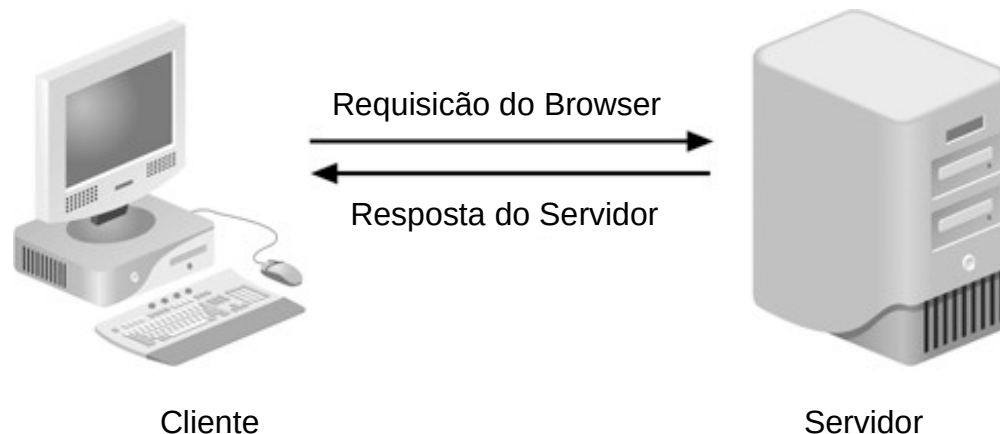




Prof. David Fernandes de Oliveira
Instituto de Computação
UFAM

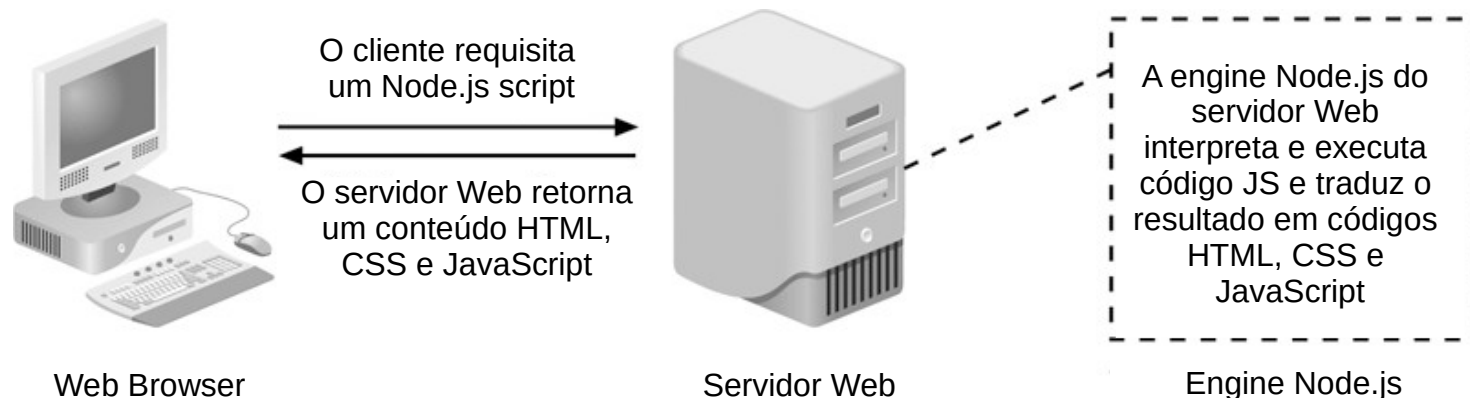
Arquitetura Cliente/Servidor

- 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



libuv documentation

docs.libuv.org/en/v1.x/



libuv
documentation


Search

Design overview

API documentation

User guide

Upgrading

 v: v1.x

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 ponta pé inicial para a criação do **Node.js**



- Na JSConf
progra
apres
execu
basea
- Aprov
cidade
facilit
aplica
- Foi o p
do No



