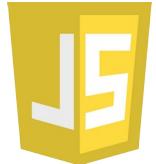
Programming, Scripting, and Markup Languages



StackOverflow Surveys



Web Frameworks 2023



Quem usa Node.js?



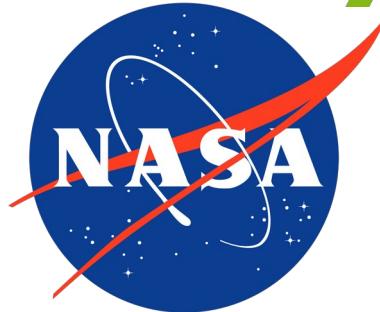
CODEBENCH

LinkedIn

ebay

PayPal™

UBER



Groupon

NETFLIX

YAHOO!®



Trello



GoDaddy™

Walmart
Save money. Live better.



Versões do Node.js

- Atualmente, o Node.js é mantido em suas principais versões:
Long Term Support (LTS) e **Current**

Download Node.js®

20.11.1 LTS

Recommended For Most Users

21.7.0 Current

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

For information about supported releases, see the [release schedule](#).



Versões do Node.js

- O Node.js está disponível nos repositórios da maioria das distribuições Linux, mas pode estar desatualizado
- Para instalar a versão mais recente (LTS), recomenda-se usar o pacote NVM:

```
$ nvm install lts
```

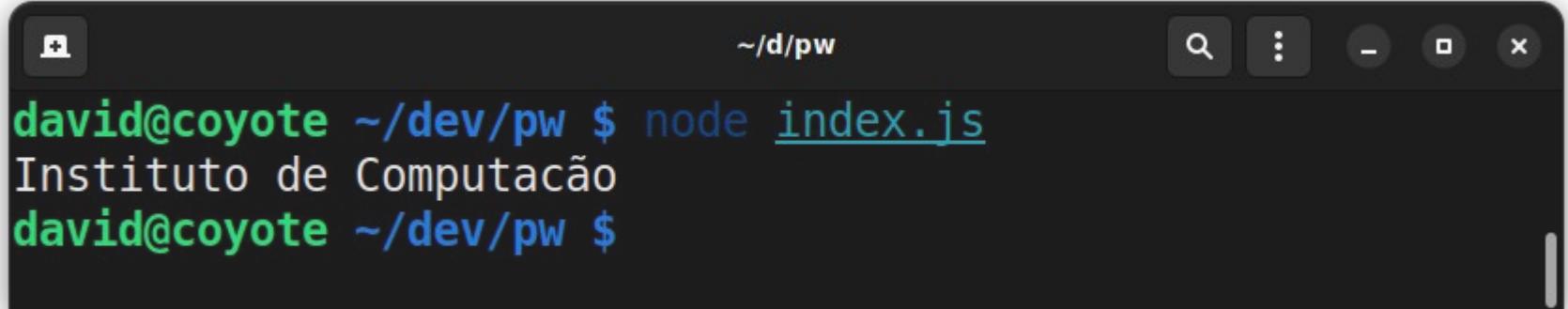
- Node Version Manager (NVM) é uma ferramenta que permite gerenciar várias versões do Node.js sistemas Linux
 - Com o NVM, pode-se alternar entre diferentes versões do Node.js, instalar novas e remover aquelas que não precisa mais



Executando Código JavaScript

- Para executar um código JavaScript usando **node.js** basta usar o intepretador **node** instalado no sistema operacional

```
// arquivo index.js
console.log("Instituto de Computacão")
```



A screenshot of a terminal window on a Mac OS X system. The window title bar shows the path `~/d/pw`. The terminal itself has a dark background and displays the following text:

```
david@coyote ~/dev/pw $ node index.js
Instituto de Computacão
david@coyote ~/dev/pw $
```



Módulos Embarcados do Node

- O Node.js é um intepretador JavaScript leve, cujos módulos embarcados (**built-in** ou **core modules**) provêem apenas funcionalidades básicas para o programador
 - Os módulos embarcados são carregados automaticamente assim que o processo Node.js inicia
 - Mesmo assim, é necessário importar (via **require** ou **import**) tais módulos antes de usá-los em sua aplicação:

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

- Alguns módulos embarcados do Node.js: **http**, **url**, **path**, **fs**, **os**, **sys**, **tty**, **cluster**, **process**, **timers** e **util**



Hello World Cliente Servidor

- Também podemos gerar conteúdo que poderá ser acessado pelo browser usando o protocolo HTTP

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

const server = http.createServer(function(req,res){
    res.writeHead(200,{"Content-Type":"text/html;charset=utf-8"});
    res.write("Instituto de Computação");
    res.end();
});

server.listen(3333);
```



Hello World Cliente Servidor

- Também podemos gerar conteúdo que poderá ser acessado pelo browser usando o protocolo HTTP

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, {"Content-Type": "text/html; charset=utf-8"});
  res.write("Instituto de Computação");
  res.end();
});
server.listen(3333);
```

http é um módulo do NodeJS

Os módulos podem ser importados através do comando **require**



O Módulo FS

- O **módulo FS** fornece operações de I/O que permitem acesso e interação com o sistema de arquivos
- Esse módulo não precisa ser instalado, já que ele faz parte do núcleo (core) do Node.js

```
const fs = require('fs')

fs.rename('imagem1.png', 'imagem2.png', function (err) {
  if (err) throw new Error(err);
});
```

Notem que o último parâmetro é uma função de **callback**, que é executada após a renomeação ser concluída.



O Módulo FS

- O **módulo FS** fornece operações de I/O que permitem acesso e interação com o sistema de arquivos
- Esse módulo não precisa ser instalado, já que ele faz parte do núcleo (core) do Node.js

```
const fs = require('fs')

fs.rename('imagem1.png', 'imagem2.png', function (err) {
  if (err) throw new Error(err);
})
```

Perceba que o método de importar bibliotecas do Node é diferente de outras linguagens. No node, a função **require()** retorna um objeto contendo um conjunto de métodos. No nosso exemplo, a função **rename** é um dos métodos do objeto retornado por **require('fs')**

após a renomeação
ser concluída.



O Módulo FS

- O **módulo FS** fornece operações de I/O que permitem acesso e interação com o sistema de arquivos
- Esse módulo não precisa ser instalado, já que ele faz parte do núcleo (core) do Node.js

```
const fs = require('fs')

fs.rename('imagem1.png', 'imagem2.png', function (err) {
  if (err) throw new Error(err);
})
```

Note que essa é uma forma bem simples e elegante de resolver o problema de conflitos de nomes entre as bibliotecas. Em outras linguagens, como Java e PHP, esse problema foi resolvido através de uma técnica chamada **NameSpaces**.

após a renomeação
ser concluída.



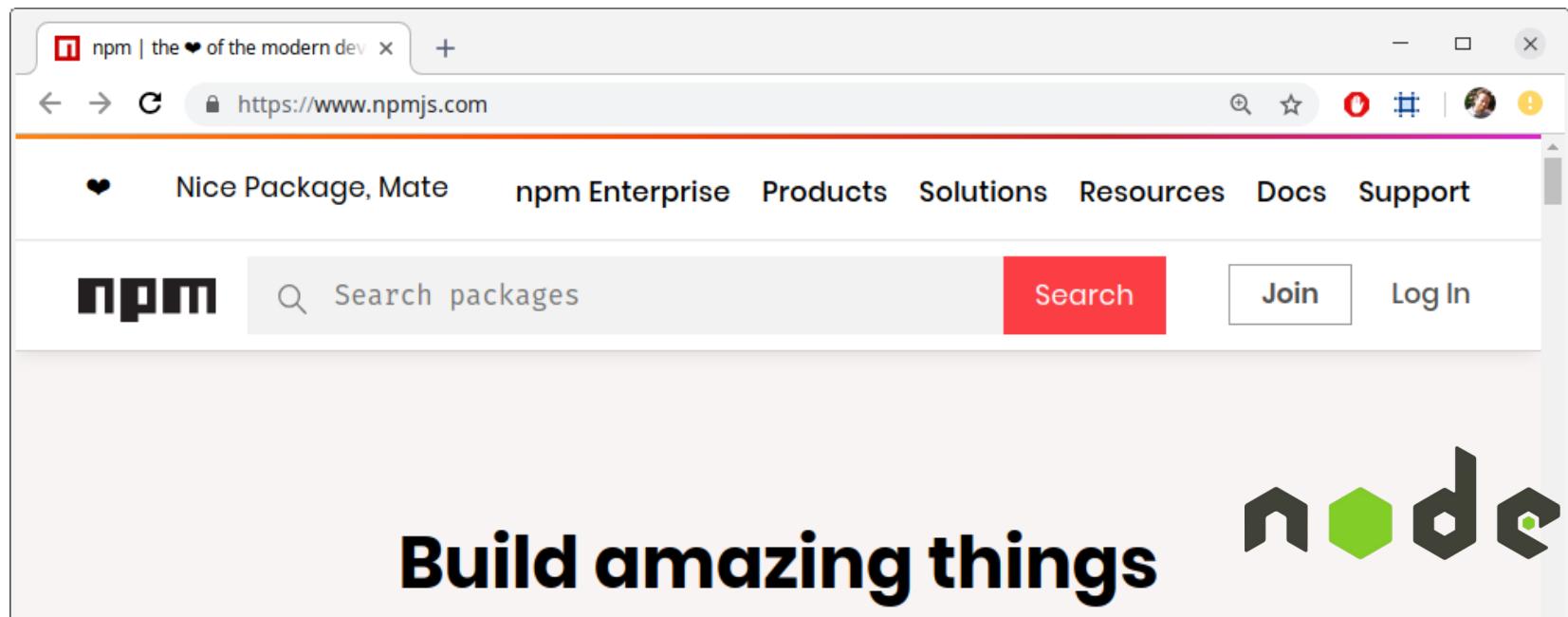
O Módulo FS

- O **módulo FS** possui vários outros métodos, dentre os quais podemos destacar:
 - **fs.access()**: verifica se o arquivo existe
 - **fs.chmod()**: muda as permissões de acesso do arquivo
 - **fs.close()**: fecha um descritor de arquivo
 - **fs.copyFile()**: copia um arquivo
 - **fs.mkdir()**: cria um novo diretório
 - **fs.open()**: abre um arquivo para leitura
 - **fs.readdir()**: lê o conteúdo de um diretório
 - **fs.readFile()**: lê o conteúdo de um arquivo
 - **fs.symlink()**: cria um link simbólico
 - **fs.unlink()**: apaga um arquivo ou link
 - **fs.writeFile()**: escreve dados em um arquivo



Node Package Manager – NPM

- O **Node Package Manager, NPM**, é uma ferramenta de linha de comando usada para instalar, atualizar ou desinstalar pacotes Node.js em sua aplicação
- O NPM possui um repositório de pacotes Node.js de código aberto: <https://www.npmjs.com/>



Build amazing things

Node Package Manager – NPM

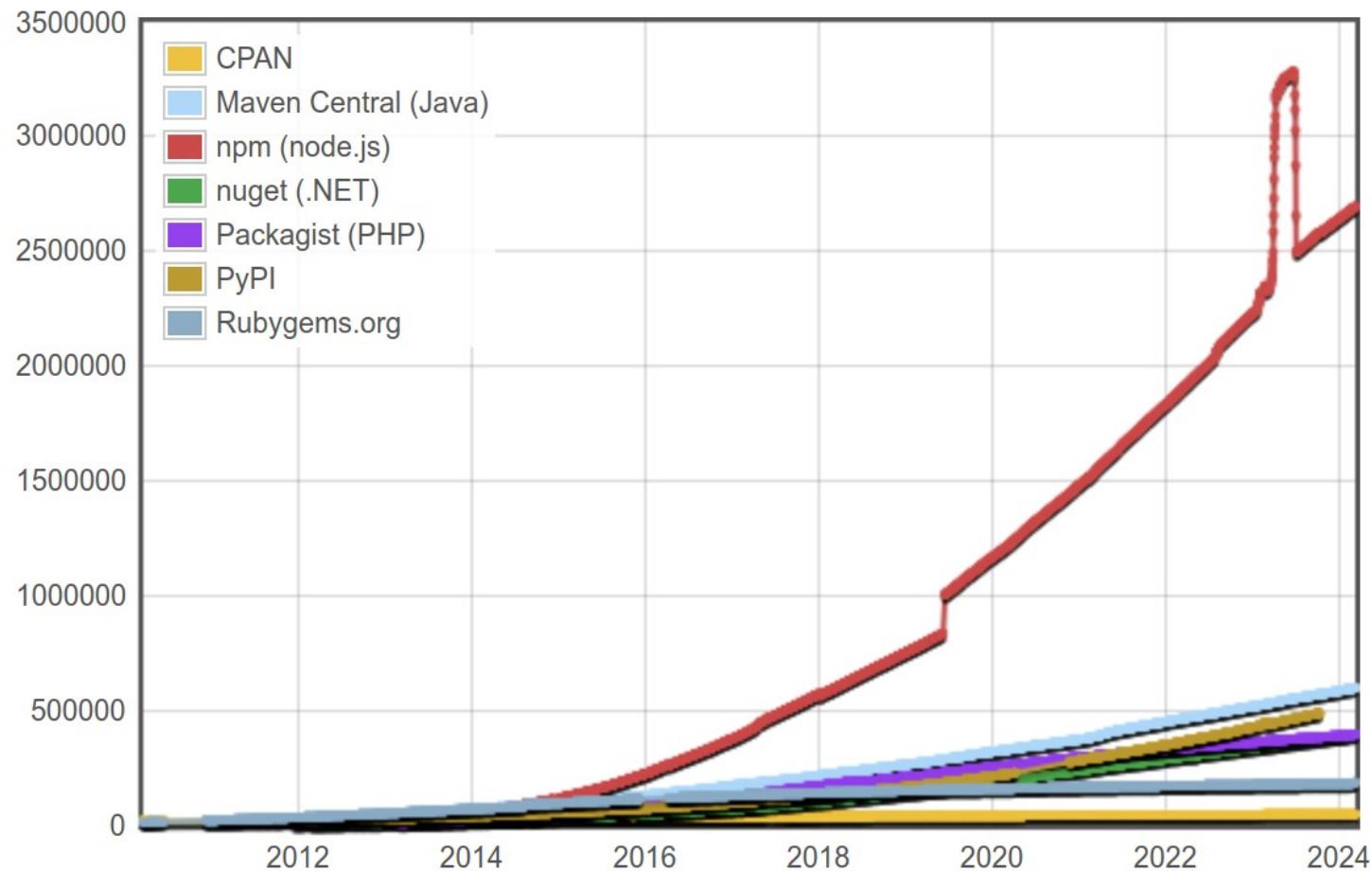
- O **Node Package Manager, NPM**, é uma ferramenta de linha de comando usada para instalar, atualizar ou desinstalar pacotes Node.js em sua aplicação
- O NPM possui um repositório de pacotes Node.js de código aberto: <https://www.npmjs.com/>

A comunidade Node.js ao redor do mundo é muito atuante e desenvolve inúmeros módulos de código aberto que são publicados neste repositório.

