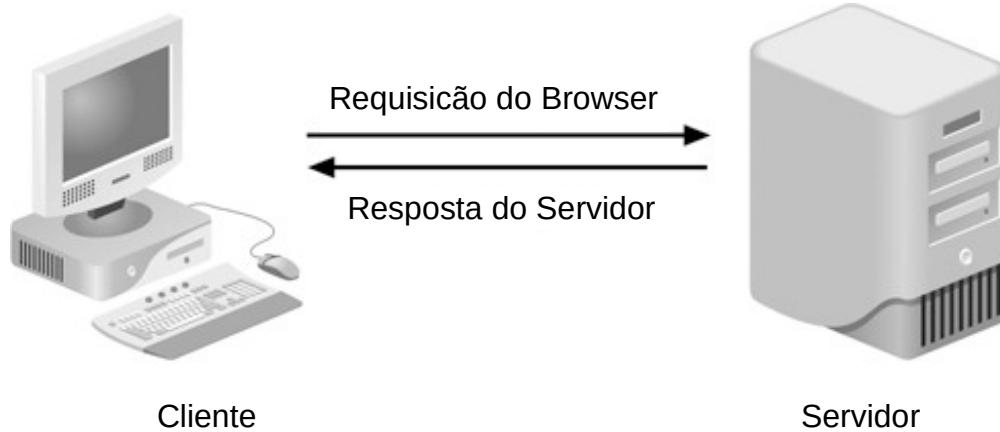




**Prof. David Fernandes de Oliveira  
Instituto de Computação  
UFAM**

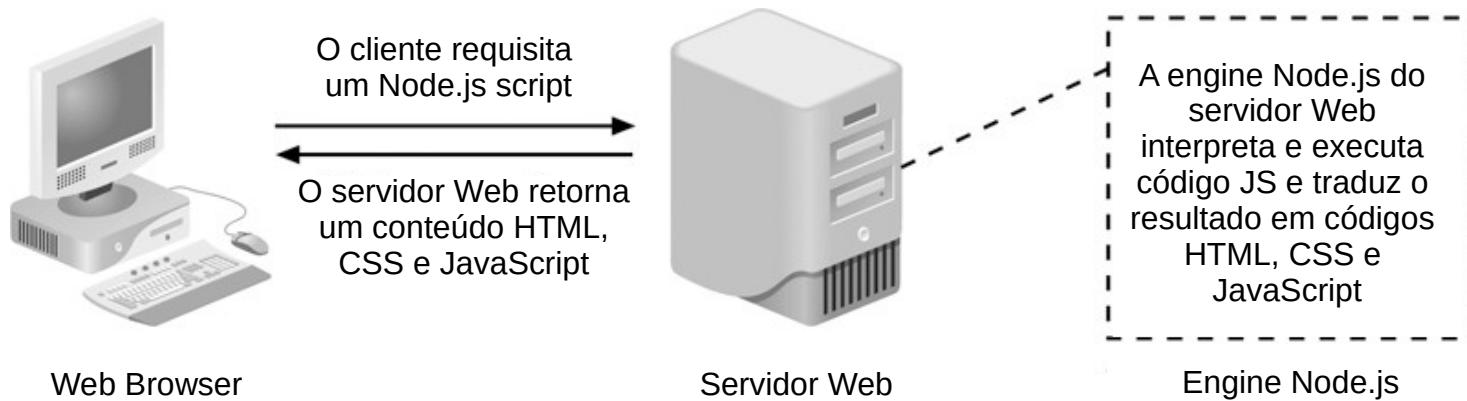
# Arquitetura Cliente/Servidor

- 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 interpretador 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**



# Node is: Javascript engine



## 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](#).

### 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
- Thread pool

### Table of Contents

- Welcome to the libuv documentation
  - Overview
  - Features
  - Documentation
  - Downloads
  - Installation

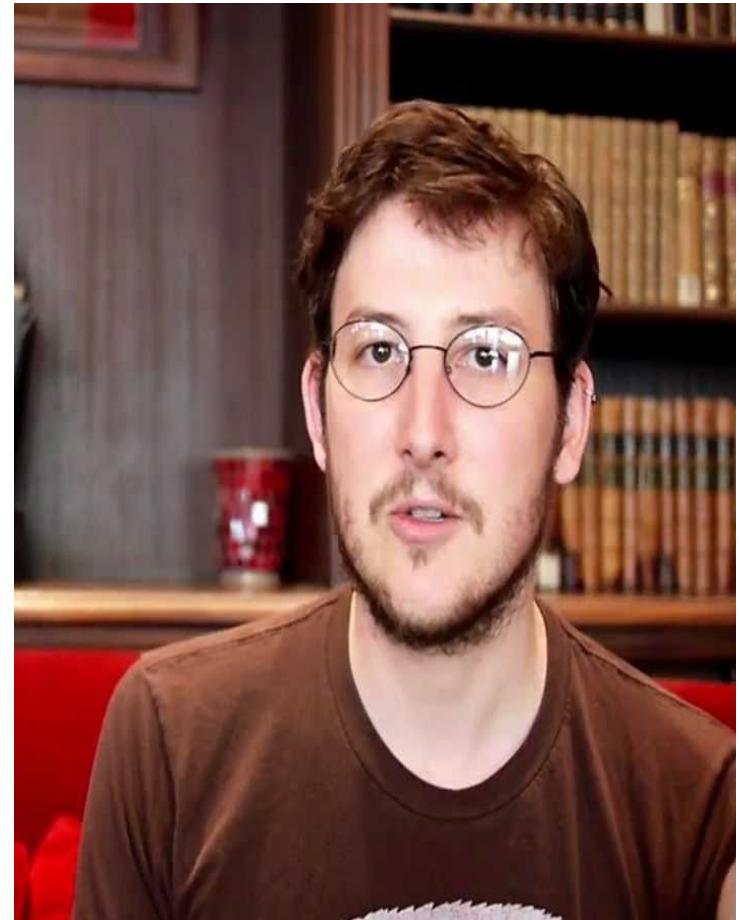
### Next topic

[Design overview](#)