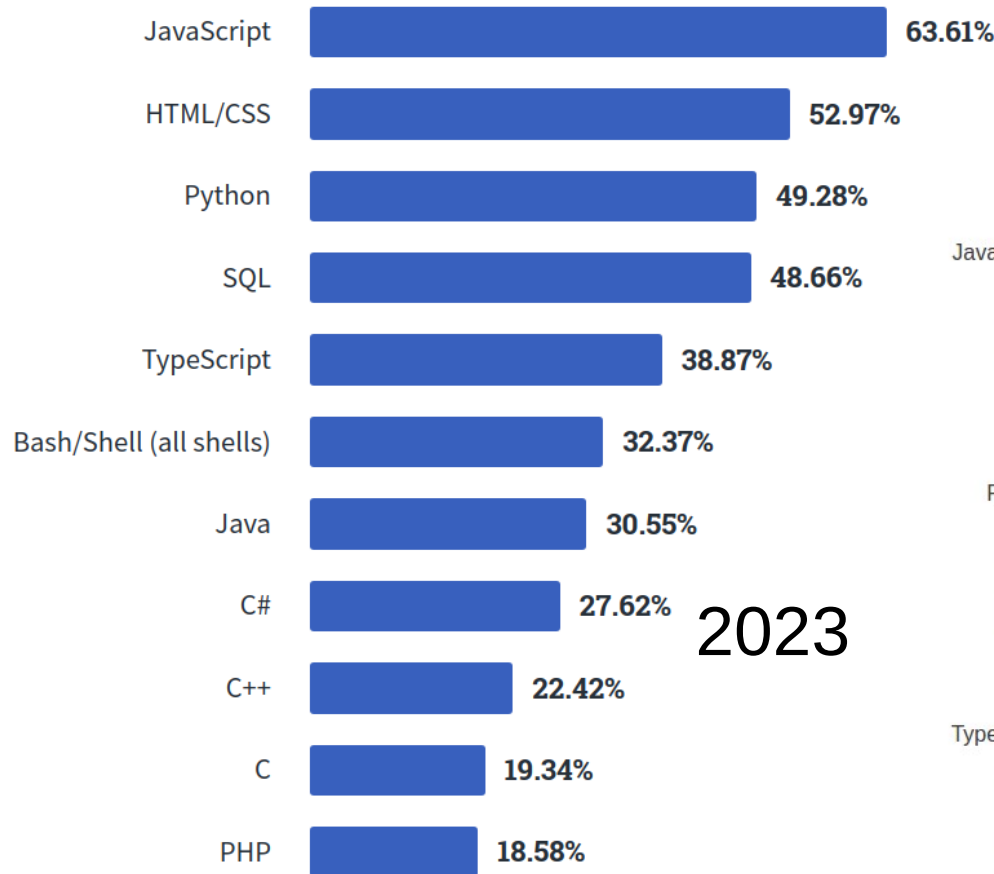
Porque usar Node.js?