Build amazing things

The image shows a screenshot of a web browser displaying the npmjs.com homepage. A blue callout box is overlaid on the page, containing the text: "A comunidade Node.js ao redor do mundo é muito atuante e desenvolve inúmeros módulos de código aberto que são publicados neste repositório." Below the callout, there is a slogan: "Build amazing things". To the right of the slogan, the Node.js logo is displayed, consisting of the word "node" in lowercase letters where each letter is a different shape, followed by a small green "JS" logo.

Node Package Manager – NPM

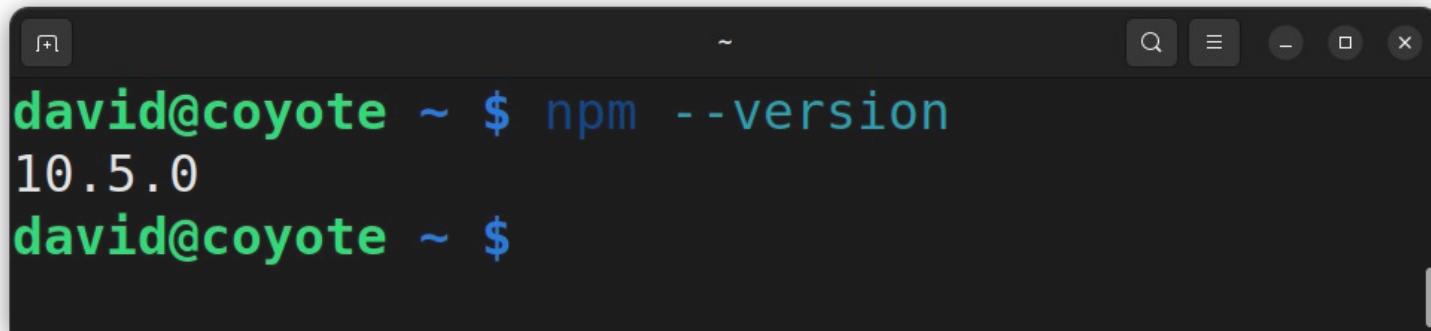


<http://www.modulecounts.com/>



Node Package Manager – NPM

- O **Node Package Manager**, NPM, precisa ser instalado no sistema operacional
 - No Linux Ubuntu, Debian e derivados, ele pode ser instalado com o comando **apt install npm**



```
david@coyote ~ $ npm --version
10.5.0
david@coyote ~ $
```

A screenshot of a dark-themed terminal window. The window title bar is visible at the top. Inside the terminal, the user 'david' is at the prompt '~ \$'. They type 'npm --version' and press enter. The terminal displays the output '10.5.0' followed by another prompt. The window has standard window controls (minimize, maximize, close) in the top right corner.

O Arquivo package.json

- O primeiro passo no desenvolvimento de uma aplicação node.js é a criação de um arquivo chamado **package.json**
 - A função desse arquivo é armazenar vários metadados sobre a aplicação, incluindo suas dependências

```
{  
  "name": "hello-world",  
  "author": "David Fernandes",  
  "private": true,  
  "version": "0.0.1",  
  "dependencies": {}  
}
```

O Arquivo package.json

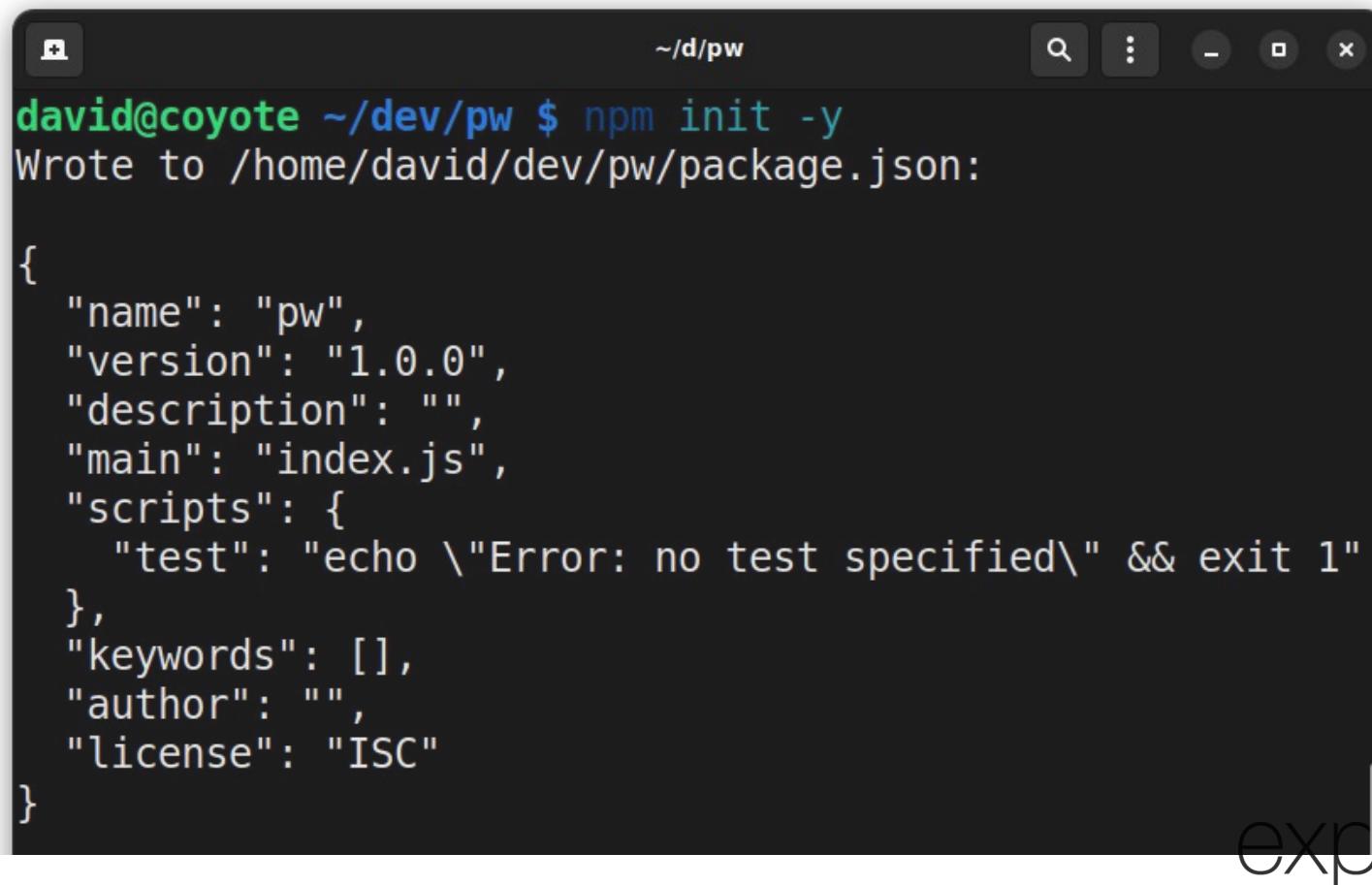
- O primeiro passo no desenvolvimento de uma aplicação node.js é a criação de um arquivo chamado **package.json**
 - A função desse arquivo é armazenar vários metadados sobre a aplicação, incluindo suas dependências

```
{  
  "name": "hello-world",  
  "author": "David Fernandes",  
  "private": true,  
  "version": "0.0.1",  
}
```

Para criar um novo arquivo **package.json** com os valores desejados, use o comando **npm init**. Esse comando fará algumas perguntas para o programador, e criará um arquivo **package.json** baseado nas suas respostas.

O Arquivo package.json

- Outra opção é executar o comando **npm init** com a opção **-y**, que irá criar um package.json com valores iniciais

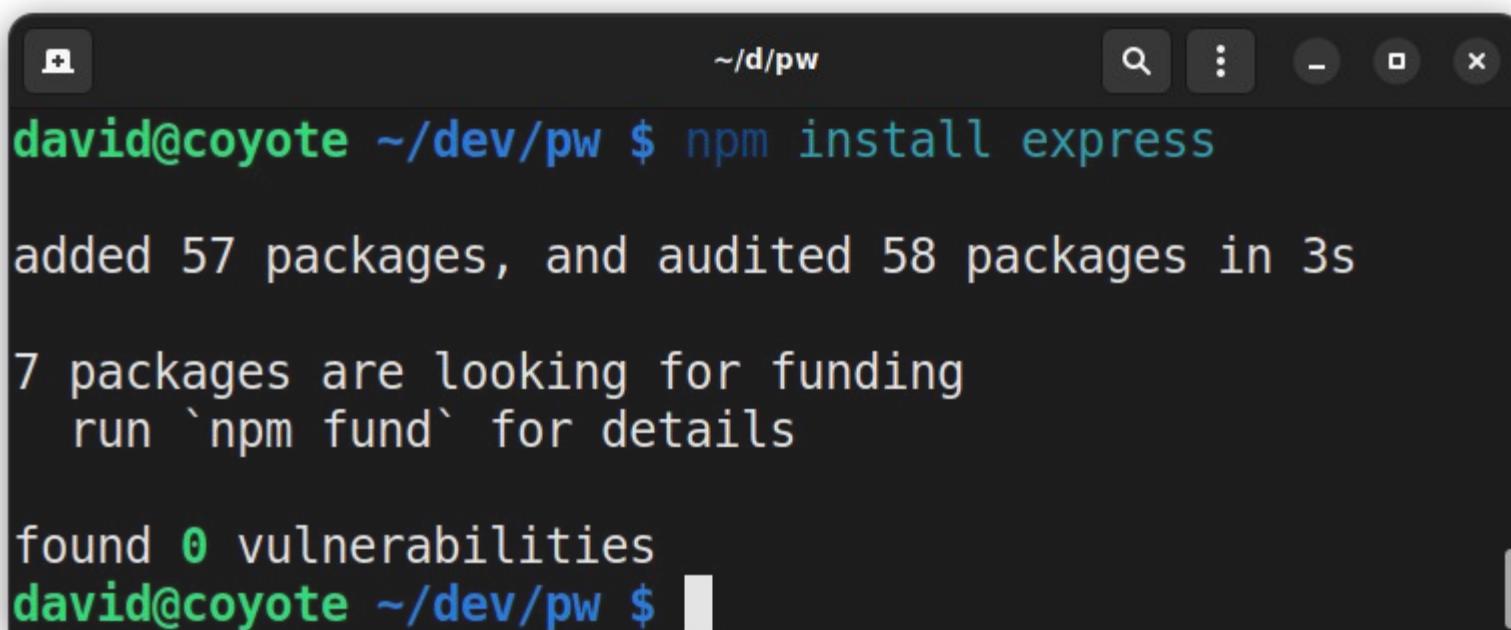


```
david@coyote ~/dev/pw $ npm init -y
Wrote to /home/david/dev/pw/package.json:

{
  "name": "pw",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Adicionando o Express no Projeto

- Para incluirmos o framework express no projeto, basta executarmos o comando **npm install express**
 - O pacote express é adicionado automaticamente como uma dependência do projeto



```
~d/pw
david@coyote ~/dev/pw $ npm install express
added 57 packages, and audited 58 packages in 3s
7 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
david@coyote ~/dev/pw $
```

Adicionando o Express no Projeto

- Para incluirmos o framework express no projeto, basta executar:

```
david@coyote ~/dev/pw $ cat package.json
{
  "name": "pw",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  }
}
david@coyote ~/dev/pw $
```

Adicionando o Express no Projeto

- Para incluirmos o framework express no projeto, basta executar:

```
david@coyote ~/dev/pw $ cat package.json
{
  "name": "pw",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {}
```

```
david@coyote ~/dev/pw $ npm install express
```

Observe que a definição de dependência do Express se refere à versão 4.18.2. A versão de um dado módulo é representada por três valores: **Major version** (4), **Minor version** (18) e **Patch version** (2).

```
run "license": "ISC",
  "dependencies": {
found 0 "express": "^4.18.2"
david@coyote ~/dev/pw $ }
```

```
david@coyote ~/dev/pw $
```

Adicionando o Express no Projeto

- Para incluirmos o framework express no projeto, basta executar



```
~/d/pw
```

A screenshot of a terminal window with a dark theme. The title bar says "~/d/pw". The window contains the command "npm install express". The terminal is running on a Mac OS X system, as indicated by the window controls.

Note também que a versão do Express é prefixada por um ^, indicando que ele pode ser atualizado caso seja lançado uma nova **minor version** do framework. Isto é, sua aplicacão é dependente do Express versão 4.*.*

Alguns módulos podem ser prefixados com um ~, indicando eles só podem ser atualizados caso seja lançado um novo patch desses módulos.

Observe que a definição de dependência do Express se refere à versão 4.18.2. A versão de um dado módulo é representada por três valores: **Major version** (4), **Minor version** (18) e **Patch version** (2).

```
run "license": "ISC", details
  "dependencies": {
found 0 "express": "^4.18.2"
david@coyote ~/dev/pw $ 
}
david@coyote ~/dev/pw $
```

A screenshot of a terminal window showing the output of the "npm install express" command. The terminal shows the command being run, followed by the output which includes the installed module "express" at version "4.18.2". The prompt "david@coyote ~/dev/pw \$" appears twice at the end.

Node Package Manager – NPM

- O comando `npm install` instala o pacote desejado e todas suas dependências no diretório `node_modules`

```
david@coyote ~/dev/pw $ ls node_modules
accepts/
array-flatten/
body-parser/
bytes/
call-bind/
content-disposition/
content-type/
cookie/
cookie-signature/
debug/
depd/
destroy/
ee-first/
encodeurl/
david@coyote ~/dev/pw $ escape-html/
etag/
express/
finalhandler/
forwarded/
fresh/
function-bind/
get-intrinsic/
has/
has-symbols/
http-errors/
iconv-lite/
inherits/
ipaddr.js/
media-typer/
merge-descriptors/
methods/
mime/
mime-db/
mime-types/
ms/
negotiator/
object-inspect/
on-finished/
parseurl/
path-to-regexp/
proxy-addr/
qs/
range-parser/
raw-body/
safe-buffer/
safer-buffer/
send/
serve-static/
setprototypeof/
side-channel/
statuses/
toidentifier/
type-is/
unpipe/
utils-merge/
vary/
```



Node Package Manager – NPM

- O comando **npm install** instala o pacote desejado e todas suas dependências no diretório **node_modules**