# 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**

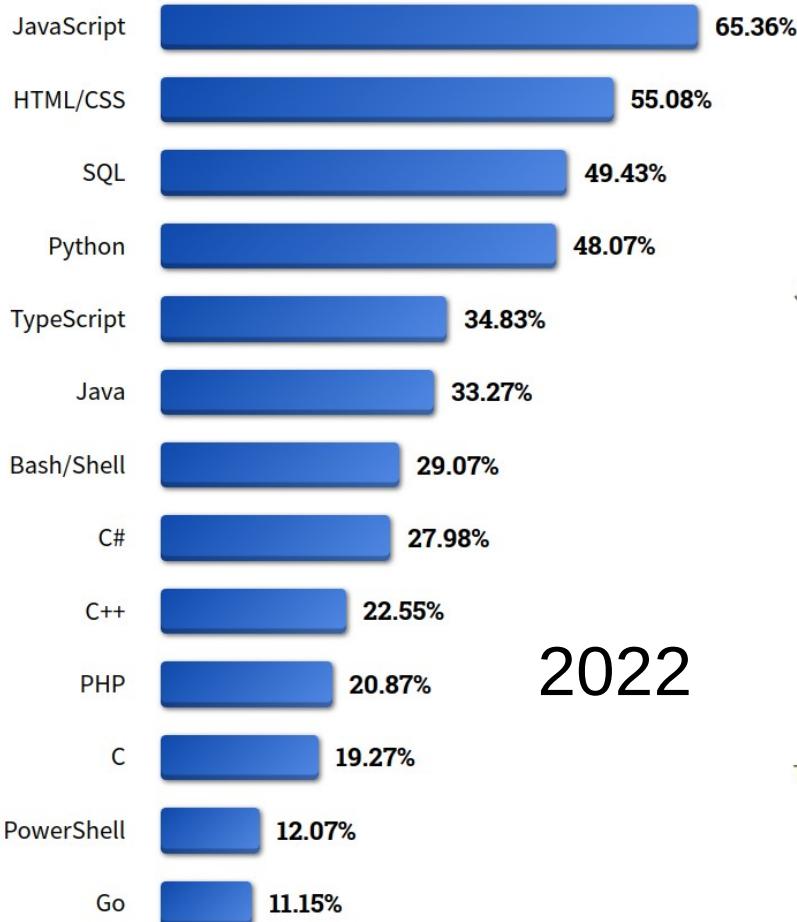


# Porque usar Node.js?

- **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

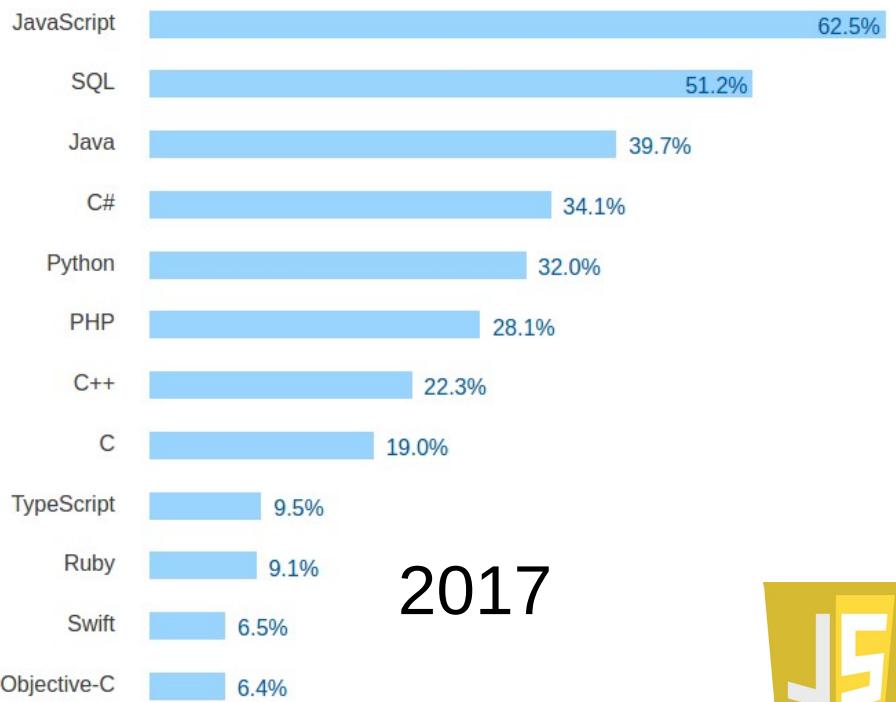


# StackOverflow Surveys



2022

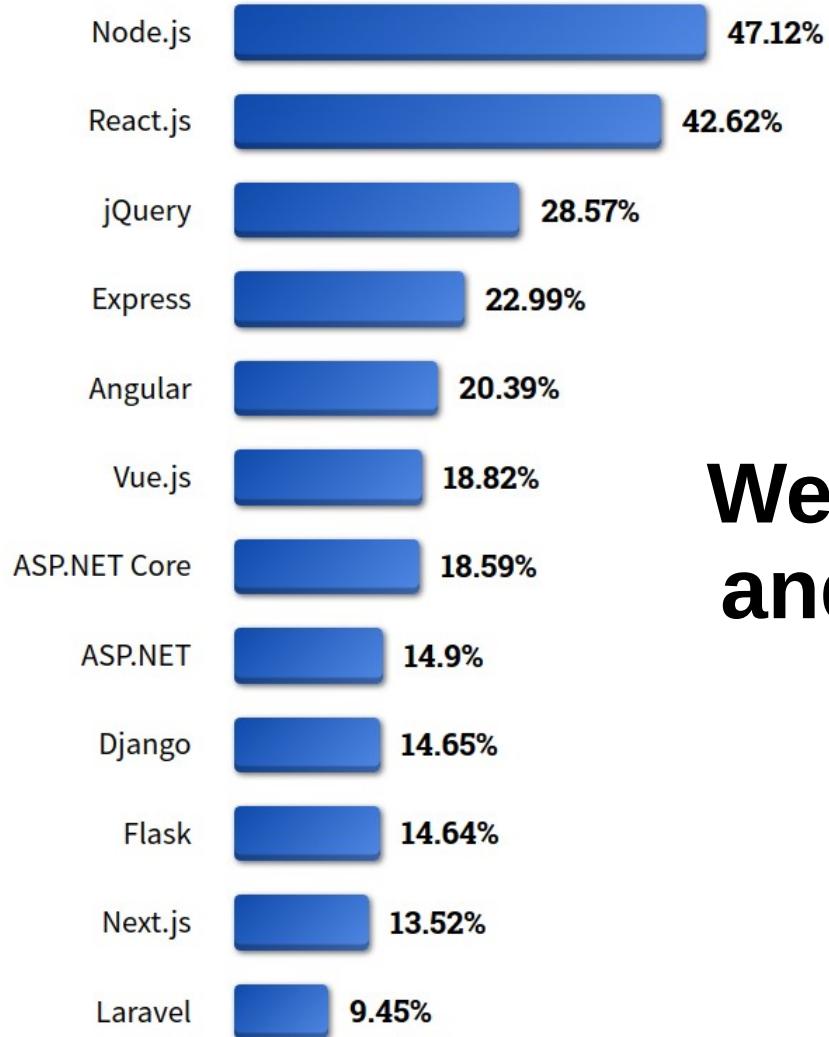
## Programming, Scripting, and Markup Languages



2017



# StackOverflow Surveys



**Web Frameworks  
and technologies**

2022



# 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 for Linux (x64)

18.16.0 LTS

Recommended For Most Users

20.1.0 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

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

## Download for Linux (x64)

O Node.js está disponível nos repositórios da maioria das distribuições Linux. Nas distribuições baseadas no debian, por exemplo, pode-se instalar o Node.js através do gerenciador de pacotes apt:

```
$ sudo apt install nodejs
```

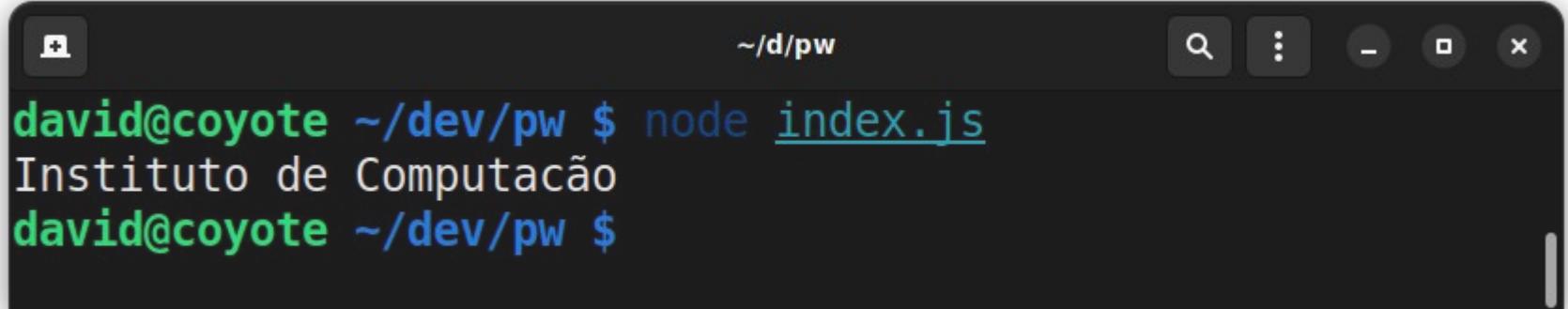
For information about supported releases, see the [release schedule](#).



# 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**) 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) {
    return console.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) {
    return console.error(err)
  }
});
```

Note 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')**



# 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) {
    return console.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 com através de uma técnica chamada **NameSpaces**. No Node.js, a função **rename** é um dos métodos do objeto retornado por **require('fs')**.



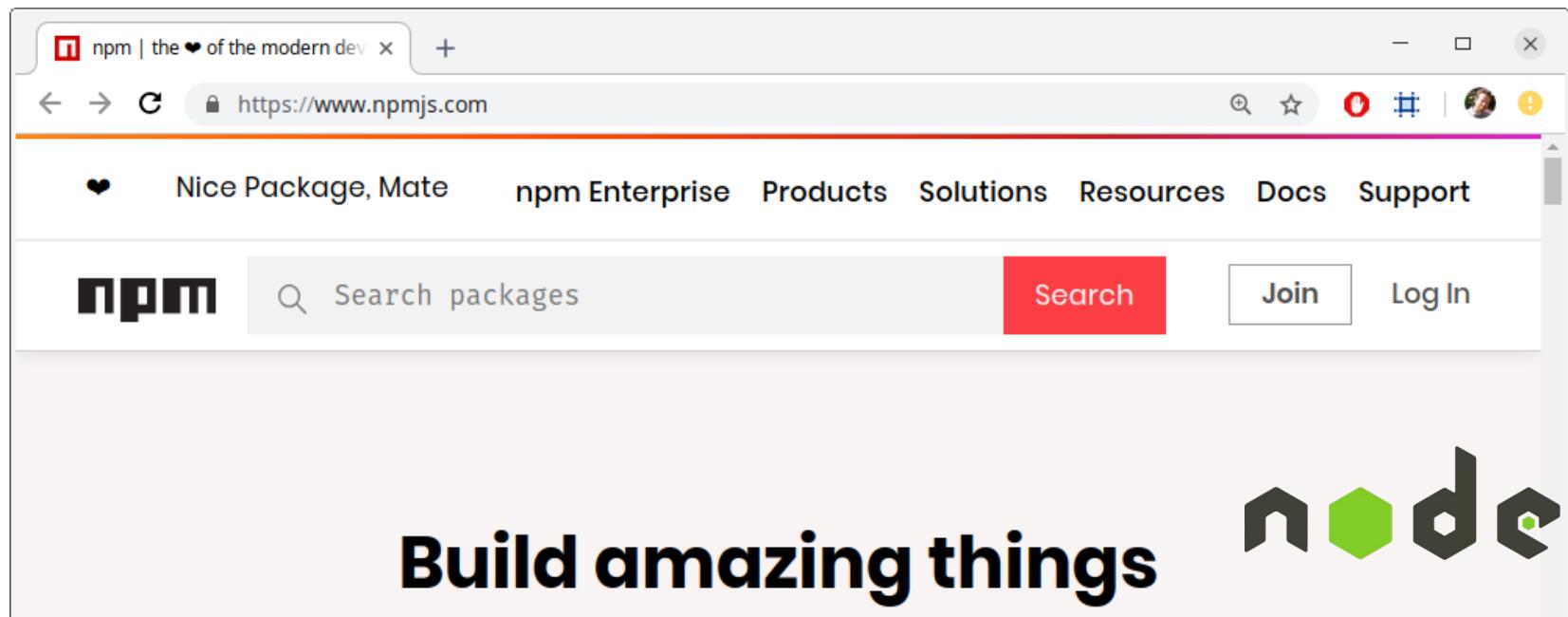
# O Módulo FS

- 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/>



# 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 screenshot of a web browser window displaying the npmjs.com homepage. The URL in the address bar is <https://www.npmjs.com>. The page features a large blue callout box 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 browser window, there is a "Build amazing things" slogan and the Node.js logo.

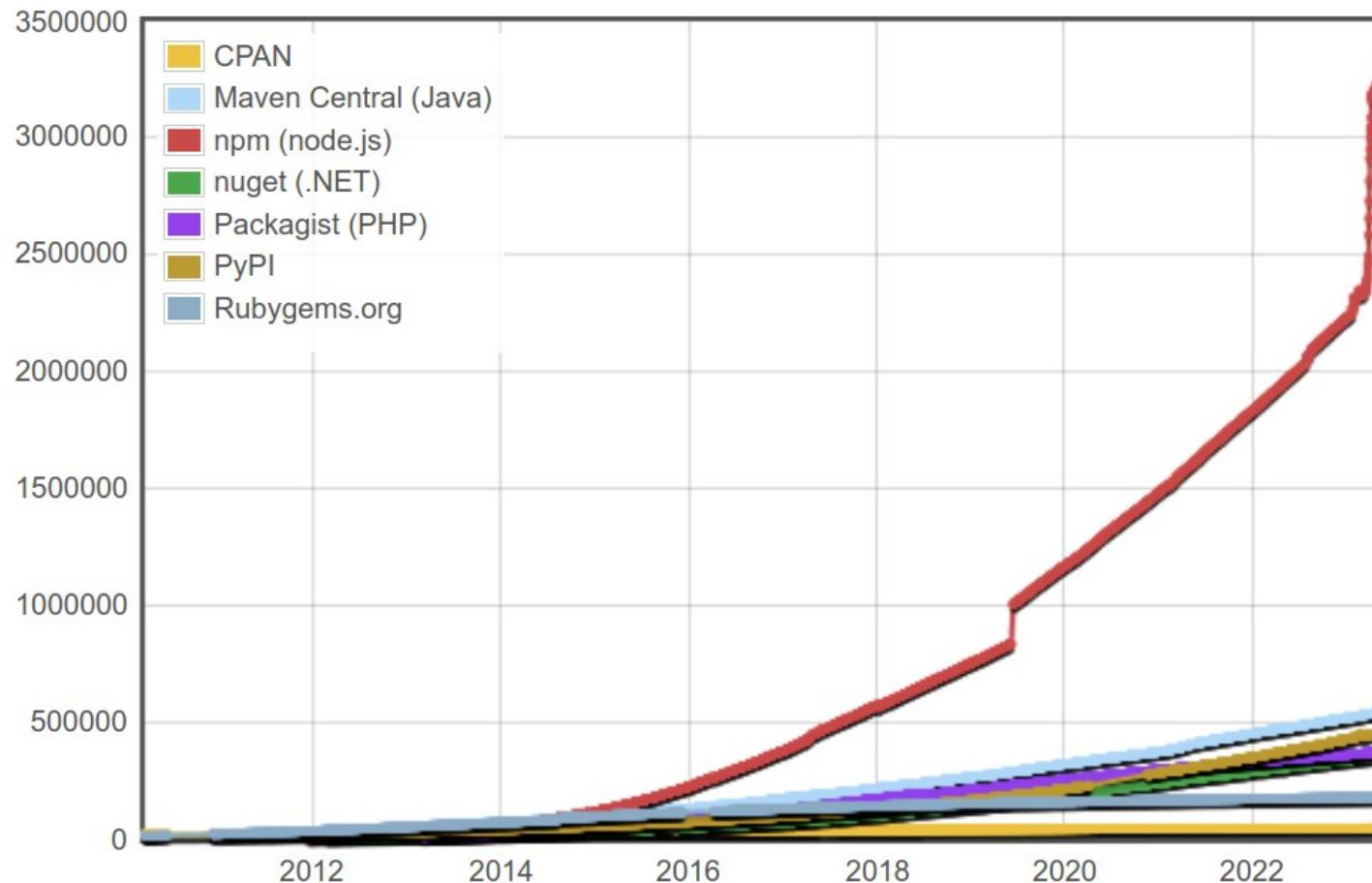
A comunidade Node.js ao redor do mundo é muito atuante e desenvolve inúmeros módulos de código aberto que são publicados neste repositório.

Build amazing things

node.js

TM

# 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 é instalado com o comando **apt install npm**



A screenshot of a terminal window titled "fish /home/david/dev/pw". The window shows the command `david@coiote ~ /dev/pw $ npm --version` followed by the output `9.6.6`. The terminal has a dark theme with light-colored text and icons.