- **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

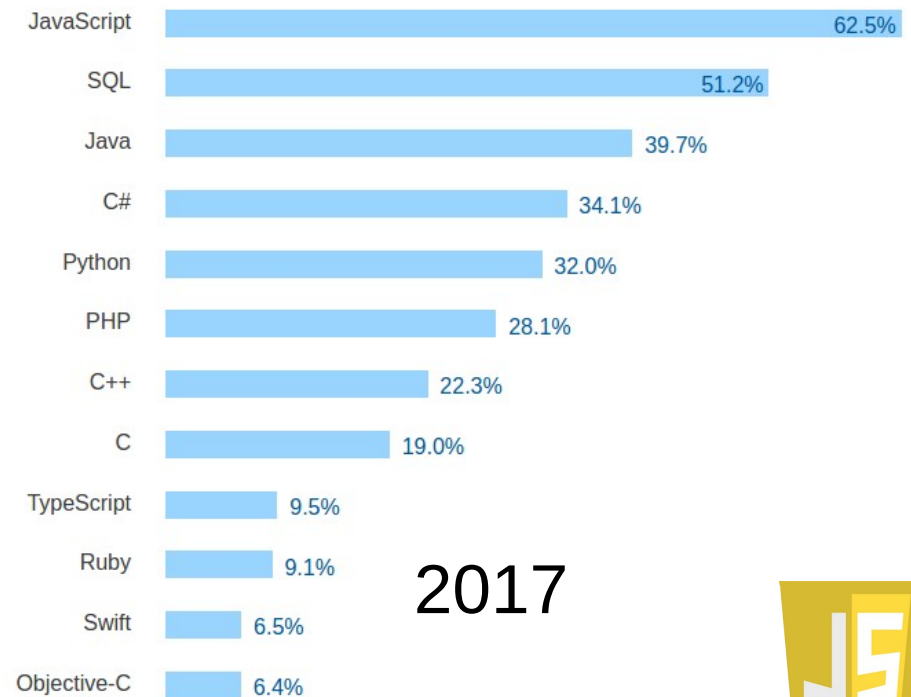


StackOverflow Surveys

Programming, Scripting, and Markup Languages



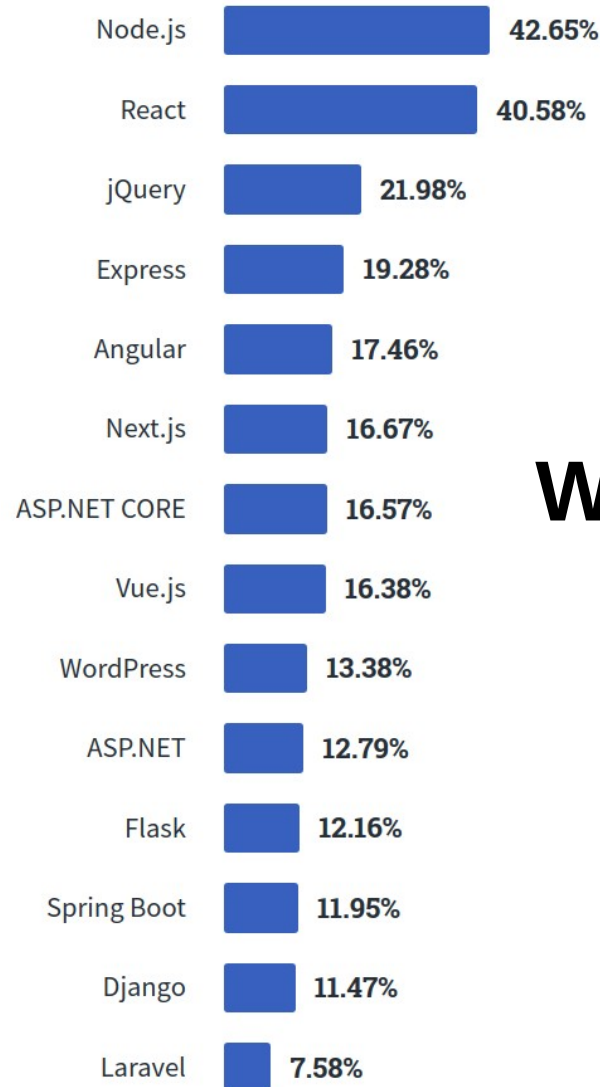
2023



2017



StackOverflow Surveys



Web Frameworks 2023



Quem usa Node.js?



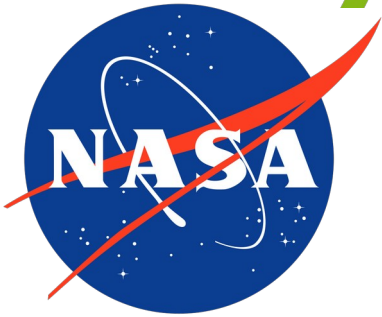
CODEBENCH

Linked 

ebay

PayPaltm

UBER



GROUPON

NETFLIX

YAHOO!

IBM

 **Trello**



GoDaddyTM

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 Computação")
```



A terminal window with a dark background and a title bar showing the path ~/d/pw. The terminal displays the command 'node index.js' being executed by 'david@coyote' in the directory ~/dev/pw. The output of the command is 'Instituto de Computação'.