```
~d/pw
Outros comandos do NPM:
$ npm update <package_name>
$ npm uninstall <package_name>
$ npm ls
```


david@coyote ~/dev/pw \$ npm ls --all

pw@1.0.0 /home/david/dev/pw

 express@4.18.2

 accepts@1.3.8

 mime-types@2.1.35

 |- Ocom

 |- todas

 |- neg

 |- array

 |- body-

 |- byt

 |- con

 |- davide

 |- accel

 |- arra

 |- body

 |- des

 |- htt

 |- ico

 |- pa

 |- s

 |- cont

 |- on-

 |- qs@

 |- raw

 |- deb

 |- h

 |- lept

 |- ee-fir

 |- u

 |- typ

 |- unp

 |- davi

 |- i

 |- typ

 |- unp

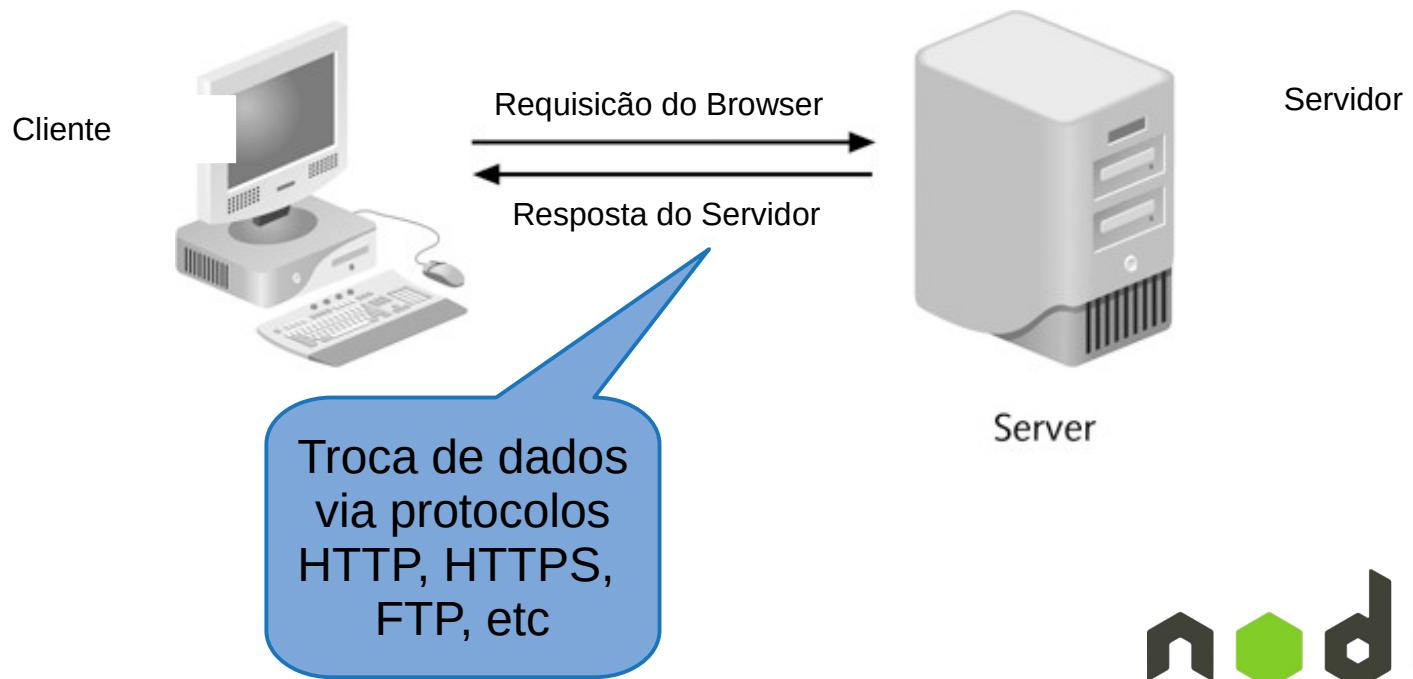
 |- davi



**Meu backup do diretório
node_modules!**

Servidores Web

- O Node.JS pode ser usado para desenvolver todo tipo de aplicação, mas o seu principal uso é a criação de **Web Apps**
- Nesse contexto, um **servidor Web** é um sistema que responde a solicitações de clientes feitas pela World Wide Web



Servidores Web

- Para criarmos um servidor Web com o Node.js, podemos utilizar os módulos built-in **http** ou **https**

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

const server = http.createServer(function(req,res){
    res.writeHead(200,{"Content-Type":"text/html;charset=utf-8"});
    res.write("Instituto de Computação");
    res.end();
});

server.listen(3333);
```



Servidores Web

- Para criarmos um servidor Web com o Node.js, podemos utilizar os módulos built-in **http** ou **https**

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

const server = http.createServer(function(req,res){
    res.writeHead(200, {"Content-Type": "text/html; charset=utf-8"});
    }, {
        req - objeto representando a requisição do usuário.  

        res - objeto representando a resposta enviada para o usuário.
    });
server.listen(3333);
```

david@coyote ~/dev/pw \$ node index.js



Passagem de Parâmetro – ARGV

- Os argumentos de linha de comando podem ser acessados através do objeto **process.argv**

```
process.argv.forEach((val, index) => {
  console.log(`#${index}: ${val}`)
})
```



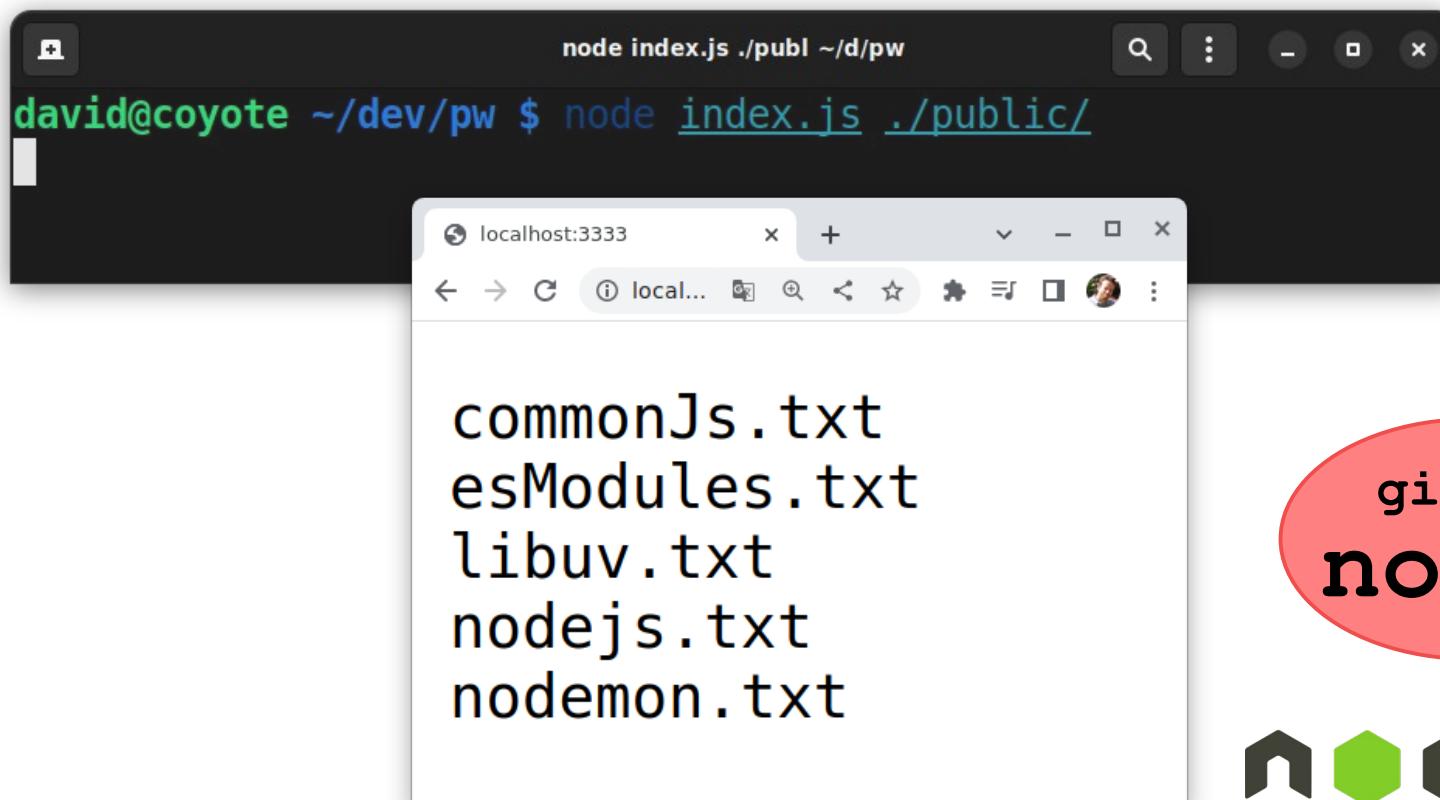
A screenshot of a terminal window titled 'david@coyote ~/dev/pw'. The command 'node index.js icomp ufram' is run, followed by four lines of output showing the arguments at indices 0 through 3.

```
david@coyote ~/dev/pw $ node index.js icomp ufram
0: /usr/bin/node
1: /home/david/dev/pw/index.js
2: icomp
3: ufram
david@coyote ~/dev/pw $
```



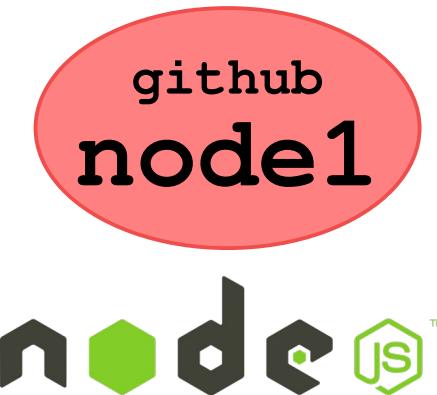
Exercício I – Parte 1

- Usando a função **readdir** do módulo **fs**, desenvolva um programa que aceita o nome de um diretório como parâmetro, e então cria um servidor Web capaz de retornar uma página contendo a lista de arquivos e subdiretórios do diretório informado



```
node index.js ./public ~d/pw
david@coyote ~/dev/pw $ node index.js ./public/
localhost:3333
```

commonJs.txt
esModules.txt
libuv.txt
nodejs.txt
nodemon.txt



Variáveis de Ambiente

- Variáveis de ambiente são definidas fora de um programa, geralmente por um provedor da nuvem ou um SO
- No Node, as variáveis de ambiente constituem uma ótima maneira de definir as configurações locais de um ambiente
 - Como URLs, portas, chaves de autenticação, senhas, etc
- Por exemplo, para criar uma variável de ambiente para armazenar a senha local do banco de dados:

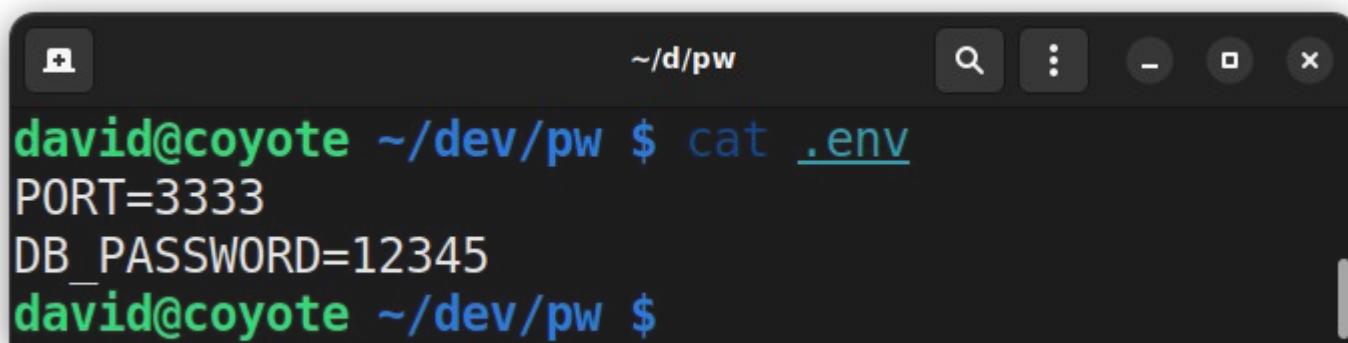
```
process.env.DB_PASSWORD="12345"
```

Por convenção,
as variáveis de
ambiente são
escritas em
caixa alta



Variáveis de Ambiente

- Uma opção para definir as variáveis de ambiente é através de arquivos não versionados no diretório raiz da aplicação
 - Exemplos de nomes para esses arquivos são **.env**, **.env.development**, **.env.production**



```
david@coyote ~/dev/pw $ cat .env
PORT=3333
DB_PASSWORD=12345
david@coyote ~/dev/pw $
```



Variáveis de Ambiente

- Para que as variáveis de ambiente sejam carregadas na aplicação, podemos usar pacotes como o **dotenv**

```
~/.d/pw
david@coyote ~/dev/pw $ npm install dotenv
up to date, audited 59 packages in 864ms
7 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
david@coyote ~/dev/pw $ |
```



Variáveis de Ambiente

- Para que as variáveis de ambiente sejam carregadas na aplicação, podemos usar pacotes como o **dotenv**

```
const http = require('http');
require('dotenv').config();

const PORT = process.env.PORT ?? 3333;

const server = http.createServer(function (req, res) {
  res.writeHead(200, {"Content-Type": "text/html; charset=utf-8"});
  res.write("Instituto de Computação");
  res.end();
});

server.listen(PORT);
```



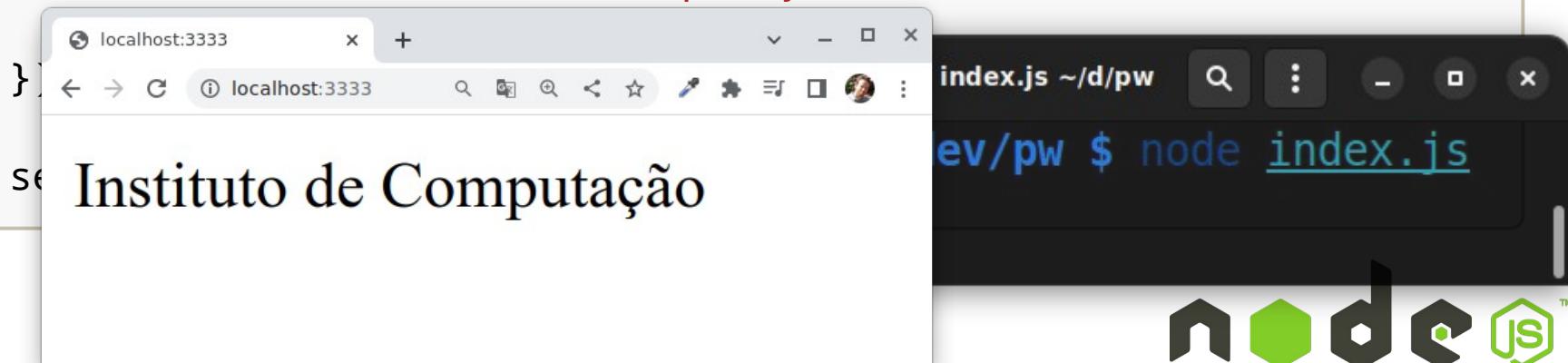
Variáveis de Ambiente

- Para que as variáveis de ambiente sejam carregadas na aplicação, podemos usar pacotes como o **dotenv**