```
david@coiote ~ /dev/pw $ npm --version
9.6.6
david@coiote ~ /dev/pw $ 
```



# 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

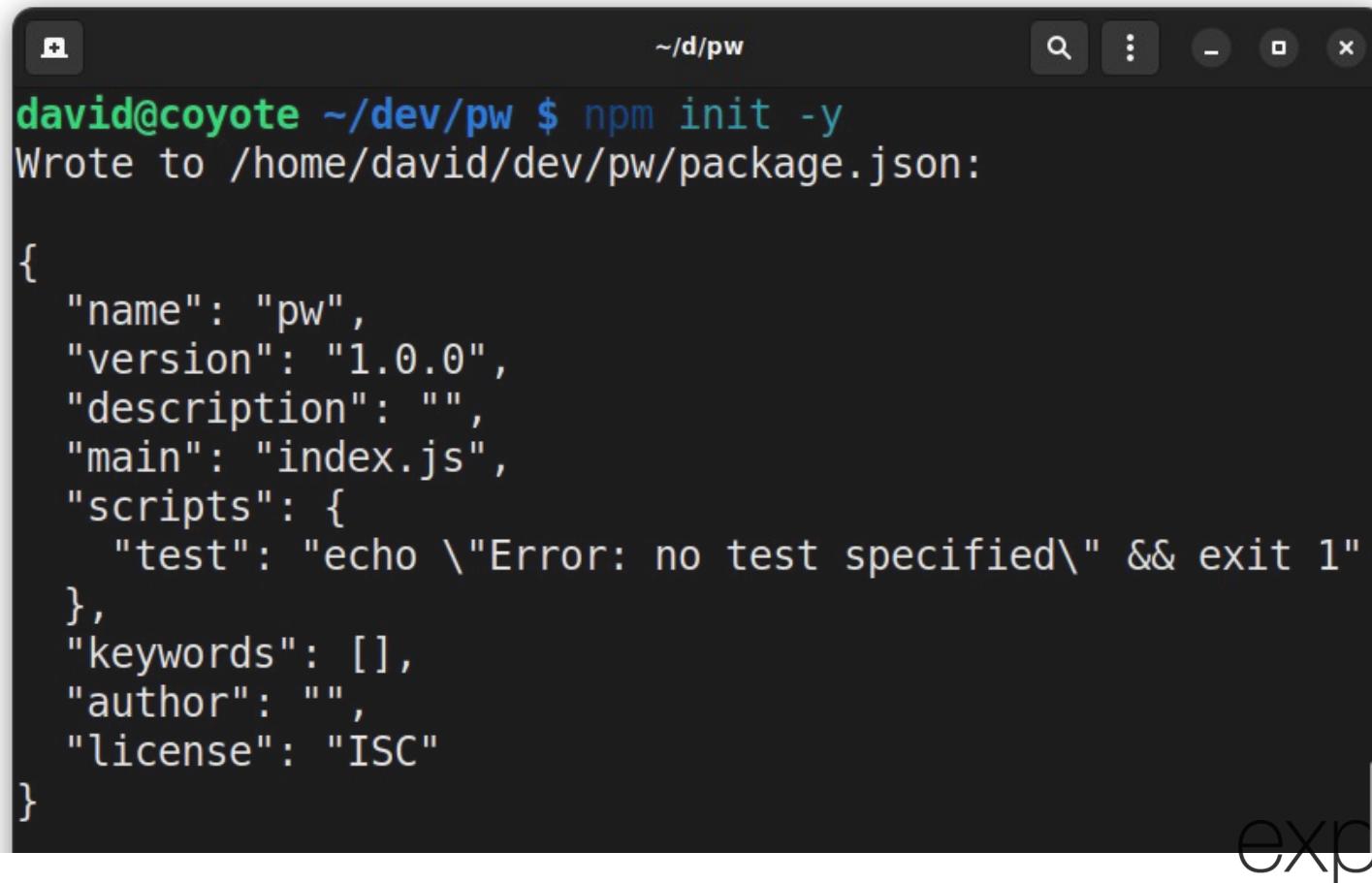
- O primeiro passo no desenvolvimento de uma aplicação node.js é a criação de um arquivo chamado **package.json**
  - A função desse arquivo é armazenar vários metadados sobre a aplicação, incluindo suas dependências

```
{  
  "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

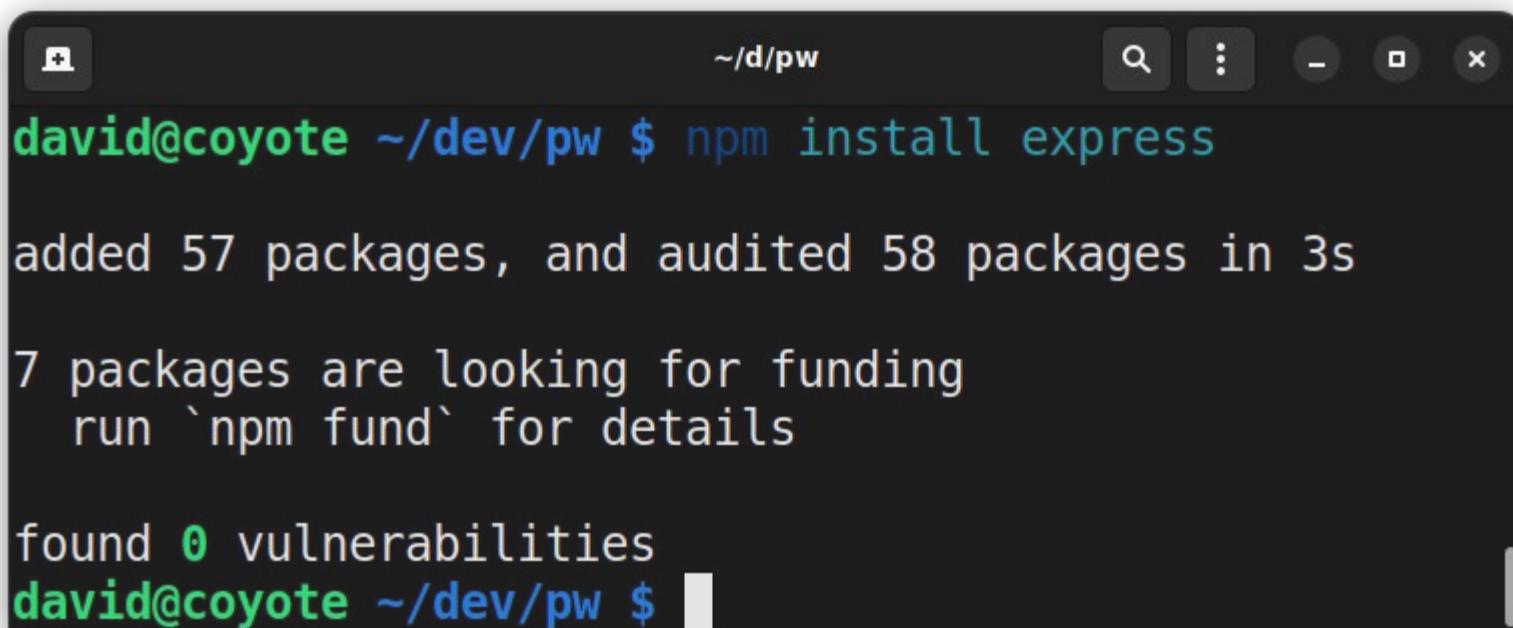


```
~d/pw
david@coyote ~/dev/pw $ npm init -y
Wrote to /home/david/dev/pw/package.json:

{
  "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:

- O pacote express é adicionado automaticamente como uma dependência do projeto

```
{  
  "name": "pw",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {
```

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).

```
  "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

```
~/d/pw
```

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 aplicação é dependente do Express versão 4.\*.\*

Alguns módulos podem ser prefixados com um ~, indicando eles devem 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 $
```

# 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

content-type/          function-bind/    ms/
cookie/                get-intrinsic/   negotiator/
cookie-signature/      has/           object-inspect/
debug/                 has-symbols/    on-finished/
depd/                  http-errors/    parseurl/
destroy/               iconv-lite/     path-to-regexp/
ee-first/              inherits/      proxy-addr/
encodeurl/             ipaddr.js/    qs/
david@coyote ~/dev/pw $
```

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

      mime-db@1.52.0

      negotiator@0.6.3

    array-flatten@1.1.1

    body-parser@1.20.1

      bytes@3.1.2

      content-type@1.0.5 deduped

      debug@2.6.9 deduped

      depd@2.0.0 deduped

      destroy@1.2.0 deduped

      http-errors@2.0.0 deduped

      iconv-lite@0.4.24

      safer-buffer@2.1.2

      on-finished@2.4.1 deduped

      qs@6.11.0 deduped

      raw-body@2.5.1

      bytes@3.1.2 deduped

      http-errors@2.0.0 deduped

      iconv-lite@0.4.24 deduped

      unpipe@1.0.0 deduped

      type-is@1.6.18 deduped

      unpipe@1.0.0

    content-disposition@0.5.4

      safe-buffer@5.2.1 deduped

    content-type@1.0.5

# Node Package Manager – NPM

• **Comando `install`** instala o pacote desejado e todas suas dependências no diretório `node_modules`