```
~/d/pw  
david@coyote ~/dev/pw $ node index.js  
Instituto de Computação  
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');
```

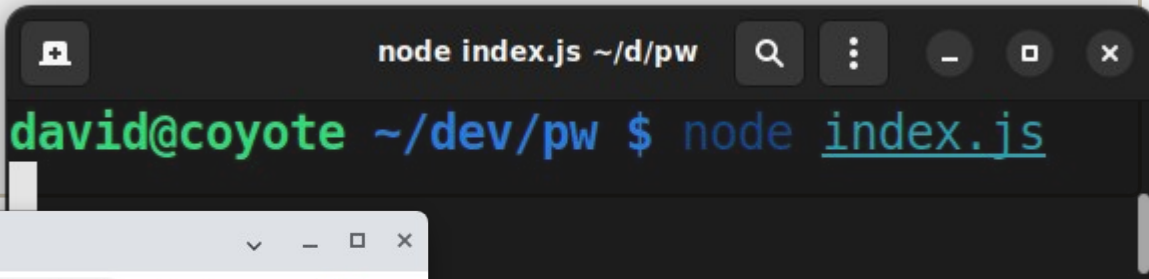
- Alguns módulos embarcados do Node.js: **http**, **url**, **path**, **fs**, **os**, **sys**, **tty**, **cluster**, **process**, **timers** e **util**



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(  
  res.writeHead(200, {"Content-Type": "text/plain; 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**

Instituto de Computação