```
const http = require('http');
require('dotenv').config();

const PORT = process.env.PORT ?? 3333;

const server = http.createServer(function (req, res) {
  res.writeHead(200, {"Content-Type": "text/html; charset=utf-8"});
  res.write("Instituto de Computação");
})
```



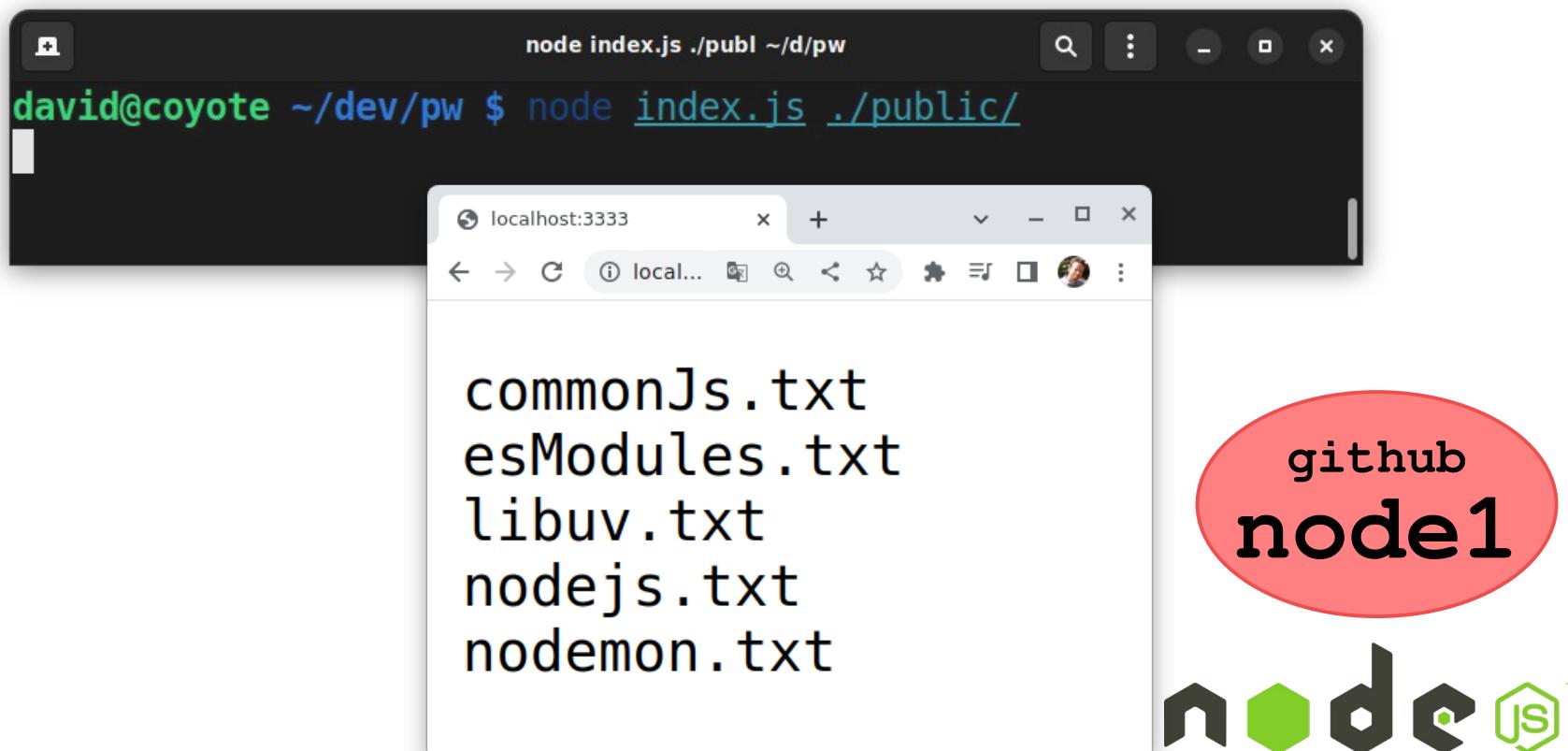
Variáveis de Ambiente

- As arquivos `.env` devem ser adicionados no `.gitignore`, pois as variáveis de ambiente estão relacionadas com o ambiente do usuário que está rodando a aplicação
- No entanto, quando um novo desenvolvedor faz o clone do repositório, é importante que ele saiba o nome das variáveis que ele precisa definir em seu ambiente
- Para resolver isso, pode-se criar arquivos versionáveis contendo variáveis de ambientes de exemplo
 - Por exemplo, tais arquivos podem ter nomes como `.env.example`



Exercício I – Parte 2

- Crie um arquivo **.env** para armazenar a porta que será usada pelo servidor Web de sua aplicação. Adicione o arquivo **.env** no **.gitignore**, e em seguida crie um arquivo **.env.example** contendo um exemplo de arquivo **.env** válido.



Scripts de Execução

- A propriedade **scripts** do arquivo **package.json** permite a criação de atalhos para scripts relacionados com a app

```
david@coyote ~/dev/pw $ cat package.json
{
  "name": "pw",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "dotenv": "^16.0.3",
    "express": "^4.18.2"
  }
}
david@coyote ~/dev/pw $
```

Ao ser criado,
o **package.json**
vem só com um
script test de
exemplo



Scripts de Execução

- A propriedade **scripts** do arquivo `package.json` permite a criação de atalhos para scripts reutilizáveis na app

The screenshot shows two terminal windows. The top window displays the command `cat package.json`, showing the initial empty JSON file:

```
{}
```

The bottom window shows the command `npm test` being run, which triggers the `test` script defined in the `scripts` section of the `package.json` file:

```
david@coyote ~/dev/pw $ npm test
> pw@1.0.0 test
> echo "Error: no test specified" && exit 1
Error: no test specified
david@coyote ~/dev/pw $
```

A blue callout bubble points from the text "Para executar o script de exemplo," to the `npm test` command in the bottom window.

A second blue callout bubble points from the text "Ao ser criado, o package.json vem só com um script test de exemplo" to the `test` script definition in the `scripts` section of the `package.json` file.

Icons for Node.js, NPM, and JavaScript are visible in the bottom right corner.

Scripts de Execução

- A propriedade **scripts** do arquivo `package.json` permite a criação de atalhos para scripts reutilizáveis na app

Para executar o script de exemplo, podemos usar o comando `npm test`

```
david@coyote ~/dev/pw $ cat package.json
{
```

Normalmente, para executar um script, precisamos usar o comando `npm run <script>`. No entanto, os comandos **npm test**, **npm start**, **npm restart** e **npm stop** são aliases para `npm run test`, `npm run start`, `npm run restart` e `npm run stop`, respectivamente.

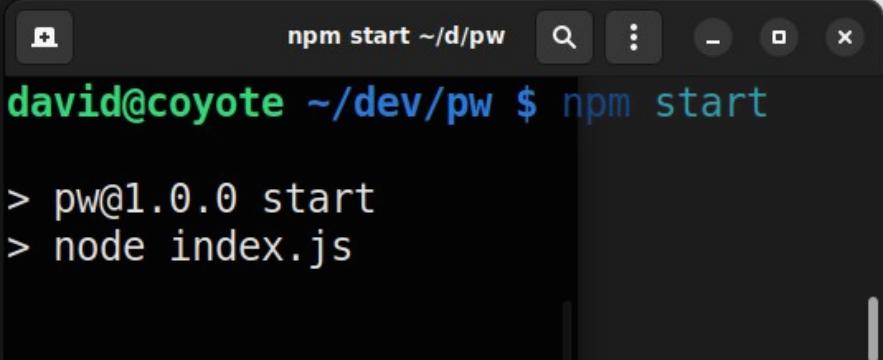
o package.json
vem só com um
script test de
exemplo

```
Error: no test specified
david@coyote ~/dev/pw $ 
dependencies: {
    "dotenv": "^16.0.3",
    "express": "^4.18.2"
}
}
david@coyote ~/dev/pw $
```



Scripts de Execução

- Podemos remover o script de exemplo (test) e adicionar um script para executar a aplicação que está sendo desenvolvida



A screenshot of a terminal window titled "npm start ~/d/pw". The command "npm start" is being run, and the output shows the package.json script being executed: "node index.js".

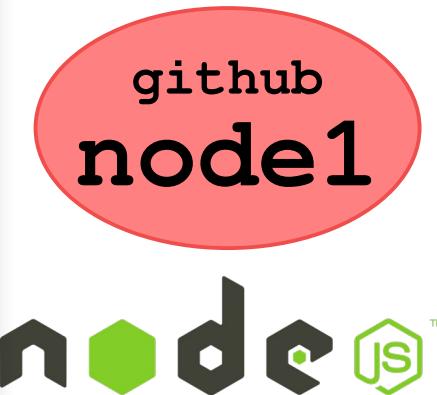
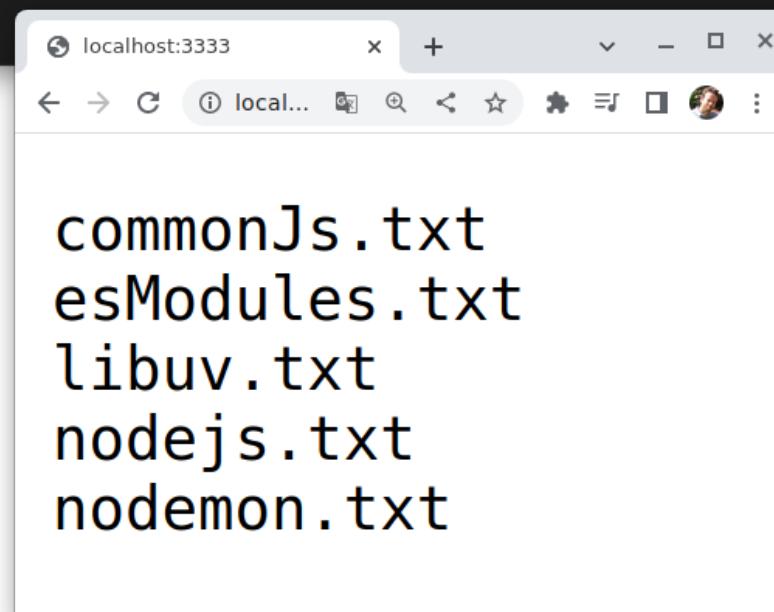
```
~d/pw
david@coyote ~/dev/pw $ cat package.json
{
  "name": "pw",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "dotenv": "^16.0.3",
    "express": "^4.18.2"
  }
}
david@coyote ~/dev/pw $ npm start
npm start ~/d/pw
david@coyote ~/dev/pw $ npm start
> pw@1.0.0 start
> node index.js
```



Exercício I – Parte 3

- Crie um script **npm start** para a sua aplicação.

```
npm start ./public ~/d/pw
david@coyote ~/dev/pw $ npm start ./public
> pw@1.0.0 start
> node index.js ./public
```



Nodemon

- Sempre que uma alteração é feita no código da aplicação, é preciso reiniciá-la para que as alterações tenham efeito
- O nodemon é um pacote que serve para eliminar essa etapa extra de nosso seu fluxo de trabalho
- A ideia desse pacote é muito simples: reiniciar a aplicação sempre que houver uma mudança no seu código fonte



nodemon



Nodemon

- Para usar o nodemon, você pode instalá-lo como uma dependência de **desenvolvimento** de sua aplicação

```
david@coyote ~/dev/pw $ npm install nodemon --save-dev
added 32 packages, and audited 91 packages in 1.13s
10 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
david@coyote ~/dev/pw $
```

Uma dependência de desenvolvimento não é instalada em um ambiente de produção



Nodemon

- O nodemon cria um link simbólico em **node_modules/.bin**, que deverá ser usado para iniciar a aplicação

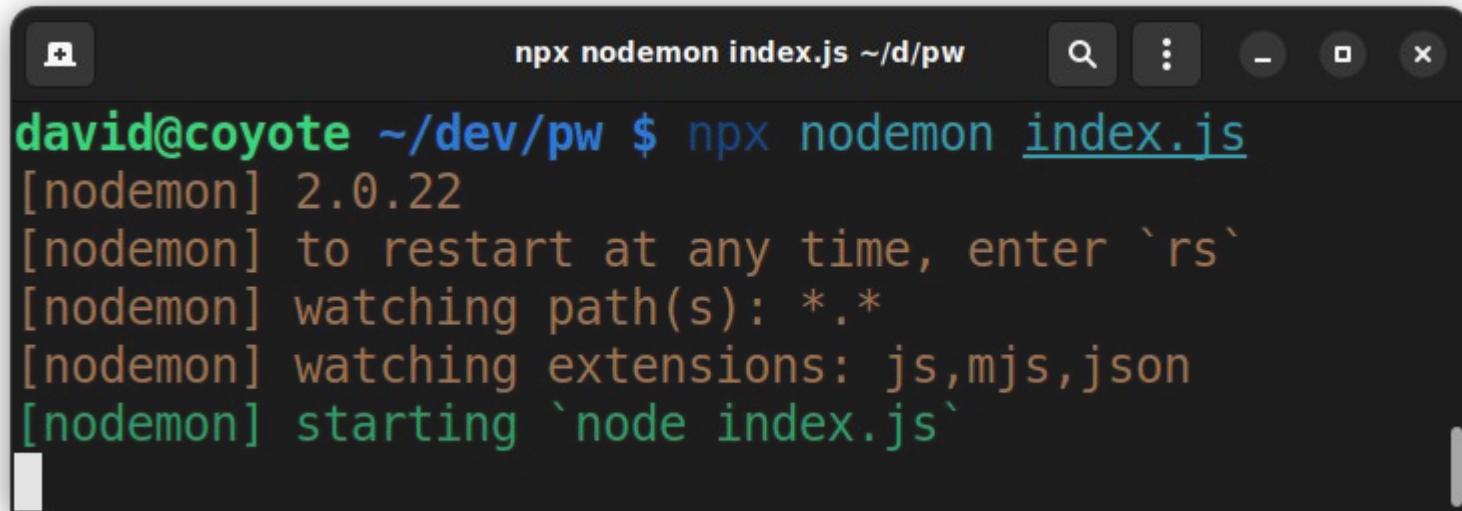
```
david@coyote ~/dev/pw/node_modules/.bin $ ls -la
total 8
drwxrwxr-x  2 david david 4096 mai 14 07:11 .
drwxrwxr-x 89 david david 4096 mai 14 07:11 ..
lrwxrwxrwx  1 david david   14 mai 11 09:49 mime -> ../mime/cli.js*
lrwxrwxrwx  1 david david   25 mai 14 07:11 nodemon -> ../node/bin/nodemon.js*
lrwxrwxrwx  1 david david   25 mai 14 07:11 nodetouch -> ../touch/bin/nodetouch.js*
lrwxrwxrwx  1 david david   19 mai 14 07:11 nopt -> ../nopt/bin/nopt.js*
lrwxrwxrwx  1 david david   20 mai 14 07:11 semver -> ../semver/bin/semver*
david@coyote ~/dev/pw/node_modules/.bin $
```

- Os arquivos executáveis do diretório **.bin** podem ser executados através do comando **npx**
 - npx** é um executor de pacote **npm**, e serve para executar scripts dos pacotes instalados via **npm**



Nodemon

- Para inicializar a aplicação através do nodemon, podemos usar o comando **npx nodemon index.js**



A screenshot of a terminal window titled "npx nodemon index.js ~/d/pw". The window shows the command being run and its output. The output includes the nodemon version (2.0.22), instructions to restart (`rs`), the watched path (`./*`), extensions (`.js,.mjs,.json`), and the command starting (`node index.js`).