• **Comandos do NPM:**

`update <package_name>`

`http-errors@2.0.0 <package_name>`

  iconv-lite@0.4.24

  safer-buffer@2.1.2

  on-finished@2.4.1 deduped

  qs@6.11.0 deduped

  raw-body@2.5.1

  bytes@3.1.2 deduped

  http-errors@2.0.0 deduped

  iconv-lite@0.4.24 deduped

  unpipe@1.0.0 deduped

  type-is@1.6.18 deduped

  unpipe@1.0.0

  content-disposition@0.5.4

    safe-buffer@5.2.1 deduped

  content-type@1.0.5

  function-bind/  
  get-intrinsic/  
  has/  
  ms/  
  negotiator/  
  object-inspect/  
  on-finished/  
  parseurl/  
  path-to-regexp/  
  proxy-addr/  
  qs/  
  setprototypeof/  
  side-channel/  
  statuses/  
  toidentifier/  
  type-is/  
  unpipe/  
  utils-merge/  
  vary/



```
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
```

```
  |  m
```

```
  |  neg
```

```
  |  array
```

```
  |  body-
```

```
  |  byt
```

```
  |  con
```

```
  |  davide
```

```
  |  accel
```

```
  |  arra
```

```
  |  body-
```

```
  |  htt
```

```
  |  ico
```

```
  |  pa
```

```
  |  s
```

```
  |  cont
```

```
  |  on-
```

```
  |  qs@
```

```
  |  raw
```

```
  |  lebug
```

```
  |  lebd
```

```
  |  leptor
```

```
  |  ee-fir
```

```
  |  u
```

```
  |  typ
```

```
  |  unp
```

```
  |  davie
```

```
  |  davi
```

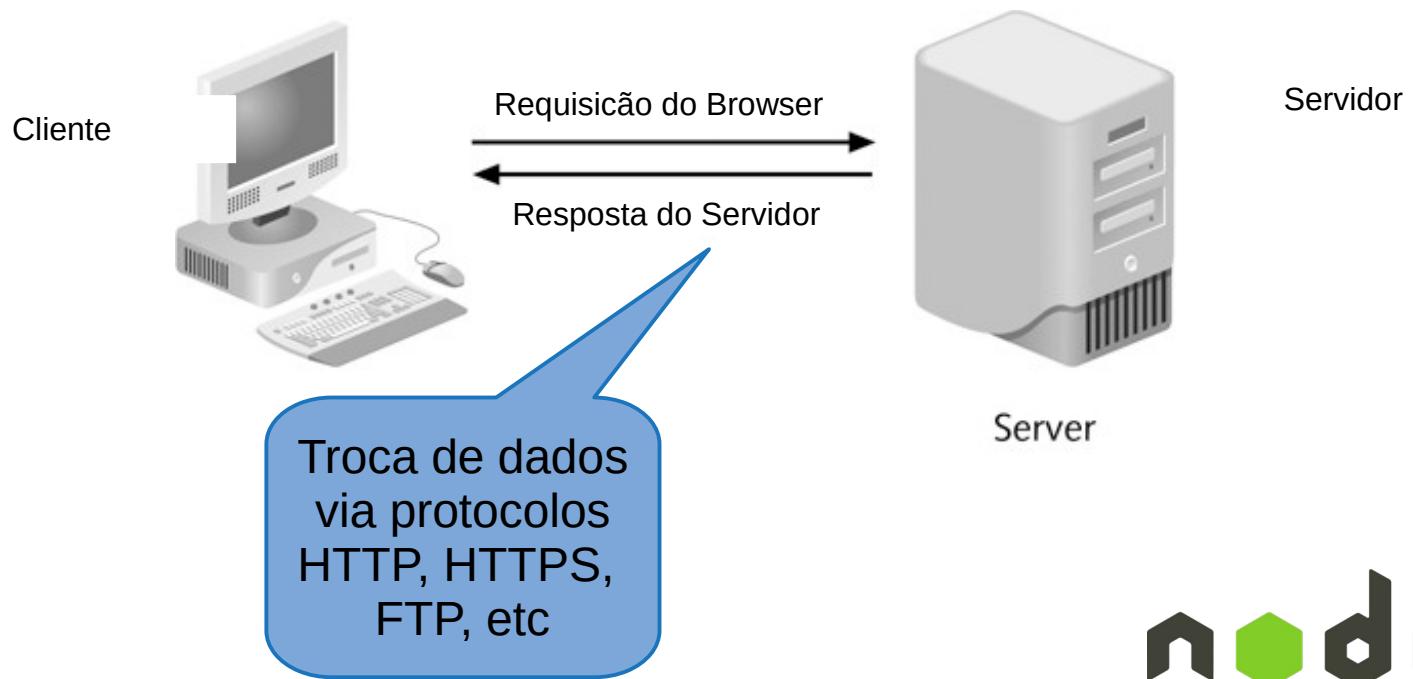
```
  |  un
```



**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}`)
})
```



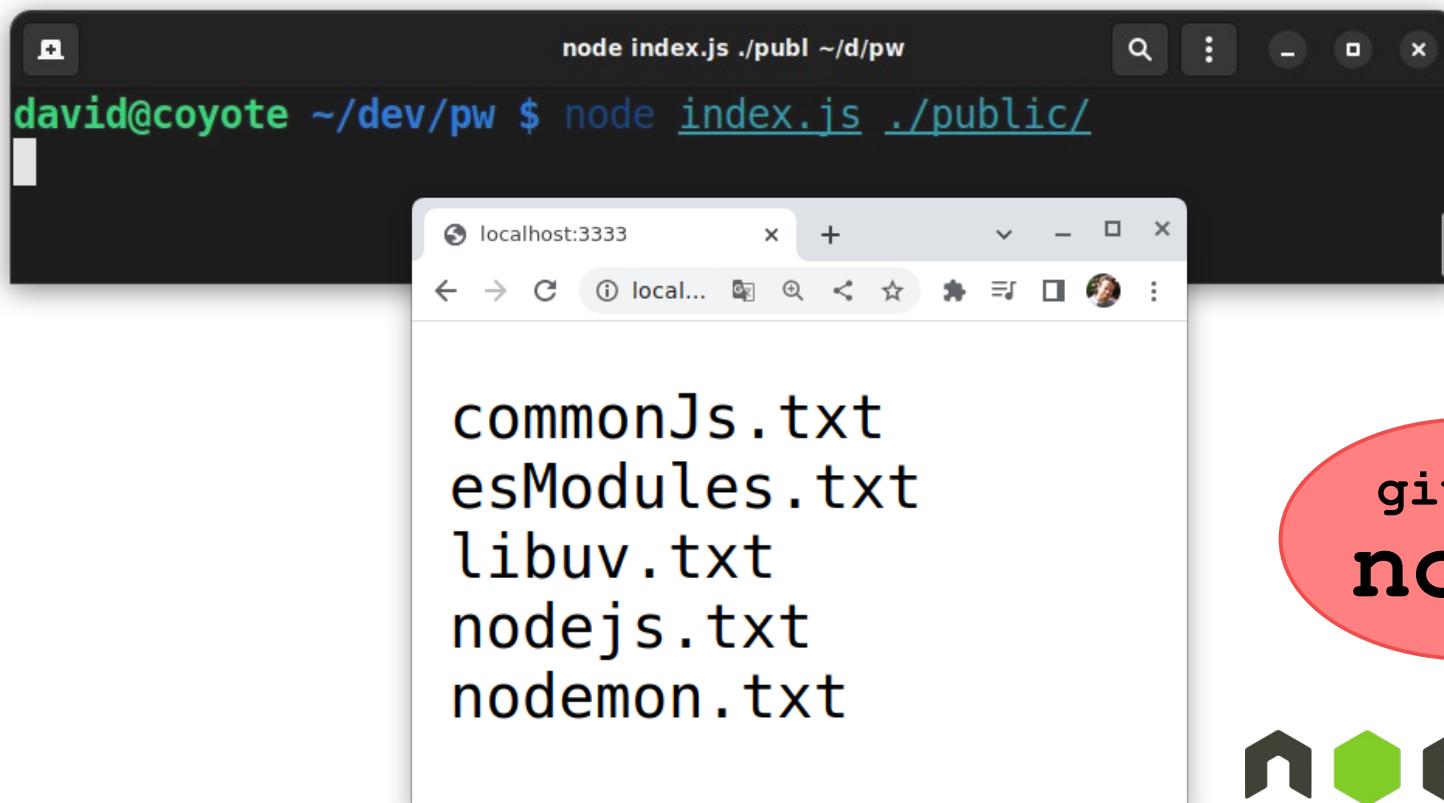
A screenshot of a terminal window titled 'david@coyote ~/dev/pw'. The command 'node index.js icomp ufram' is run, and the output shows the arguments indexed from 0 to 3: '/usr/bin/node', 'index.js', 'icomp', and 'ufam'.

```
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 $
```



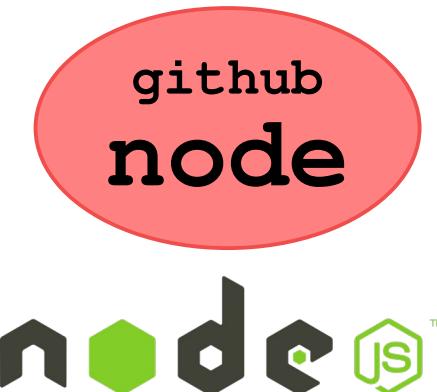
# Exercício

- Usando a função **readdir** do módulo **fs**, desenvolva um programa que aceita o nome de um diretório como parâmetro, 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:

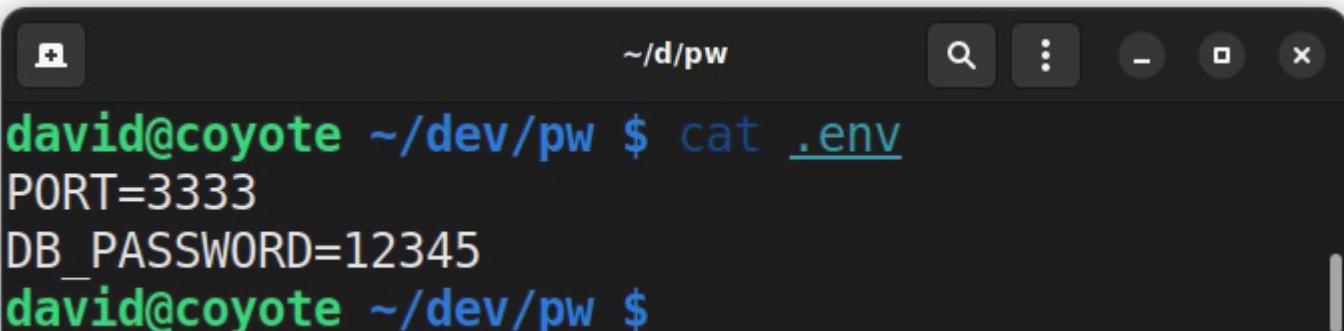
```
process.env.DB_PASSWORD="12345"
```

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);
```