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.



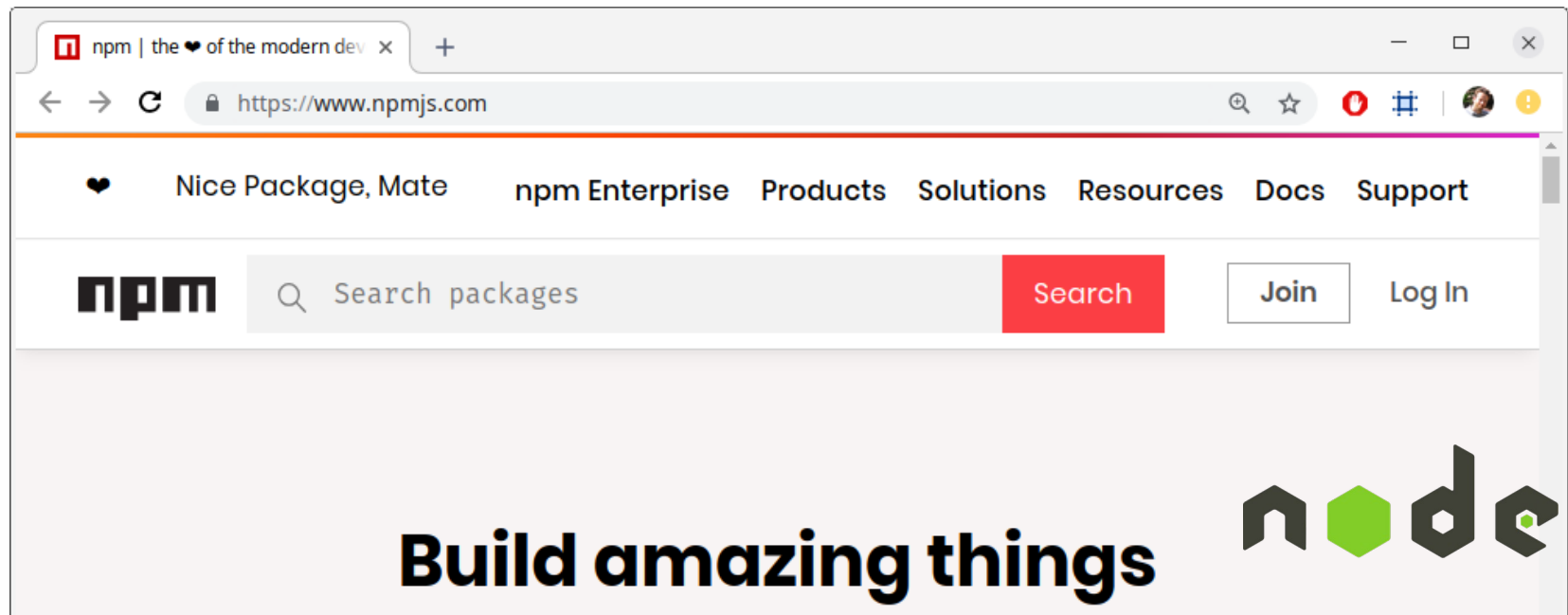
O Módulo FS

- O **módulo FS** possui vários outros métodos, dentre os quais podemos destacar:
 - **fs.access()**: verifica se o arquivo existe
 - **fs.chmod()**: muda as permissos 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



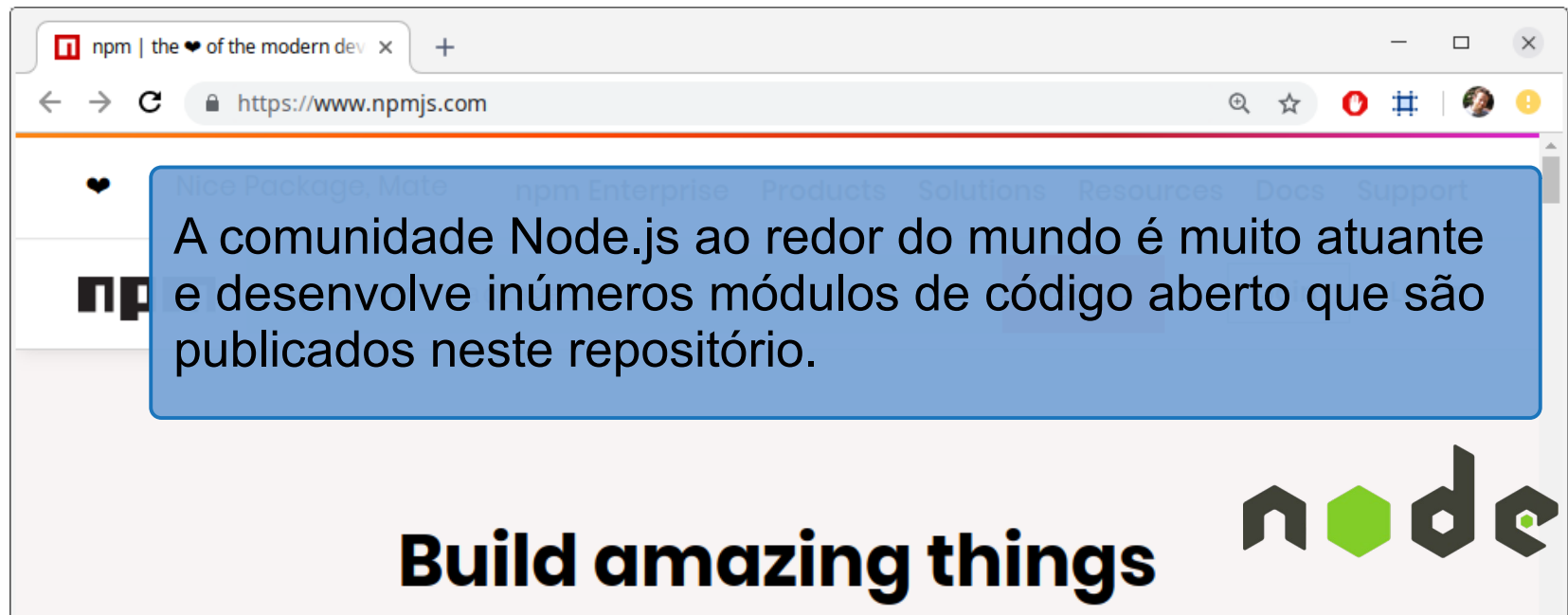
Node Package Manager – NPM

- 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/>

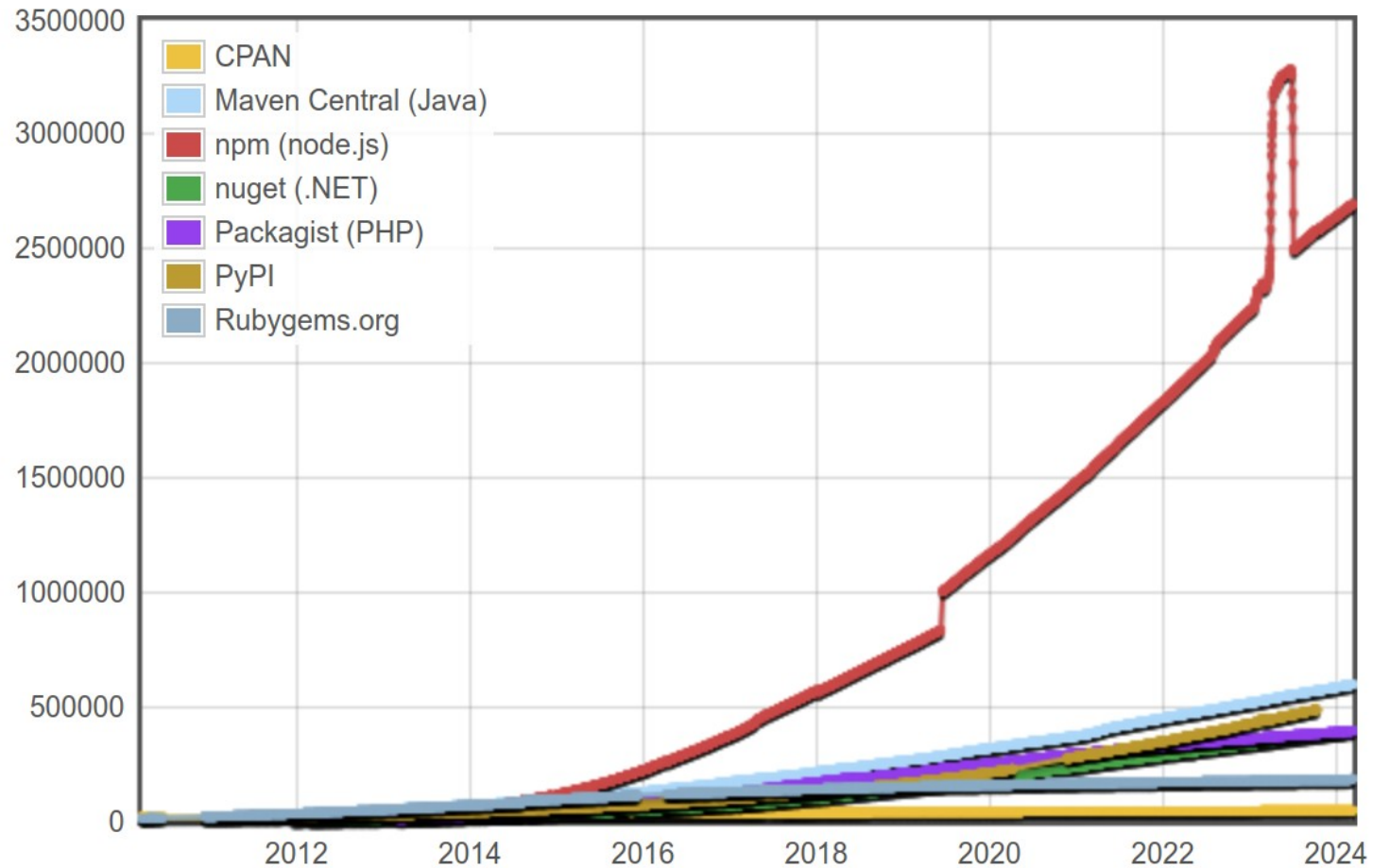


Node Package Manager – NPM