```
david@coyote ~/dev/pw $ npx nodemon index.js
[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ./*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
```



Nodemon

- Para inicializar a aplicação através do nodemon, podemos usar o comando **npx nodemon index.js**

```
npx nodemon index.js ~/d/pw
david@coyote ~/dev/npx $ npx nodemon index.js ~/d/pw
[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`[nodemon] watching path(s): *.*[nodemon] watching extensions: js,mjs,json[nodemon] starting `node index.js`[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
```

Quando ocorre uma mudança no código, o nodemon reinicia a aplicação



Nodemon

- Podemos editar o **package.json** e criar um script para ambiente de desenvolvimento e outro para produção

```
{  
  "name": "pw",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "npx nodemon index.js",  
    "start:prod": "node index.js"  
  },  
  "keywords": [],  
  "license": "ISC",  
  "dependencies": {  
    "dotenv": "^16.4.5"  
  },  
  "devDependencies": {  
    "nodemon": "^3.1.0"  
  }  
}
```



Exercício I – Parte 4

- Instale o nodemon em sua aplicação e adicione os scripts **start** e **start:prod** no **package.json** de sua aplicação

The terminal window shows the command `npm start ./public ~/d/pw` being run, followed by the output of the `start` script which is `nodemon index.js ./public`. The browser window shows the URL `localhost:3333` and a list of files: `commonJs.txt`, `esModules.txt`, `libuv.txt`, `nodejs.txt`, and `nodemon.txt`.

github
node1

node JS

Variável de Ambiente NODE_ENV

- Sua aplicação pode precisar de configurações diferentes para ambientes de **produção** e **desenvolvimento**
- O Node.js sempre assume que a aplicação está rodando em um ambiente de **desenvolvimento**
- No entanto, podemos definir o ambiente de execução através da variável de ambiente **NODE_ENV**
- Por exemplo, em ambientes Linux isso pode ser feito através do seguinte comando (em shell bash):

```
export NODE_ENV=production
```



Variável de Ambiente NODE_ENV

- Também podemos definir o ambiente de execução usando os scripts do arquivo **package.json**

```
"scripts": {  
  "start": "NODE_ENV=development nodemon index.js",  
  "start:prod": "NODE_ENV=production node index.js"  
},
```

- A partir disso, ao invés de termos apenas um arquivo **.env**, podemos definir dois arquivos separados:
 - Um arquivo **.env.development** que será usado em ambiente de desenvolvimento, e
 - Um arquivo **.env.production** que será usado em ambiente de produção



Variável de Ambiente NODE_ENV

- Para selecionar o arquivo de configurações correto de acordo com o seu ambiente, podemos usar o seguinte código:

```
const dotenv = require("dotenv")
dotenv.config({ path: `./.env.${process.env.NODE_ENV}` })
```



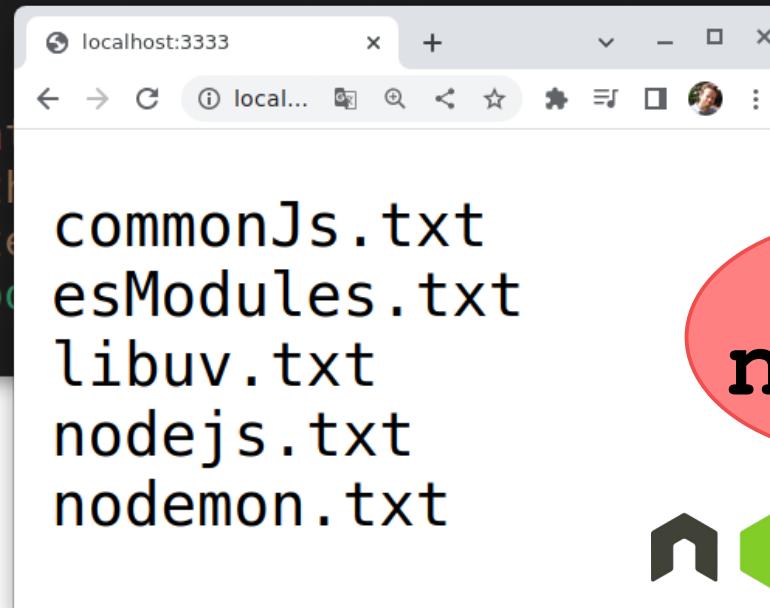
Exercício I – Parte 5

- Renomeie o arquivo `.env` para `.env.development`, e então crie um arquivo `.env.production` para o ambiente de produção. Programe o seu script para usar o arquivo correto durante a execução

```
npm start ./public ~/d/pw
david@coyote ~/dev/pw $ npm start ./public

> aulas@1.0.0 start
> NODE_ENV=development nodemon index.js ./public

[nodemon] 2.0.22
[nodemon] to restart at any time, enter `nodemon --restart`
[nodemon] watching path...
[nodemon] watching extensions...
[nodemon] starting `node ./index.js`
```



Módulos do NodeJS

- Também é possível criar seus próprios módulos

```
/* Módulo str_helper.js */

function upper (str) {
  return str.toUpperCase();
}

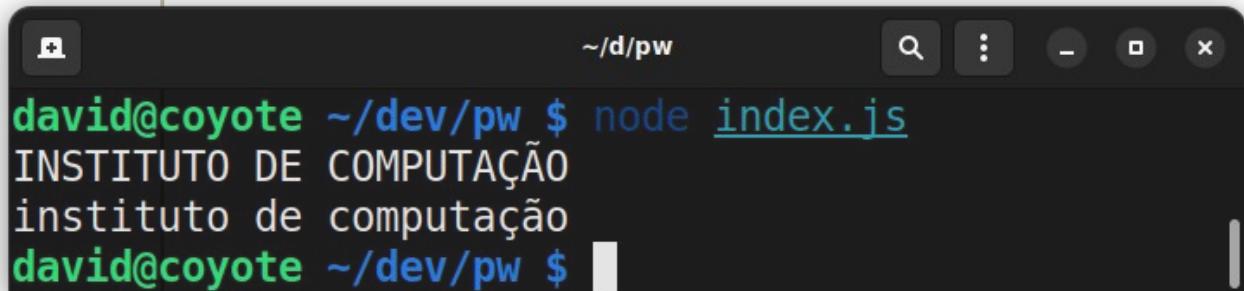
function lower (str) {
  return str.toLowerCase();
}

module.exports = {
  upper:upper,
  lower:lower
};
```

```
/* Arquivo index.js */

const strHelper = require('./str_helper')
let icomp = "Instituto de Computação";

console.log(strHelper.upper(icomp));
console.log(strHelper.lower(icomp));
```



A terminal window with a dark background and light-colored text. The prompt is 'david@coyote ~/dev/pw \$'. The user runs 'node index.js' and the terminal shows the output of the module: 'INSTITUTO DE COMPUTAÇÃO' on one line and 'instituto de computação' on the next. The terminal window has standard OS X-style controls at the top.

```
david@coyote ~/dev/pw $ node index.js
INSTITUTO DE COMPUTAÇÃO
instituto de computação
david@coyote ~/dev/pw $
```

Módulos do NodeJS

- Também é possível criar seus próprios módulos

```
/* Módulo str_helper.js */

function upper (str) {
  return str.toUpperCase();
}

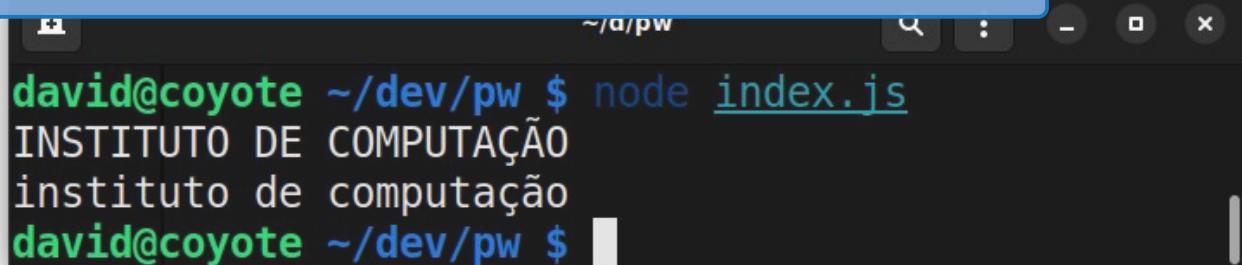
function lower (str) {
  return str.toLowerCase();
}

module.exports = {
  upper:upper,
  lower:lower
};
```

```
/* Arquivo index.js */
```

```
const strHelper = require('./str_helper')
let icomp = "Instituto de Computação";
```

Note que é preciso usar o objeto **module.exports** para definir que variáveis e funções poderão ser acessadas ou executadas por quem importar o módulo.



```
david@coyote ~/dev/pw $ node index.js
INSTITUTO DE COMPUTAÇÃO
instituto de computação
david@coyote ~/dev/pw $
```

Módulos do NodeJS

- Em linguagens como PHP e Ruby, todas as variáveis e funções declaradas nas bibliotecas passam a pertencer ao escopo global das aplicações que as importam

The image shows a code editor interface with two files:

- index.php**:

```
<?php
include 'biblioteca.php';
// imprime o quadrado de 2
echo quadrado(2);
```
- biblioteca.php**:

```
<?php
function quadrado ($valor) {
    return $valor * $valor;
}
```

To the right, a browser window displays the result of running index.php, showing the output "4".

At the bottom of the code editor, there are status icons and text: "0 0" (warning count), "Ln 4, Col 20", "Spaces: 2", "UTF-8", "LF", "PHP", and a smiley face icon.

On the far right, there are logos for Node.js, NPM, and a green hexagon icon.

Módulos do NodeJS

- Em linguagens como PHP e Ruby, todas as variáveis e funções declaradas nas bibliotecas passam a pertencer ao escopo global das aplicações que as importam

O Node.js, por outro lado, permite que os desenvolvedores de bibliotecas informem, através do objeto **module.exports**, quais funções e variáveis poderão ser importadas pelas aplicações

```
5 echo quadrado(2);  
6  
biboteca.php ✘  
1 <?php  
2  
3 function quadrado ($valor) {  
4     return $valor * $valor;  
5 }  
4
```

Ln 4, Col 20 Spaces: 2 UTF-8 LF PHP ☺ 🔔

Exercício I – Parte 6

- Adapte a aplicação desenvolvida até este momento, de forma que seja adicionado um link para cada arquivo do diretório informado. O conteúdo HTML não precisa ter head nem body, apenas os links (vide imagem). Cada link deverá ser gerado por uma função **createLink**, presente um módulo **utils.js** separado.

```
function createLink(filename) {  
    return `<a href="/${filename}">${filename}</a><br>\n`;  
}
```

The screenshot shows two browser windows. The left window, titled 'localhost:3333', displays a list of files: commonJs.txt, esModules.txt, libuv.txt, nodejs.txt, and nodemon.txt, each underlined as a link. The right window, titled 'view-source:localhost:3333', shows the source code of the page. It consists of a table with file names as rows. A red oval highlights the word 'node1' in the last row, which corresponds to the 'nodejs.txt' file. The 'github' logo is also visible in the top right of the source code window.

| | | |
|---|--|--------|
| 1 | commonJs.t | github |
| 2 | esModule | |
| 3 | libuv.txt | |
| 4 | nodejs.txt | |
| 5 | nodemon.txt | node1 |
| 6 | | |

Continua...

Exercício I – Parte 6

- A listagem de links deverá ser mostrada quando o usuário acessa o / da aplicação. Ao clicar em um link, o usuário poderá ver o conteúdo do arquivo. Use a propriedade url do objeto req para identificar a url do browser. Adicione um link voltar nas páginas de conteúdo, para que o usuário possa voltar para a página de links.

The screenshot shows a web application interface. On the left, a sidebar displays a list of files with blue underlined links: commonJs.txt, esModules.txt, libuv.txt, nodejs.txt, and nodemon.txt. A mouse cursor arrow points towards the 'nodejs.txt' link. On the right, the main content area shows the file 'nodejs.txt' has been selected. The page title is 'localhost:3333/nodejs.txt'. The content of the file is displayed as:

Voltar

De acordo com sua definição oficial, o Node é um runtime, que nada mais é do que um conjunto de códigos, API's, ou seja, são bibliotecas responsáveis pelo tempo de execução (é o que faz o seu programa que funciona como um interpretador de JavaScript fora do ambiente do navegador web).

A red oval on the right side contains the text 'github' above 'node1'.

CommonJs vs ES modules

- O mecanismo de modularização do Node.js, que adota **require** e **module.exports**, é conhecido como **CommonJs**
- O CommonJs foi criado para o Node.js em 2009, pois naquela época o EcmaScript não suportava a criação de módulos

```
// Arquivo util.js
```

```
module.exports.add = function(a, b) {  
    return a + b;  
}
```

```
const { add } = require('./util')  
  
console.log(add(5, 5)) // 10
```



CommonJs vs ES modules

- Com o ES6 (2015), o EcmaScript passou a ter um mecanismo de modulização conhecido como ES modules

```
// Arquivo util.mjs
```

```
export function add(a, b) {  
    return a + b;  
}
```

```
import { add } from './util.mjs'  
console.log(add(5, 5)) // 10
```

Como o NodeJs continua adotando o CommonJs por padrão, uma das formas de notificar o NodeJs de que queremos usar o ES Modules é adotando a extensão **mjs**

CommonJs vs ES modules

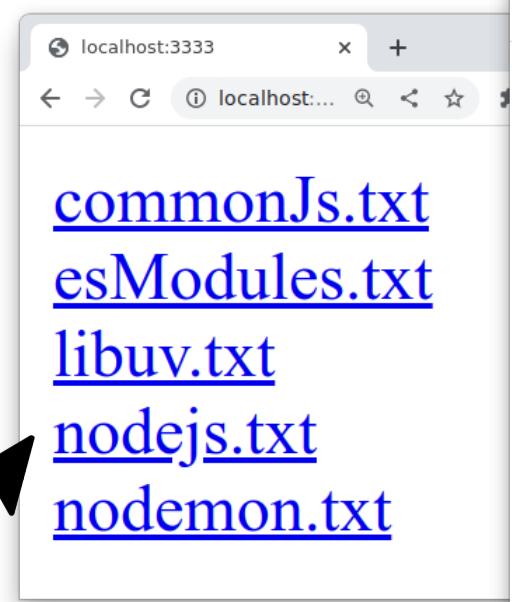
- Outra maneira de habilitar módulos ES é adicionando um campo "type: module" dentro do arquivo package.json
 - Com essa inclusão, não será preciso alterar os arquivos para a extensão **mjs**