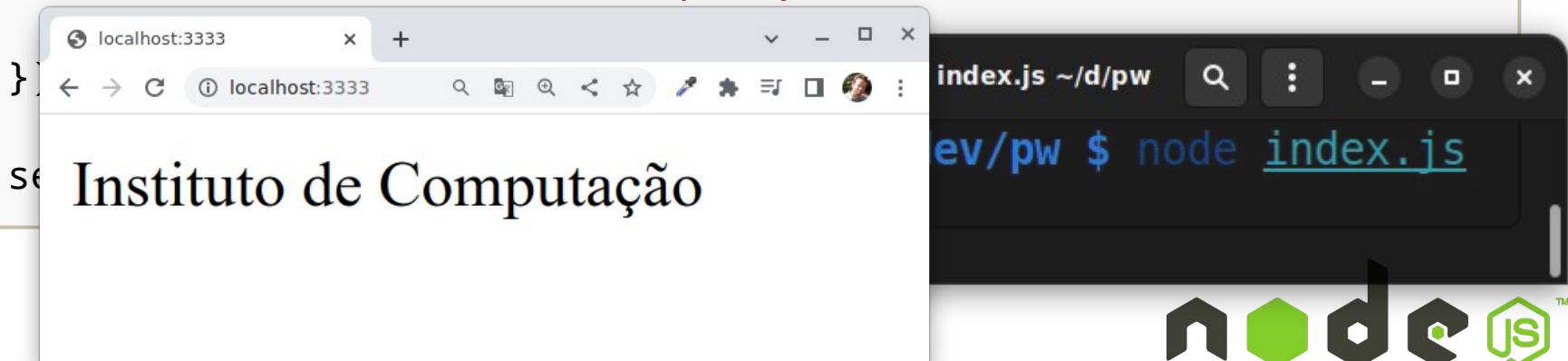
# Variáveis de Ambiente

- 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");
})
```



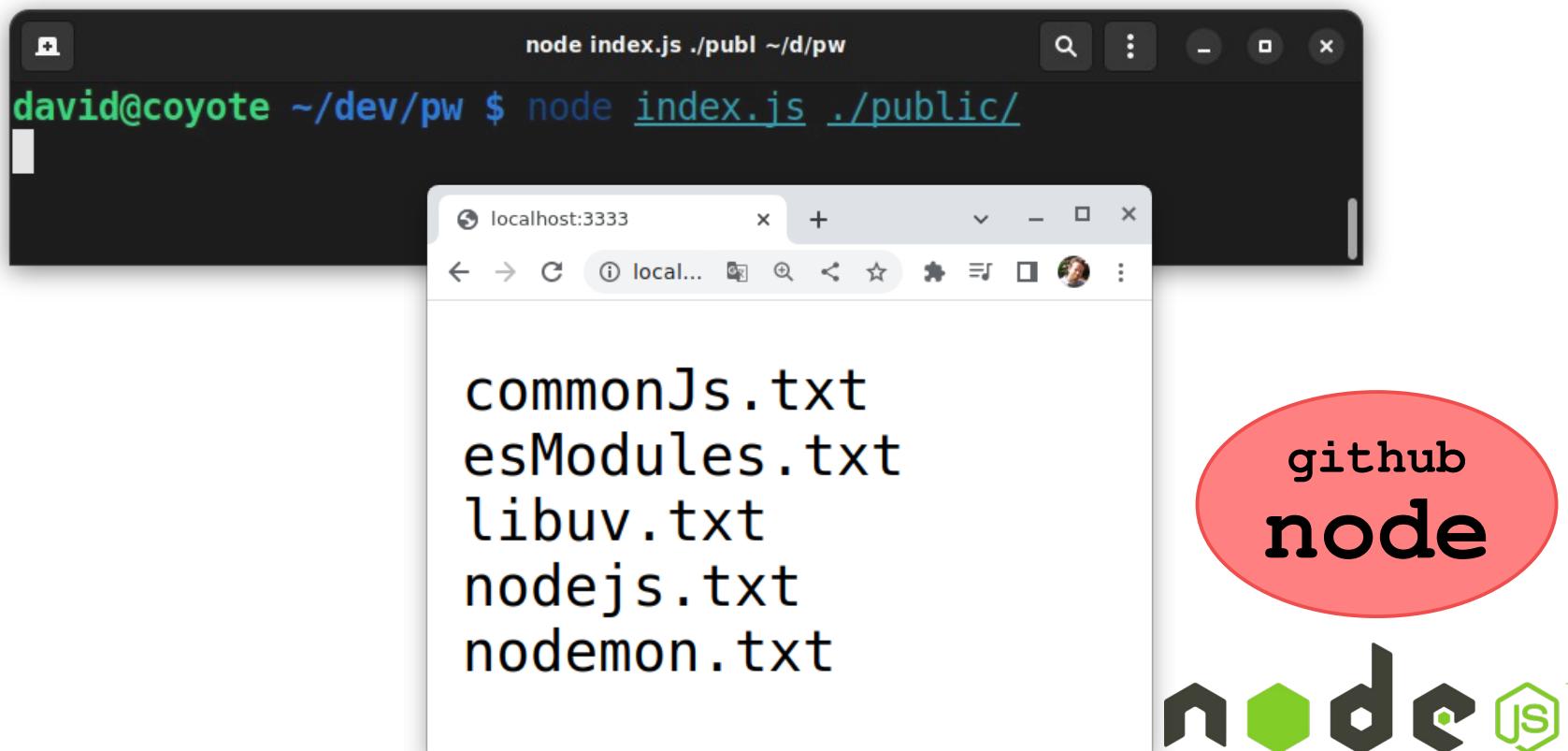
# Variáveis de Ambiente

- 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 II

- 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 JSON configuration file. The bottom window shows the command `npm test` being run, which triggers a script defined in the `scripts` section of the `package.json`. A blue callout points from the text "Para executar o script de exemplo," to the `npm test` command in the bottom window. Another blue callout 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 `package.json`.

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

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 $

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

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

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

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.

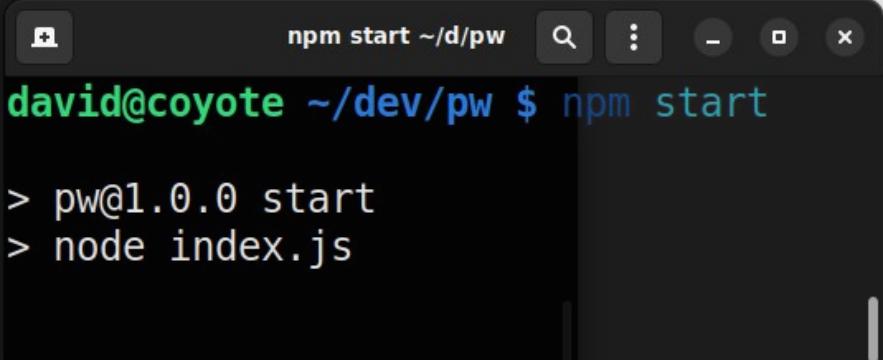
o package.json  
vem só com um  
script test de  
exemplo

```
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



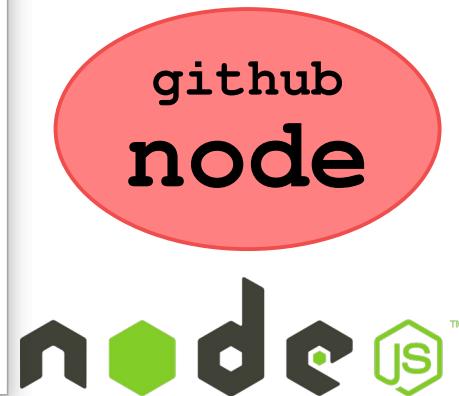
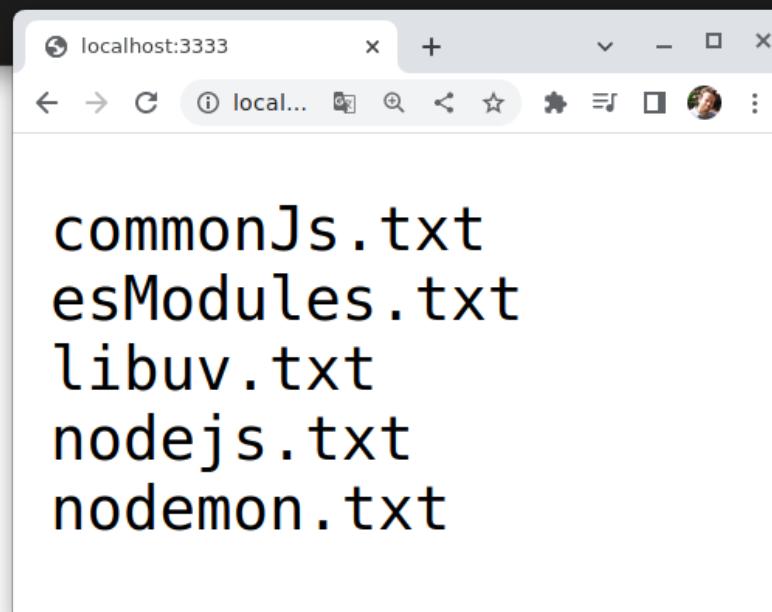
```
~d/pw
david@coyote ~/dev/pw $ cat package.json
{
  "name": "pw",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "dotenv": "^16.0.3",
    "express": "^4.18.2"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

The image shows two terminal windows. The left window displays the contents of the `package.json` file, which includes a `scripts` object with a `start` key set to `"node index.js"`. The right window shows the result of running the `npm start` command, which outputs the command being run: `> pw@1.0.0 start` followed by `> node index.js`.

# Exercício III

- 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

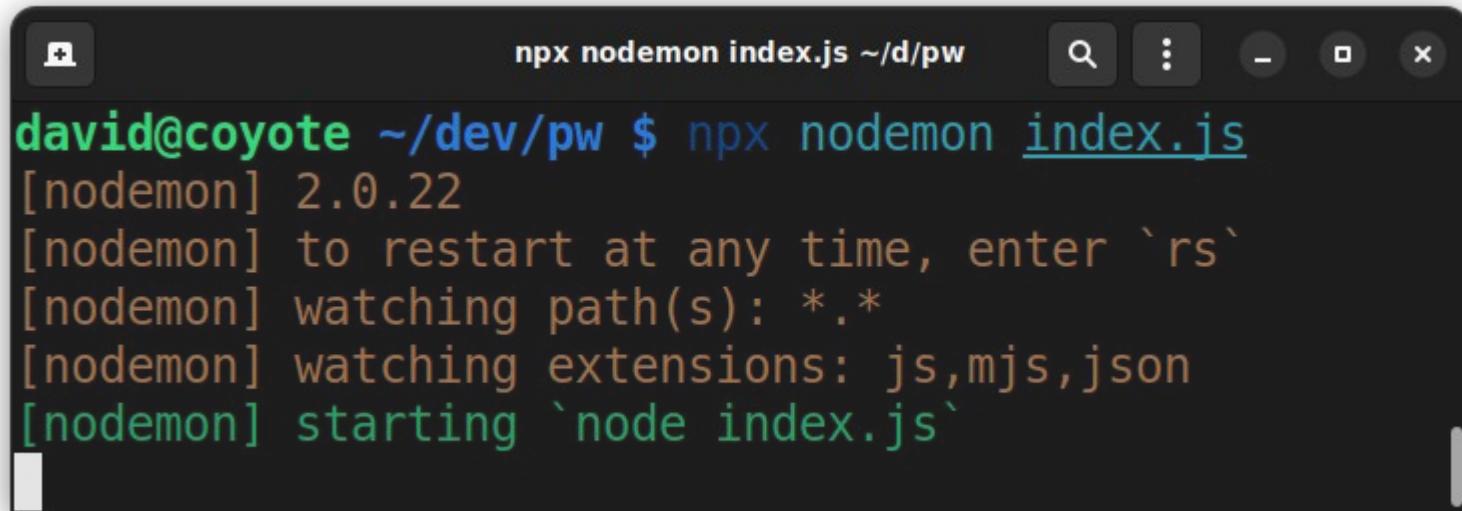
```
david@coyote ~/dev/pw/node_modules/.bin $ ls -la
total 8
drwxrwxr-x  2 david david 4096 mai 14 07:11 .
drwxrwxr-x 89 david david 4096 mai 14 07:11 ..
lrwxrwxrwx  1 david david   14 mai 11 09:49 mime -> ../mime/cli.js*
lrwxrwxrwx  1 david david   25 mai 14 07:11 nodemon -> ../node/bin/nodemon.js*
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 is in green text on a black background and includes:

```
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