- 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/>



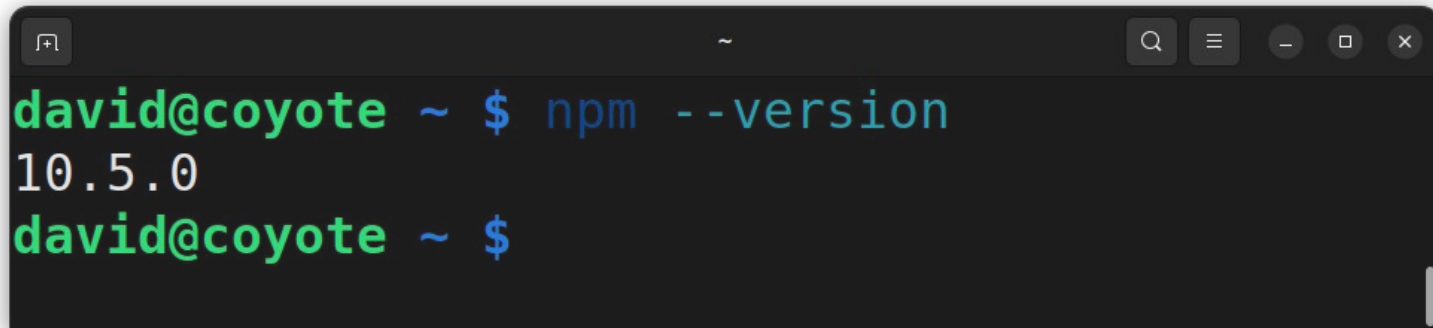
Node Package Manager – NPM



<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**

A terminal window with a dark background and light green text. The prompt is 'david@coyote ~ \$'. The command 'npm --version' has been entered and executed, resulting in the output '10.5.0'. The prompt is now 'david@coyote ~ \$'. The window has standard Linux window controls (minimize, maximize, close) in the top right corner.