```
{  
  "name": "my-app",  
  "version": "1.0.0",  
  "type": "module",  
  // ...  
}
```



Exercício II

- Refaça o exercício anterior usando ES modules.



The browser window is titled "localhost:3333/nodejs.txt". The content of the file is:

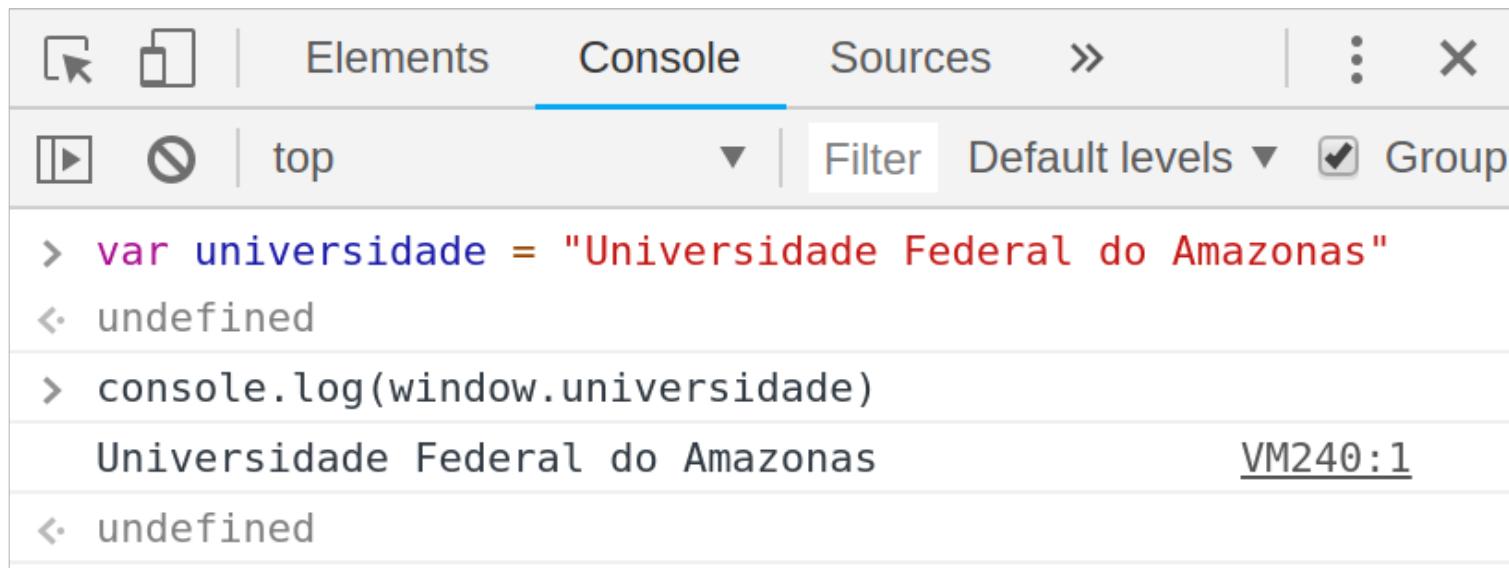
[Voltar](#)

De acordo com sua definição oficial, o Node é um runtime, que nada mais é do que um conjunto de códigos, API's, ou seja, são bibliotecas responsáveis pelo tempo de execução (é o que faz o seu programa rodar) que funciona como um interpretador de JavaScript fora do ambiente do navegador.

github
node2

O escopo Global

- No desenvolvimento frontend, o **escopo global** é definido pelo **objeto window** e todas as variáveis declaradas com **var**, **let** e **const** fora das funções pertencem a esse escopo global



The screenshot shows the developer tools console tab selected in the top navigation bar. The console interface includes icons for selection, copy, and paste, followed by tabs for Elements, Console, and Sources. Below the tabs, there are buttons for play/pause, stop, and a dropdown menu set to 'top'. To the right are 'Filter' and 'Default levels' buttons, and a checked 'Group' checkbox. The main area displays the following JavaScript interaction:

```
> var universidade = "Universidade Federal do Amazonas"
< undefined
> console.log(window.universidade)
Universidade Federal do Amazonas
< undefined
```

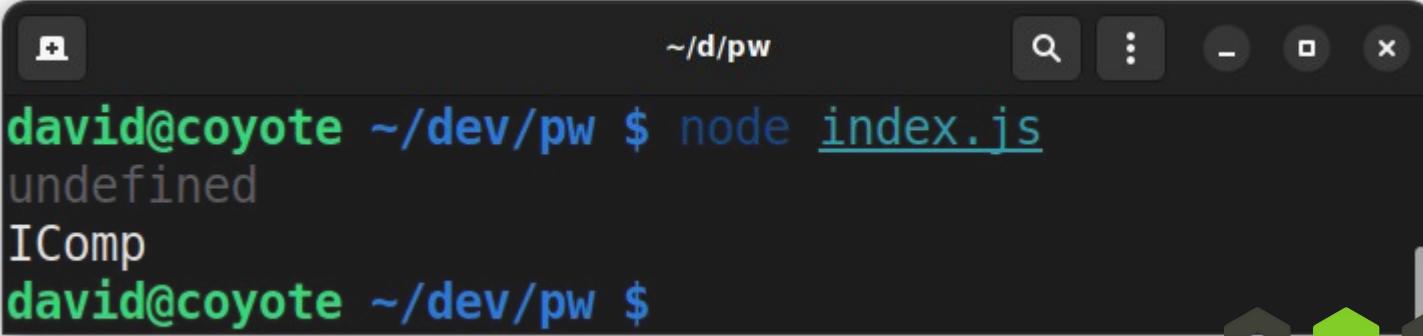
The output 'Universidade Federal do Amazonas' is timestamped as VM240:1.

O escopo Global

- No Node.js, o objeto window não existe e o escopo global é definido pelo objeto **global**
- Diferente dos browsers, as variáveis declaradas com **const** e **let** fora das funções não são armazenadas no objeto **global**

```
const universidade = "UFAM"
console.log(global.universidade)

instituto = "IComp"
console.log(global.instituto)
```



A screenshot of a terminal window titled '~/d/pw'. The command 'node index.js' is run, and the output shows two lines of text: 'undefined' and 'IComp', demonstrating that variables declared with const or let outside functions are not stored in the global object.

```
david@coyote ~/dev/pw $ node index.js
undefined
IComp
david@coyote ~/dev/pw $
```

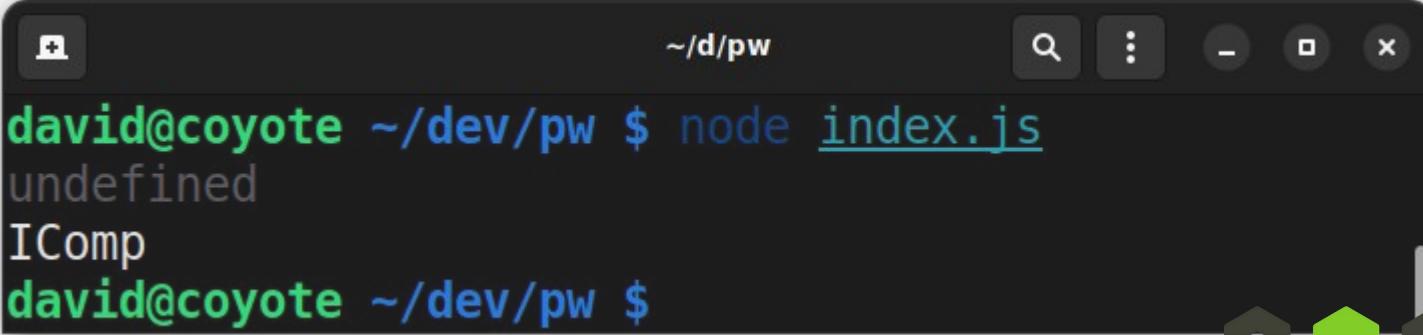


O escopo Global

- No Node.js, o objeto window não existe e o escopo global é definido pelo objeto **global**
- Diferente dos browsers, as variáveis declaradas com **const** e **let** fora das funções não são armazenadas no objeto **global**

No Node.js, as variáveis declaradas com **let** ou **const** dentro de um módulo (mesmo no módulo principal) são privadas daquele módulo, e não pertencem ao escopo global

```
INSTITUTO IComp  
console.log(global.instituto)
```



A screenshot of a terminal window titled 'david@coyote ~/dev/pw'. The window shows the command 'node index.js' being run, followed by the output 'undefined IComp'. The terminal has a dark theme with light-colored text.

```
david@coyote ~/dev/pw $ node index.js
undefined
IComp
david@coyote ~/dev/pw $
```



O escopo Global

- No Node.js, o objeto window não existe e o escopo global é definido pelo objeto **global**
- Diferente dos browsers, as variáveis declaradas com **const** e **let** fora das funções não são armazenadas no objeto **global**

No Node.js, as variáveis declaradas com **let** ou **const** dentro de um módulo (mesmo no módulo principal) são privadas daquele módulo, e não pertencem ao escopo global

INSTALAR
console

Para acessar uma variável ou função de um módulo, é necessário que esta variável ou função seja exportada através do objeto **module.exports**

```
david@coyote ~/dev/pw $ node index.js
undefined
IComp
david@coyote ~/dev/pw $
```



O escopo Global

- As propriedades definidas no objeto global podem ser acessadas sem o uso de **module.exports**
- No entanto, por razões óbvias, essa prática deve ser evitada

```
/* Módulo univ.js */
```

```
universidade = "UFAM"
```

```
/* Arquivo index.js */
```

```
const g = require("./univ");
console.log(universidade);
```



A screenshot of a terminal window titled 'david@coyote ~/dev/pw'. The window shows the command 'node index.js' being run, followed by the output 'UFAM'.

```
david@coyote ~/dev/pw $ node index.js
UFAM
david@coyote ~/dev/pw $
```

Singleton

- **Singleton** é um **Design Pattern** onde classes e módulos podem ser instanciados apenas uma única vez
- Essa única instância será acessível em qualquer parte de seu programa

```
/* Módulo counter.js */  
  
let counter = 0  
  
function inc () {  
    return ++counter;  
}  
  
module.exports = { inc }
```

```
/* Arquivo index.js */  
  
const c1 = require("./counter");  
const c2 = require("./counter");  
  
console.log(c1.inc())  
console.log(c2.inc())  
console.log(c1.inc())  
console.log(c2.inc())
```



Singleton

- Singleton é um Design Pattern onde classes e módulos podem ser instanciados apenas uma única vez
- Essa única instância é acessível em qualquer parte de seu programa

```
/* Módulo counter.js */
let counter = 0

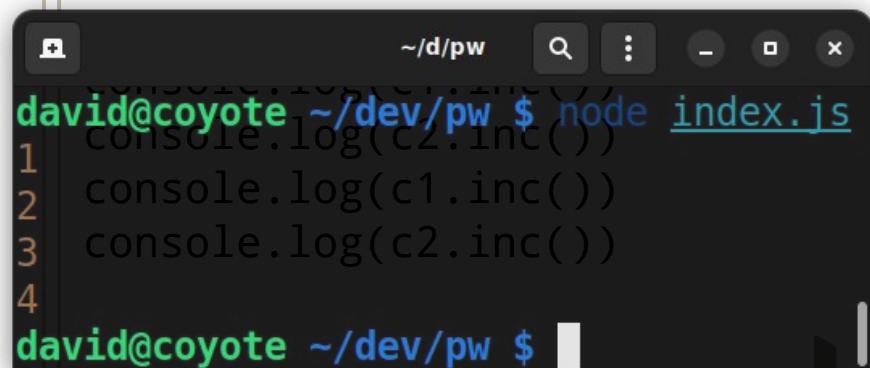
function inc () {
    return ++counter;
}

module.exports = { inc }
```

c1 e c2 são
uma única
instância

```
/* Arquivo index.js */
```

```
const c1 = require("./counter");
const c2 = require("./counter");
```



A terminal window titled 'david@coyote ~/dev/pw \$' shows the command 'node index.js' being run. The output displays three consecutive increments of the counter variable, starting from 1 and ending at 3.

```
david@coyote ~/dev/pw $ node index.js
1
2
3
```



Usando index.js

- Quando um módulo contém vários arquivos, pode-se agrupar todas as exportações desse módulo em único arquivo **index.js**

```
src
└── util
    ├── index.js
    ├── add.js
    └── sub.js
└── myapp.js
```

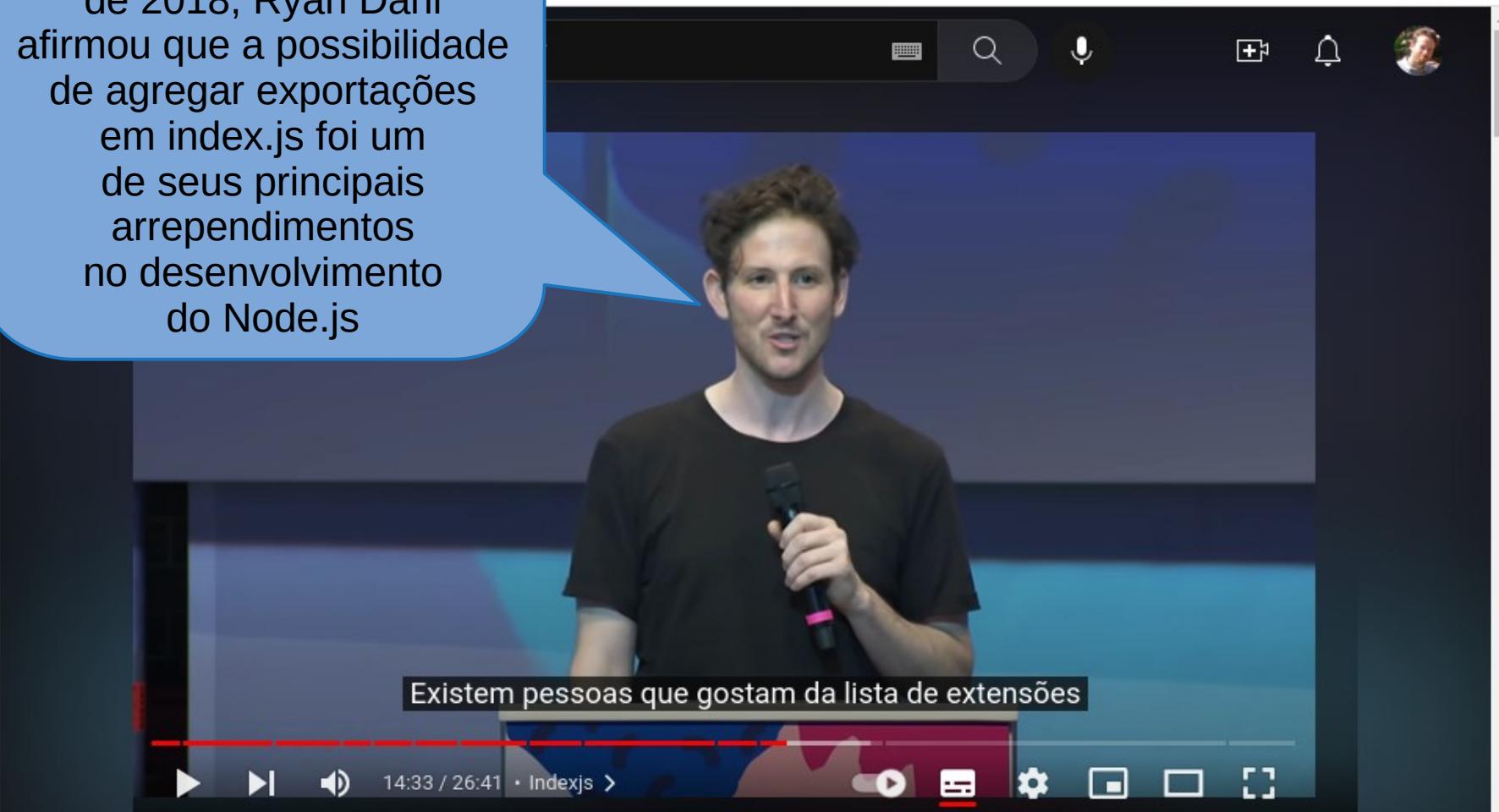
Nesse exemplo, a pasta **util** contém a implementação de um conjunto de componentes