```
~/d/pw
david@coyote ~/dev/pw $ cat package.json
{
  "name": "pw",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js",
    "start:prod": "node index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "dotenv": "^16.0.3",
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodemon": "^2.0.22"
  }
}
david@coyote ~/dev/pw $
```



# Exercício IV

- 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 `npm start` script which uses nodemon to run `index.js ./public`. The browser window shows the files available at `localhost:3333`, including `commonJs.txt`, `esModules.txt`, `libuv.txt`, `nodejs.txt`, and `nodemon.txt`.

github  
node

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

node JS

# Módulos do NodeJS

- Também é possível criar seus próprios módulos

```
/* Módulo str_helper.js */

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

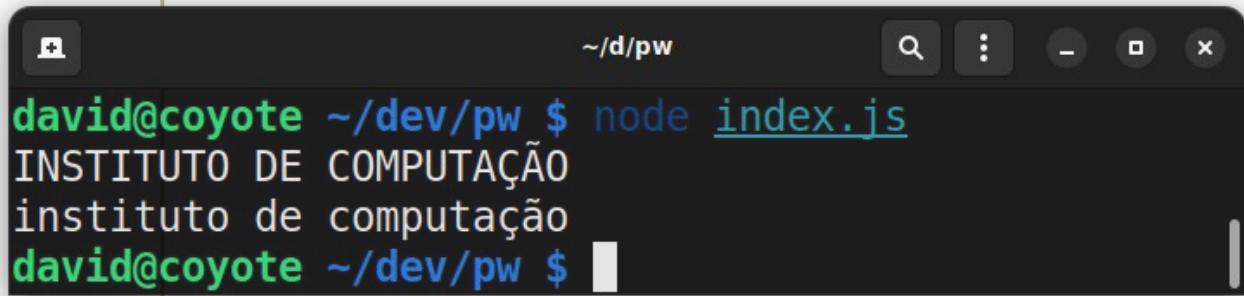
function lower (str) {
  return str.toLowerCase();
}

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();
}

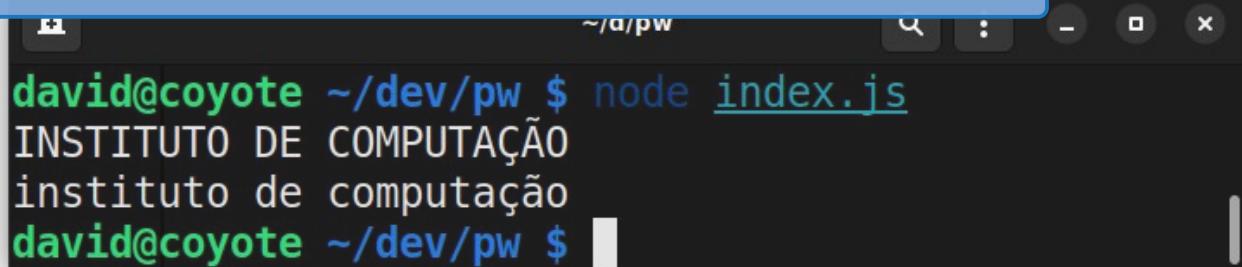
function lower (str) {
  return str.toLowerCase();
}

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;
}
```

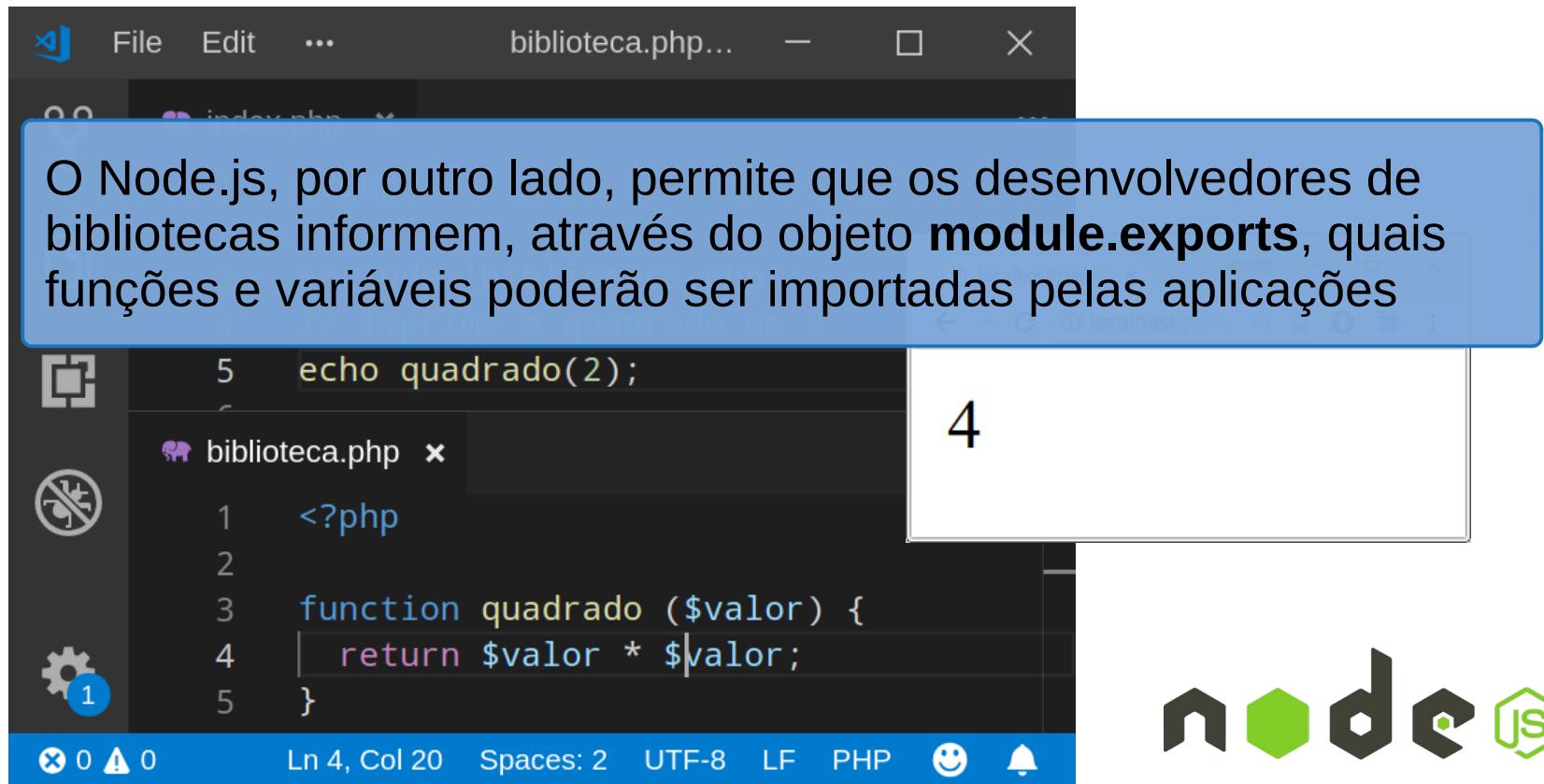
To the right, a browser window displays the result of running index.php, showing the output "4".

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
7 elephantbiblioteca.php ✘
8
9 <?php
10
11 function quadrado ($valor) {
12     return $valor * $valor;
13 }
```

4

Node.js logo

# Exercício V

- 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 util.js separado.

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

The screenshot shows a browser window with two tabs. The left tab, 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 tab, titled 'view-source:localhost:3333', shows the source code for these files. The source code consists of line numbers (1-6) followed by the corresponding HTML link for each file. The code is as follows:

```
1 <a href="/commonJs.txt">commonJs.txt</a><br>  
2 <a href="/esModules.txt">esModules.txt</a><br>  
3 <a href="/libuv.txt">libuv.txt</a><br>  
4 <a href="/nodejs.txt">nodejs.txt</a><br>  
5 <a href="/nodemon.txt">nodemon.txt</a><br>  
6
```

Continua...

# Exercício V

- 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 'node'.