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



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

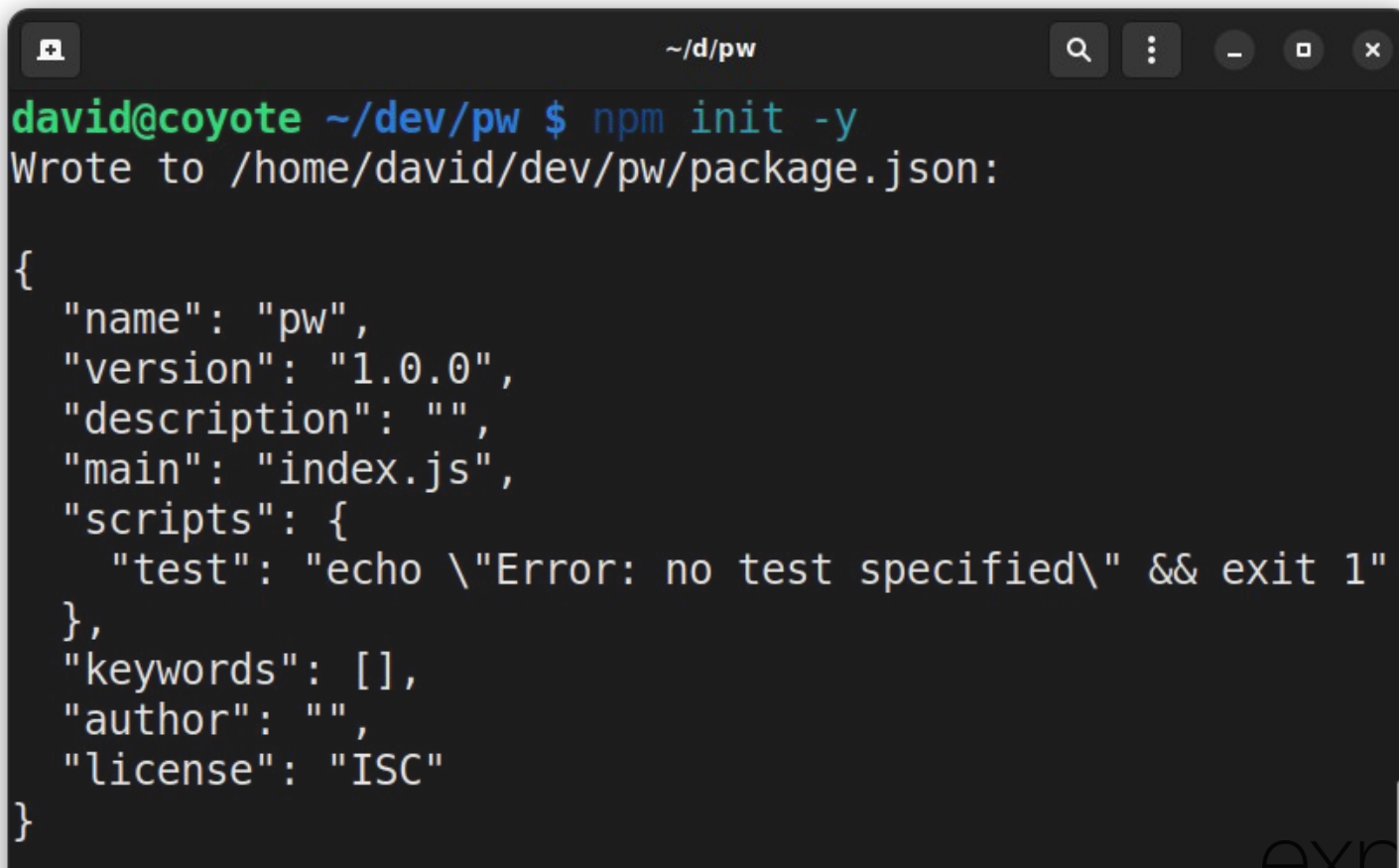
- O primeiro passo no desenvolvimento de uma aplicação node.js é a criação de um arquivo chamado **package.json**
 - A função desse arquivo é armazenar vários metadados sobre a aplicação, incluindo suas dependências

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

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

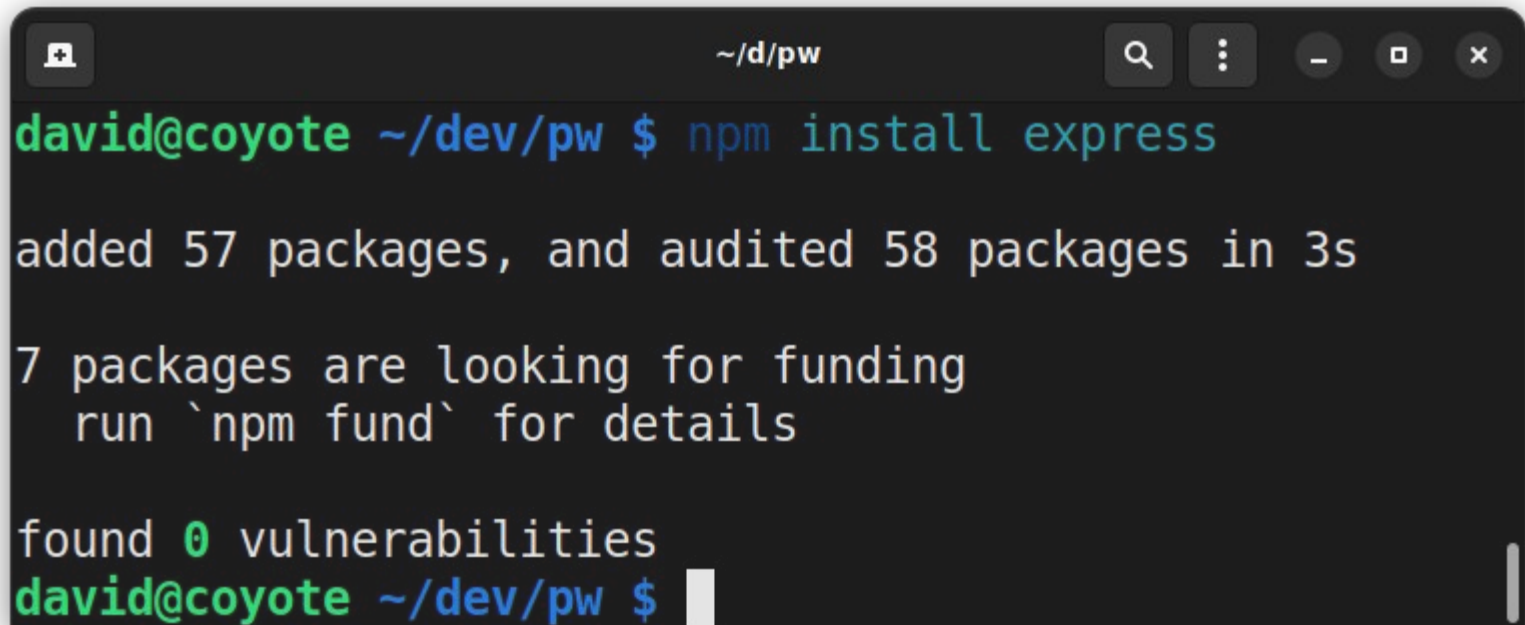