```
// util/index.js
module.exports = {
  add: require('./add.js'),
  sub: require('./sub.js')
}
```

```
// myapp.js
const { add, sub } = require('./util')
```



Na JSConf Europeia de 2018, Ryan Dahl afirmou que a possibilidade de agregar exportações em index.js foi um de seus principais arrependimentos no desenvolvimento do Node.js



10 coisas que lamento sobre Node.js - Ryan Dahl - JSConf EU



JSConf

260 mil inscritos

Inscrever-se

Gostei



Compartilhar



Deno, The next-generation [x](#) + [deno.com](#)

 **Deno** Modules Docs Deploy Community Ctrl K

Deno v1.36.4 [Free & open source](#)

Next-generation JavaScript Runtime

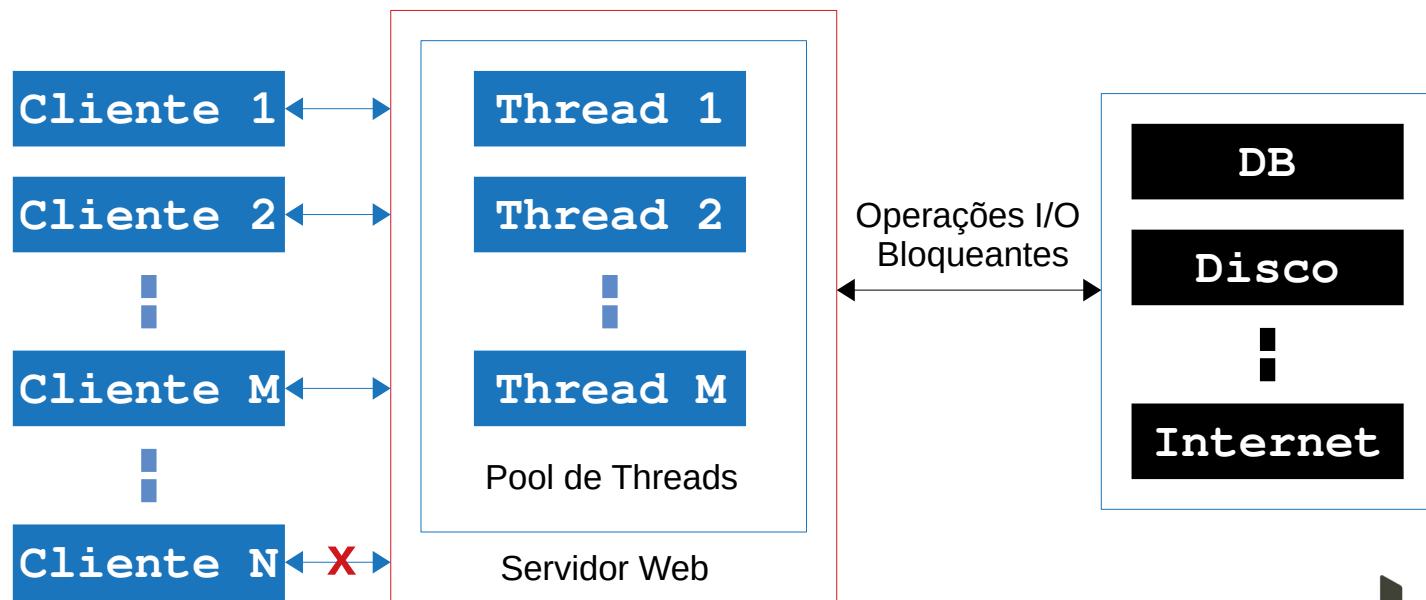
- ✓ Secure by default
- ✓ Native support for TypeScript and JSX
- ✓ Testing, linting, formatting, and more out of the box
- ✓ High performance async I/O with Rust and Tokio
- ✓ Backwards compatible with Node.js and npm

[Install Now](#) [Docs →](#)



Single Thread Event Loop

- Linguagens como PHP, Python e Java recorrem ao **multi-threading** para lidar com aplicações multi-usuários
 - Isto é, cria-se uma nova **thread** para atender cada novo usuário ou requisição



Single Thread Event Loop

- Nessa abordagem multi-thread tradicional, todas as operações de I/O são **bloqueantes**
 - Por exemplo, no programa Python abaixo, todo o código deve esperar a leitura de arquivo que ocorre na linha 3

```
#!/usr/bin/python3

dias = open("semana.txt", "r+")
conteudo = dias.read()

print (conteudo)
print ("continua...")

dias.close()
```

Operação
de I/O
bloqueante



A terminal window titled 'david@coyote ~/dev/python \$' shows the execution of a Python script named 'semana.py'. The script reads the contents of a file named 'semana.txt' and prints it to the console. The output is as follows:

```
david@coyote ~/dev/python $ python3 semana.py
Domingo
Segunda-feira
Terça-feira
Quarta-feira
Quinta-feira
Sexta-feira
Sábado
continua...
david@coyote ~/dev/python $
```

The word 'continua...' is printed in red, indicating it is part of the original code. Below the terminal window, there are icons for Node.js and JavaScript.

Single Thread Event Loop

- Nessa abordagem multi-thread tradicional, todas as operações de I/O são **bloqueantes**
 - Por exemplo, no programa Python abaixo, todo o código deve esperar a leitura de arquivo que ocorre na linha 3

```
#!/usr/bin/python3

dias = open("semana.txt", "r+")
conteudo = dias.read()

print (conteudo)
```

Operação
de I/O
bloqueante

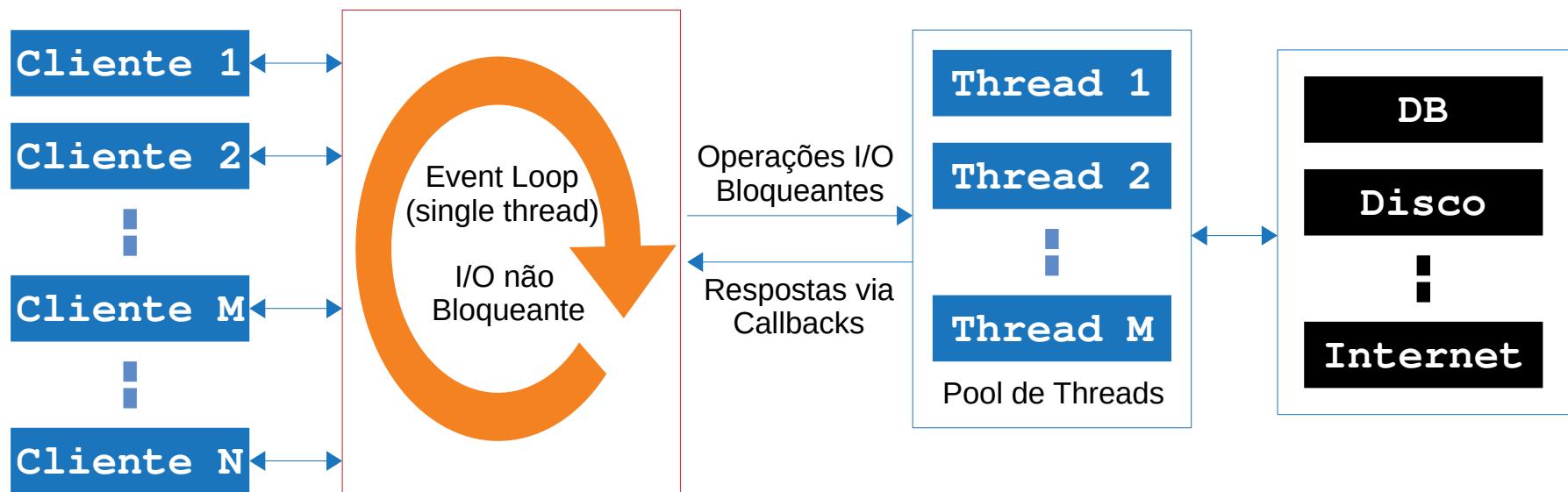
A screenshot of a terminal window titled "david@coyote ~/dev/python \$". The command "python3 semana.py" is being run. The output shows the word "Dominao" followed by a continuation ellipsis "...". Below the terminal is a decorative footer featuring icons for Node.js, Python, and JavaScript.

```
david@coyote ~/dev/python $ python3 semana.py
Dominao ...
david@coyote ~/dev/python $
```

A abordagem bloqueante também é chamada de **codificação síncrona**, pois a execução de uma linha só ocorre após a execução das linhas anteriores, seguindo a sequência do código

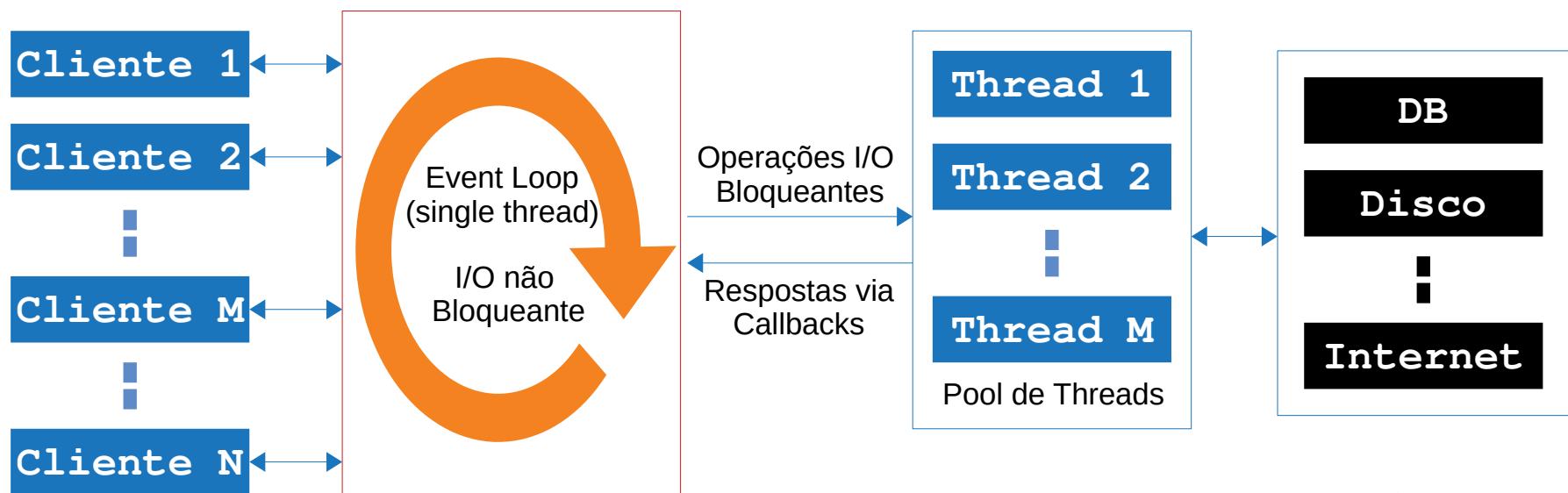
Single Thread Event Loop

- No Node.js, apenas uma thread responde por todas as requisições dos usuários – essa thread é chamada de **Event Loop**
 - Operações de I/O (acesso ao banco, leitura de arquivos, etc) são **assíncronas e não bloqueiam** a thread



Single Thread Event Loop

- No Node.js, apenas uma thread responde por todas as requisições dos clientes. Essa thread é chamada de Event Loop.
 - O Node.js é um ambiente de execução **single thread**, que no background usa múltiplas threads para executar códigos bloqueantes de I/O



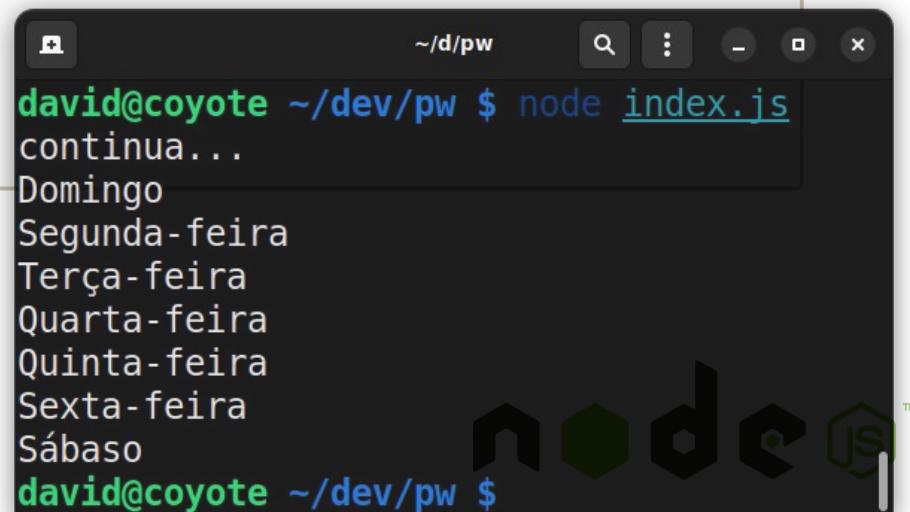
Single Thread Event Loop

- I/O não bloqueante permite que o **event loop** responda por várias requisições de usuários de forma bastante rápida
- Operações de I/O não bloqueantes provêem uma **função de callback** que é chamada quando a operação é completada

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

fs.readFile('semana.txt', 'utf8', function(err, conteudo) {
  if (err) throw new Error(err)
  console.log(conteudo);
});

console.log("continua...");
```



A screenshot of a terminal window titled 'david@coyote ~/dev/pw \$'. The command 'node index.js' was run, and the output shows the days of the week from Domingo to Sábado, followed by the text 'continua...'. The Node.js logo is visible at the bottom right of the terminal window.

```
david@coyote ~/dev/pw $ node index.js
continua...
Domingo
Segunda-feira
Terça-feira
Quarta-feira
Quinta-feira
Sexta-feira
Sábado
david@coyote ~/dev/pw $
```

Single Thread Event Loop

- Alguns métodos de I/O do Node.js também possuem versões síncronas (bloqueantes)
 - Esses métodos em geral terminam com o sufixo Sync

```
const fs = require('fs');
const conteudo = fs.readFileSync('semana.txt');
console.log(conteudo.toString());
console.log("continua...");
```



A terminal window titled 'david@coyote ~/dev/pw \$' displays the command 'node indexSync.js'. The output shows the days of the week from Domingo to Sábaso, followed by the text 'continua...'. The terminal has a dark background with light-colored text and icons.

```
david@coyote ~/dev/pw $ node indexSync.js
Domingo
Segunda-feira
Terça-feira
Quarta-feira
Quinta-feira
Sexta-feira
Sábaso
continua...
david@coyote ~/dev/pw $
```



Single Thread Event Loop

- Alguns métodos de I/O do Node.js também possuem versões síncronas (bloqueantes)
 - Esses métodos em geral terminam com o sufixo Sync

```
const fs = require('fs');
const conteudo = fs.readFileSync('semana.txt');
console.log(conteudo.toString());
console.log("continua...");
```

No entanto, o uso dessas funções deve ser evitado em aplicações multi-usuário, pois elas bloqueiam o **Event Loop** para outros usuários

```
Terça-feira
Quarta-feira
Quinta-feira
Sexta-feira
Sábado
continua...
david@coyote ~/dev/pw $
```