# CommonJs vs ES modules

- O mecanismo de modularização do Node.js, que adota **require** e **module.exports**, é conhecido como **CommonJs**
- O CommonJs foi adotado pelo Node.js em 2009, visto que o EcmaScript não possuia nenhuma forma de modularização

```
// 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

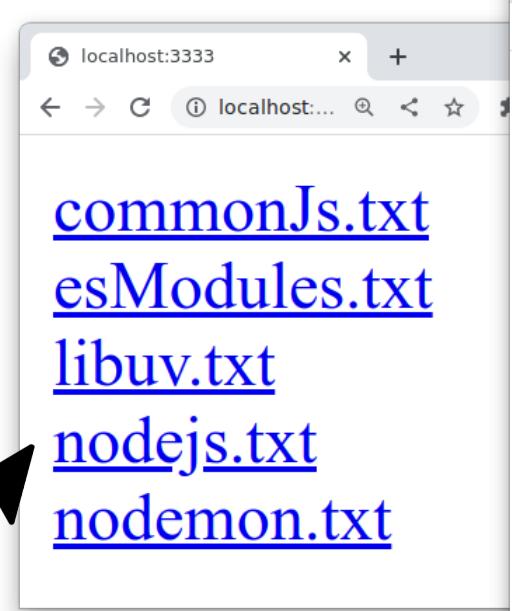
- Outra maneira de habilitar módulos ES é adicionando um campo "type: module" dentro do arquivo package.json
  - Com essa inclusão, não será preciso alterar os arquivos para a extensão **mjs**

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



# Exercício VI

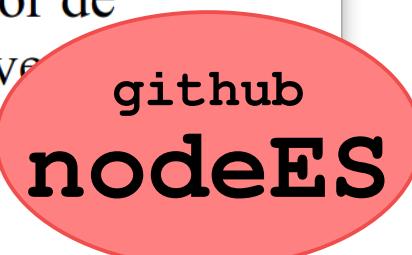
- Refaça o exercício anterior usando ES modules.



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

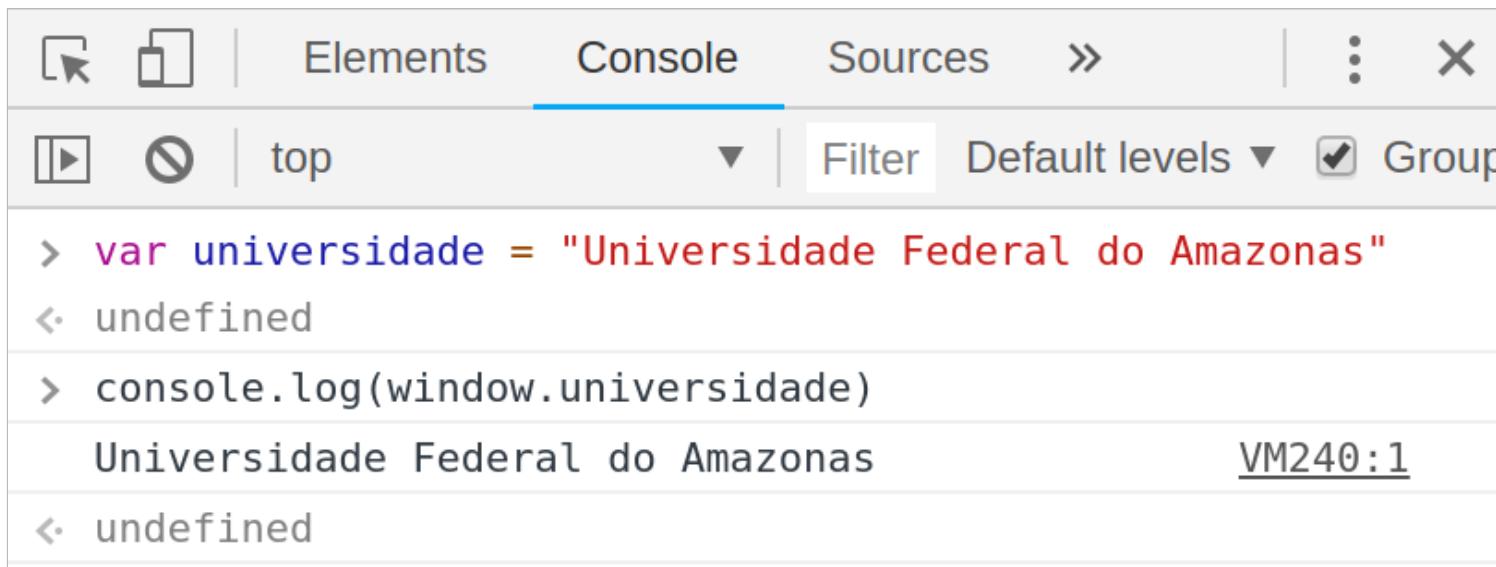
[Voltar](#)

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.



# 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 Chrome DevTools interface with the 'Console' tab selected. The console output area displays the following code and its execution results:

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

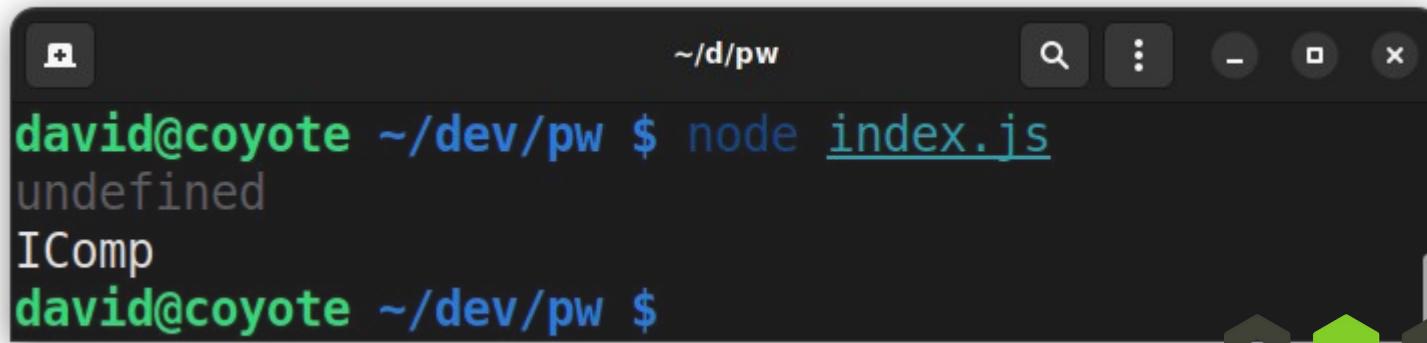
The line 'Universidade Federal do Amazonas' is highlighted in red, indicating it was output from the global scope (the window object). The file reference 'VM240:1' is shown to the right of the log output.

# O escopo Global

- No Node.js, o objeto window não existe e o escopo global é definido pelo objeto **global**
- No entanto, diferente dos browsers, as variáveis declaradas 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, followed by two outputs: 'undefined' and 'IComp'. The terminal has a dark theme with light-colored text. The Node.js logo is visible at the bottom right of the slide.

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



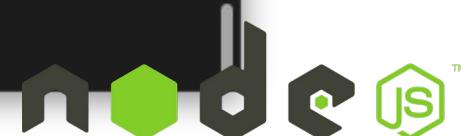
# O escopo Global

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

No Node.js, as variáveis declaradas com **var**, **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)
```

```
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**
- No entanto, diferente dos browsers, as variáveis declaradas ~~fora das funções não são armazenadas no objeto global~~

No Node.js, as variáveis declaradas com **var**, **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 global.js */  
  
universidade = "UFAM"
```

```
/* Arquivo index.js */  
  
const g = require("./global");  
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 determinadas classes podem ser instanciadas 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 determinadas classes podem ser instanciadas 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
```