Single Thread Event Loop

- Para mostrar que o Node.js é um **ambiente de execução single thread**, podemos usar um algoritmo de uso intenso da CPU

```
const isPrime = require("is-prime-number");
const http = require("http");

let counter = 0;

http.createServer((req, res) => {
  console.log(`Counter ${counter}`);
  let number = 0;
  let qtdPrimes = 0;

  while(qtdPrimes<1_000_000) {
    if (isPrime(++number)) qtdPrimes++;
  }

  res.end(`Primo ${number}`)
}).listen(3000);
```



Single Thread Event Loop

- Para mostrar que o Node.js é um **ambiente de execução single thread**, podemos usar um algoritmo de uso intenso da CPU

```
const isPrime = require("is-prime-number");
const http = require("http");
```

```
let counter = 0;
```

Ao acessar `http://localhost:3000` em duas abas do browser, conseguiremos perceberemos que a segunda requisição só inicia após o término da primeira.

```
let number = 2;
```

```
let qtdPrimes = 0;
```

```
while(qtdPrimes<1_000_000) {
```

```
    if (isPrime(++number)) qtdPrimes++;
```

```
}
```

```
res.end(`Primo ${number}`)
```

```
).listen(3000);
```



Single Thread Event Loop

- Para mostrar que o Node.js é um **ambiente de execução single thread**, podemos usar um algoritmo de uso intenso da CPU

```
const isPrime = require("is-prime-number");
const http = require("http");
```

```
let counter = 0;
```

Ao acessar <http://localhost:3000> em duas abas do browser

comparando com o exemplo anterior, Desta forma, o exemplo demonstra que, por padrão, o Node.js possui apenas uma thread – **event loop** – sendo executada em apenas um núcleo do processador.

```
while(qtdPrimes<1_000_000) {
  if (isPrime(++number)) qtdPrimes++;
}

res.end(`Primo ${number}`)

}).listen(3000);
```



Single Thread Event Loop

- No entanto, o **core module cluster** permite o uso de mais de um núcleo do processador em aplicações de uso intensivo da CPU

```
const cluster = require("cluster");
const http = require("http");
const numCPUs = require("os").cpus().length;

if (cluster.isMaster) {
    console.log("Este é o processo Master");

    for (let i = 0; i < numCPUs; i++) cluster.fork();
} else {
    http.createServer((req, res) => {
        res.end(`Eu sou um worker #${cluster.worker.id}`);
    }).listen(3344);
}
```



Single Thread Event Loop

- No entanto, o **core module cluster** permite o uso de mais de um núcleo do processador em aplicações de uso intensivo da CPU

```
const cluster = require("cluster");
const http = require("http");
const numCPUs = require("os").cpus().length;

if (cluster.isMaster) {
    O comando cluster.fork() cria um novo processo com seu
    próprio event loop, que compartilha as portas do servidor
    com os demais processos gerados pelo módulo cluster
}

http.createServer((req, res) => {
    res.end(`Eu sou um worker #${cluster.worker.id}`);
}).listen(3344);
}
```



Single Thread Event Loop

- Note que o módulo cluster cria novos **processos** – e não **threads** – cada um com seu próprio event loop
- No entanto, a partir da versão 12, o Node.js passou a suportar o uso de threads através do módulo **worker_threads**

```
const {  
  isMainThread, Worker, parentPort  
} = require('worker_threads');  
  
if (isMainThread) {  
  const worker = new Worker(__filename);  
  worker.on('message', (msg) => { console.log(msg); });  
} else {  
  parentPort.postMessage('Hello world!');  
}
```

Código executado na thread principal

Envia uma mensagem para a thread principal



Callback Hell

- Para lidar com a assincronicidade do JavaScript, é possível nos depararmos com longas cadeias de callbacks

```
request(url1, function (error1, n1) {  
    request(url2, function (error2, n2) {  
        request(url3, function (error3, n3) {  
            request(url4, function (error4, n4) {  
                request(url5, function (error5, n5) {  
                    request(url6, function (error6, n6) {  
                        processa(n1, n2, n3, n4, n5, n6);  
                    });  
                });  
            });  
        });  
    });  
});  
});
```



Callback Hell

- Para lidar com a assincronicidade do JavaScript, é possível nos depararmos com longas cadeias de callbacks



```
request(url1, function (error1, n1) {
    request(url2, function (error2, n2) {
        request(url3, function (error3, n3) {
            request(url4, function (error4, n4) {
                request(url5, function (error5, n5) {
                    request(url6, function (error6, n6) {
                        processa(n1, n2, n3, n4, n5, n6);
                    });
                });
            });
        });
    });
});
```

Muitas vezes, esse tipo de código é chamado



Muitas vezes, esse tipo de código é chamado pela comunidade de **callback hell** ou **código hadouken**

Promises

- Uma solução para os callbacks hell são as **promises**, que são objetos usados para executar funções assíncronas
- Uma promise guarda um valor que pode estar disponível agora, no futuro ou nunca

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

const promise = new Promise((resolve, reject) => {
  fs.readFile("1.txt", "utf-8", (error, data) => {
    resolve(parseInt(data));
  });
});

promise.then((data) => {
  console.log(data)
});
```



A terminal window titled '~d/pw' showing the execution of a Node.js script. The command `node promise.js` is run, followed by the output of the script which logs the number 1 to the console.

```
david@coyote ~/dev/pw $ echo "1" > 1.txt
david@coyote ~/dev/pw $ node promise.js
1
david@coyote ~/dev/pw $
```



Promises

- É possível usar o **then** para dispor as promises em sequência, de forma a evitar que o código cresça para a direita

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

function readFile (filename) {
  return new Promise(function (resolve, reject) {
    fs.readFile(filename, function(error, data) {
      resolve(parseInt(data));
    });
  });
}

readFile('1.txt')
  .then(function(data1) {
    console.log(data1);
    return readFile('2.txt')
  })
  .then(function(data2) {
    console.log(data2);
  })
}
```



A terminal window titled 'david@coyote ~/dev/pw \$' shows the command 'node promise.js' being run. The output consists of two numbers, 1 and 2, each on a new line.

```
david@coyote ~/dev/pw $ node promise.js
1
2
```



Promises

- O método estático **Promise.all()** pode ser usado para aguardar a resolução de um conjunto de **promises**

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

const p1 = new Promise(function (resolve, reject) {
  fs.readFile('./1.txt', function(error, data) {
    resolve(parseInt(data));
  });
}

const p2 = new Promise(function (resolve, reject) {
  fs.readFile('./2.txt', function(error, data) {
    resolve(parseInt(data));
  });
}

Promise.all([p1, p2]).then(function([data1, data2]) {
  console.log(data1 + data2);
});
```

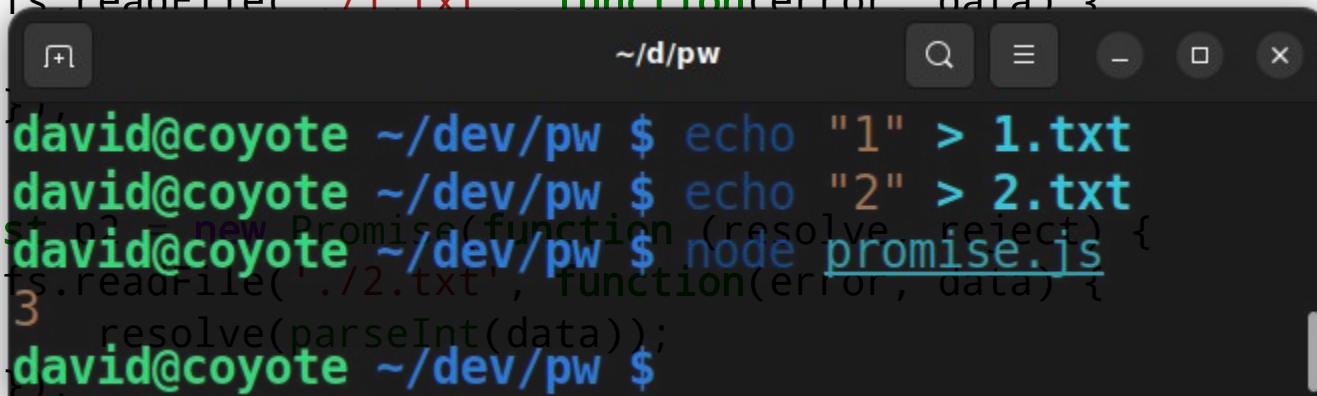


Promises

- O método estático **Promise.all()** pode ser usado para aguardar a resolução de um conjunto de **promises**

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

const p1 = new Promise(function (resolve, reject) {
  fs.readFile('1.txt', function(error, data) {
    if (error) {
      reject(error);
    } else {
      resolve(data);
    }
  });
  const p2 = new Promise(function (resolve, reject) {
    fs.readFile('2.txt', function(error, data) {
      if (error) {
        reject(error);
      } else {
        resolve(parseInt(data));
      }
    });
  });
  Promise.all([p1, p2]).then(function([data1, data2]) {
    console.log(data1 + data2);
  });
});
```



Promises

- O callback das promessas aceita os parâmetros **resolve** e **reject** – a função **resolve** deve ser chamada se não houver erros; caso contrário chama-se **reject**

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

function getUrl(url) {
    return promise = new Promise(function (resolve, reject) {
        request(url, function (error, response, body) {
            if (error) reject(error);
            else resolve(body);
        });
    });
}

getUrl('http://google.com')
    .then(function (body) {
        console.log(body);
    })
    .catch(function (error) {
        console.log(error);
    });
}
```



TM

Async e Await

- Uma alternativa ao método **Promisse.then()** são os modificadores **async** e **await**

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

function readFile (filename) {
    return new Promise(function (resolve, reject) {
        fs.readFile(filename, function(error, data) {
            resolve(parseInt(data));
        });
    });
}

async function calcularValor () {
    let valor1 = await readFile('1.txt');
    let valor2 = await readFile('2.txt');
    console.log(valor1 + valor2);
}

console.log('a');
calcularValor ();
console.log('b');
console.log('c');
```

Uma função declarada com **async** pode conter expressões **await**, que pausa a execução da função assíncrona



Async e Await

- Uma alternativa ao método **Promisse.then()** são os modificadores **async** e **await**

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

function readFile (filename) {
  return new Promise((resolve, reject) => {
    fs.readFile(filename, (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(parseInt(data));
      }
    });
  });
}

async function calcularValor () {
  let valor1 = await readFile('1.txt');
  let valor2 = await readFile('2.txt');
  console.log(`A soma é ${valor1 + valor2}`);
}

console.log('a');
calcularValor ();
console.log('b');
console.log('c');
```

A terminal window titled 'david@coyote ~/dev/pw \$' shows the execution of 'node asyncawait.js'. The output is:
david@coyote ~/dev/pw \$ node asyncawait.js
a
b
c
A soma é 10

Uma função declarada com **async** pode conter expressões **await**, que pausa a execução da função assíncrona

Async e Await

- O retorno de uma função **async** é sempre uma Promise

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

function readFile (filename) {
  return new Promise(function (resolve, reject) {
    fs.readFile(filename, function(error, data) {
      resolve(parseInt(data));
    });
  });
}

async function calculaValor() {
  const valor1 = await readFile("1.txt")
  const valor2 = await readFile("2.txt")
  return valor1 + valor2;
}

console.log('a')
console.log('b')
calculaValor().t
```

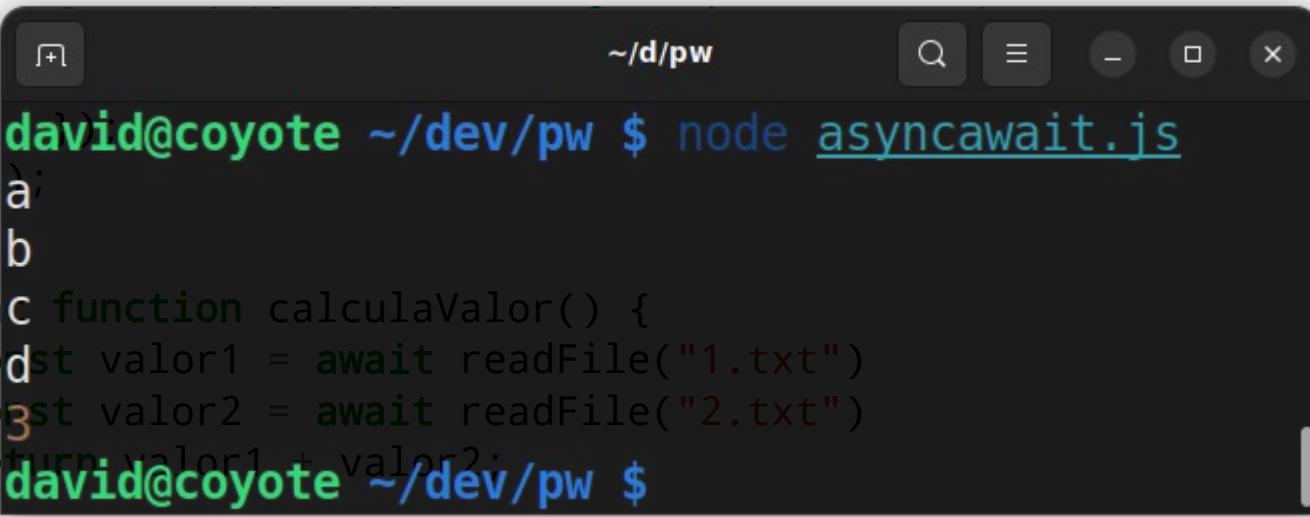


Async e Await

- O retorno de uma função **async** é sempre uma Promise

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

function readFile (filename) {
  return new Promise(function (resolve, reject) {
```



A screenshot of a terminal window titled '~d/pw'. The command entered is 'node asyncawait.js'. The output shows the execution of the code. It starts with the definition of a function 'readFile' which returns a Promise. Inside the Promise's constructor, there is a conditional block: if 'a' is truthy, it logs 'a'; if 'b' is truthy, it logs 'b'. Below this, an 'async' function 'calculaValor' is defined, which uses 'await' to call 'readFile' for files '1.txt' and '2.txt', then adds their results and returns the sum. Finally, the code ends with a closing brace for the 'calculaValor' function.

```
});}
} a;
} b;
async function calculaValor() {
  const valor1 = await readFile("1.txt");
  const valor2 = await readFile("2.txt");
  return valor1 + valor2;
}
```

```
console.log('a')
console.log('b')
calculaValor().then((data) => console.log(data))
console.log('c')
```



Módulo FS com Promises

- O **módulo fs** fornece um conjunto alternativo de métodos assíncronos que retornam objetos **Promise** e não usam callbacks
- Esse conjunto de métodos pode ser acessado por meio de **require('fs').promises**

```
const fsPromises = require('fs').promises;

const readFile = async (filePath) => {
  try {
    return await fsPromises.readFile(filePath, 'utf8');
  }
  catch(err) {
    console.log(err);
  }
}
```



Exercício III

Faça uma aplicação que é disponibilizada através de um servidor Node.js, onde o usuário informa um número x e a aplicação imprime x parágrafos Lorem Ipsum.

Note que a aplicação requer arquivos estáticos html, css e js. Use o objeto req (propriedade url) para identificar o arquivo requisitado.

Regras: é preciso usar nodemon, dotenv e fs promises (para leitura do conteúdo html, js e css que será enviado para o cliente)