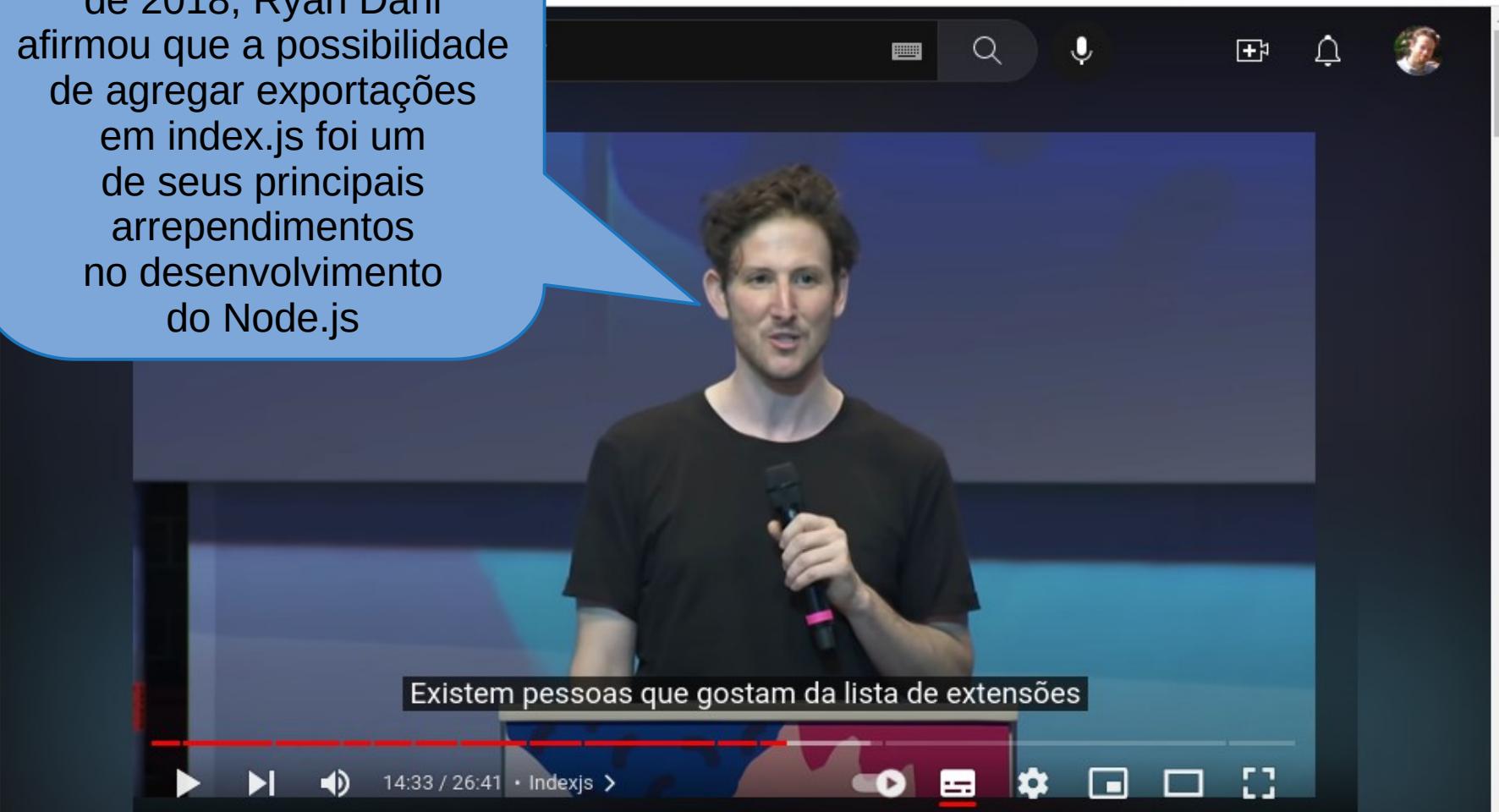
Nesse exemplo, a pasta **util** contém a implementação de um conjunto de componentes

```
// 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

Modules

Docs

Deploy

Community

Search...

Ctrl K

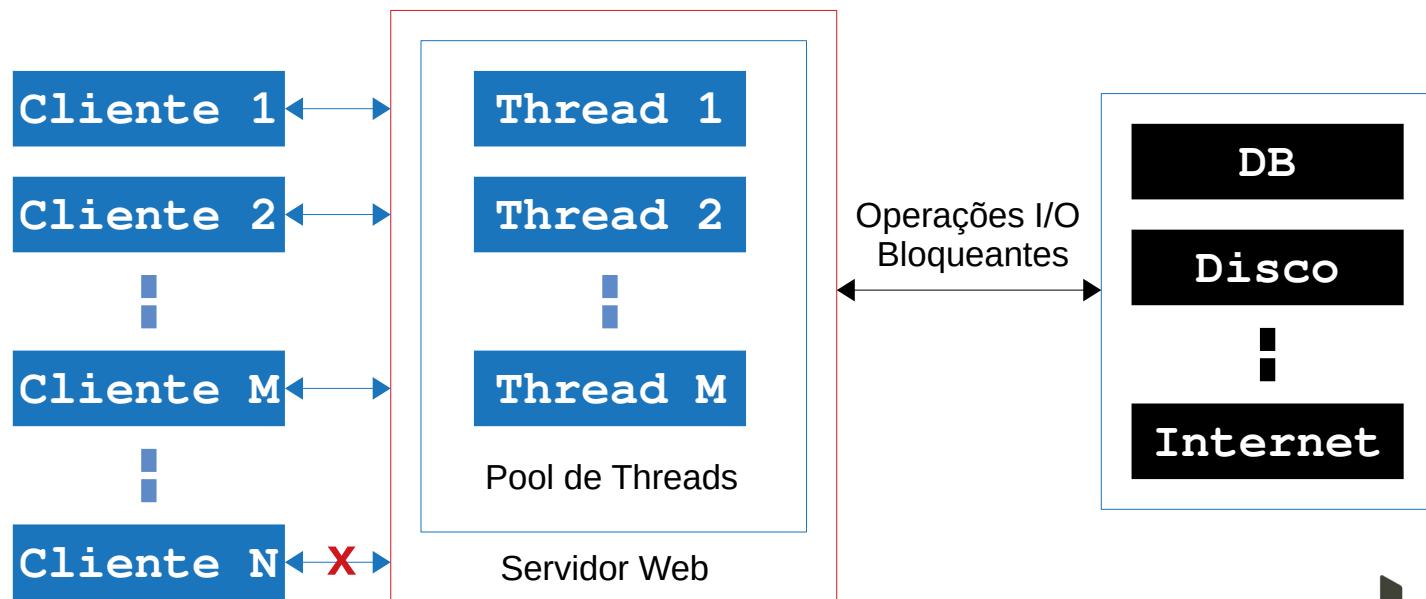
Meet Deno

# The easiest, most secure JavaScript runtime.

[Installation](#)[Documentation](#)

# 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 from a file 'semana.txt' and prints its contents followed by a continuation message. The terminal also displays the Node.js logo at the bottom.

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

# 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

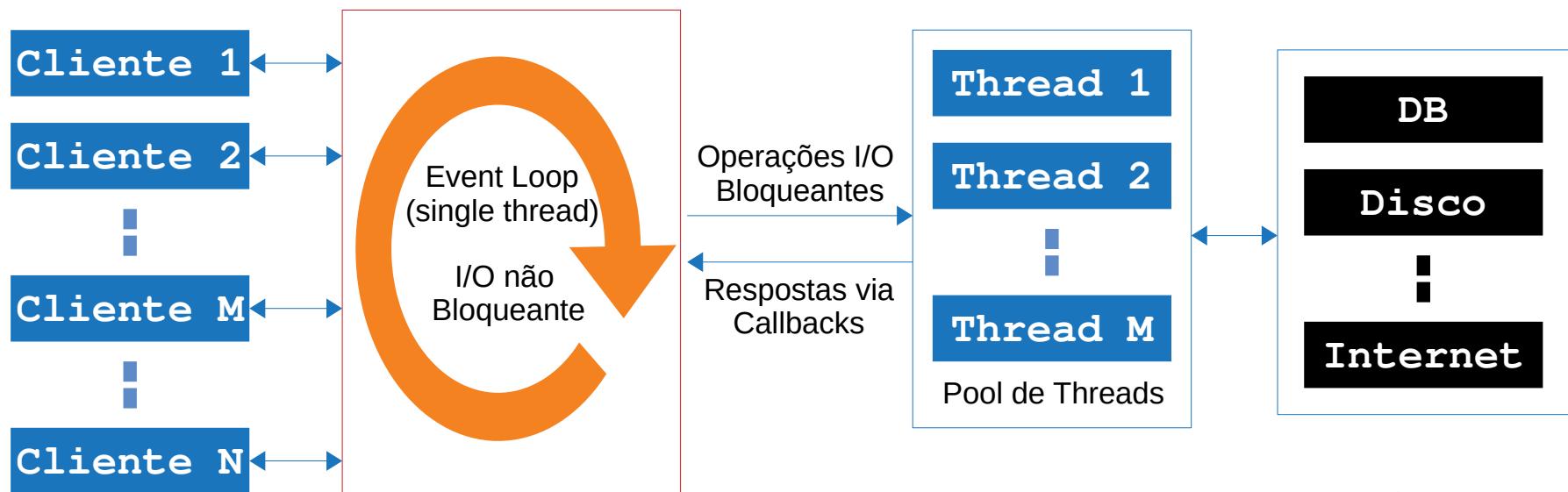
A screenshot of a terminal window titled "david@coyote ~/dev/python \$". The command "python3 semana.py" is being run. The output shows the word "Dominao" followed by a continuation ellipsis "...". The terminal has a dark theme with light-colored text. In the bottom right corner, there are small icons 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

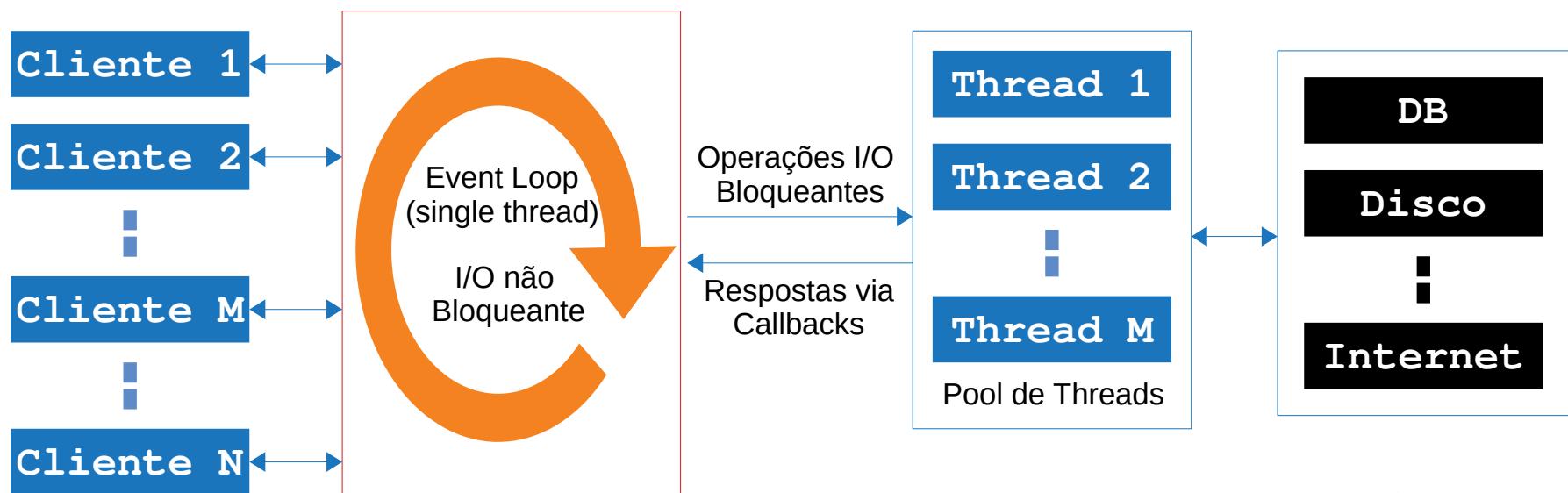
# Single Thread Event Loop

- 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) {
    console.log(conteudo);
});

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

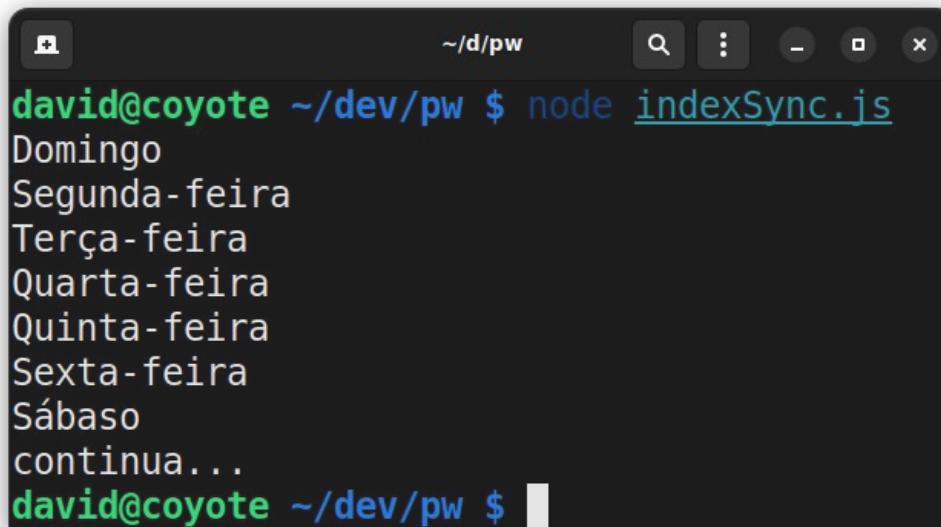


A screenshot of a terminal window titled 'david@coyote ~/dev/pw \$'. The window shows the command 'node index.js' being run. The output of the script is displayed below the command, showing the days of the week: 'continua...', 'Domingo', 'Segunda-feira', 'Terça-feira', 'Quarta-feira', 'Quinta-feira', 'Sexta-feira', and 'Sábado'. The terminal has a dark theme with light-colored text and icons.

# 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 theme with white text and a light gray background.



# 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 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;

http.createServer((req, res) => {
    if (req.url === "/") {
        res.end(`Primo ${counter}`);
    }
}).listen(3000);
```

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

Note que, por padrão, o node.js possui apenas um **event loop** sendo executado em apenas um núcleo do processador.



# 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 de código empilhado.



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

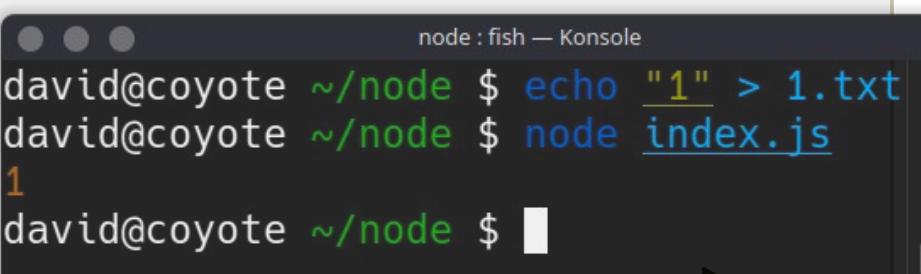
# Promises

- 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)
});
```



The terminal window shows the command `node index.js` being run, which reads the file `1.txt` containing the number `1` and logs it to the console.

```
node : fish — Konsole
david@coyote ~/node $ echo "1" > 1.txt
david@coyote ~/node $ node index.js
1
david@coyote ~/node $
```



# 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);
  })
}

node : fish — Konsole
david@coyote ~/node $ node index.
1
2
david@coyote ~/node $
```



# 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(parseInt(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);
});
```

A terminal window titled "node : fish — Konsole" displays the execution of a Node.js script. The script reads two files, "1.txt" and "2.txt", containing the numbers "1" and "2" respectively, and logs their sum to the console.

```
david@coyote ~/node $ echo "1" > 1.txt
david@coyote ~/node $ echo "2" > 2.txt
david@coyote ~/node $ node index.js
3
```

# 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);
    });
}
```



# 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(function (resolve, reject) {
    fs.readFile(filename, function (err, data) {
      if (err) {
        reject(err);
      } else {
        resolve(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');
```

node : fish — Konsole

```
david@coyote ~/node $ echo "1" > 1.txt
david@coyote ~/node $ echo "2" > 2.txt
david@coyote ~/node $ node index.js
a
```

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



# 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 VII

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)