```
david@coyote ~/dev/pw $ npm init -y
Wrote to /home/david/dev/pw/package.json:

{
  "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

exec

~ / d / pw 🔍 ⋮ − □ ×

- O `package.json` é adicionado automaticamente como uma dependência do projeto

```
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 $
```

express **JS**

Adicionando o Express no Projeto

- Para incluirmos o framework express no projeto, basta

```
executed
~ /d/pw
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"
  },
  "author": "David",
  "license": "ISC"
}

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

added 1 package, and audited 1 package in 1s
found 0 vulnerabilities

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

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).

express JS

Adicionando o Express no Projeto

- Para incluirmos o framework express no projeto, basta

executar



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


Node Package Manager – NPM

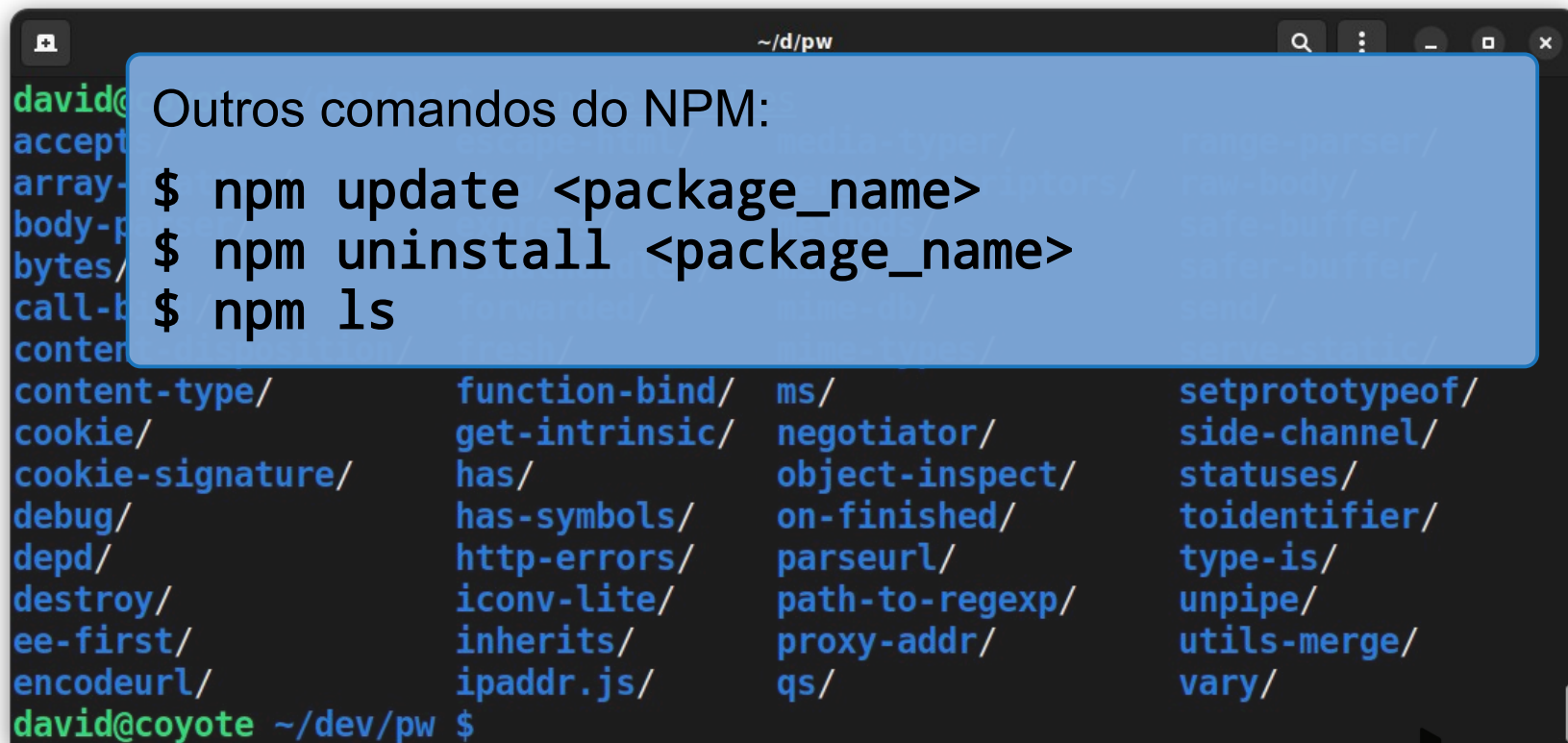
- O comando `npm install` instala o pacote desejado e todas suas dependências no diretório `node_modules`

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



Node Package Manager – NPM

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



A terminal window with a dark background. At the top, a blue semi-transparent box contains the text "Outros comandos do NPM:" followed by three commands: "\$ npm update <package_name>", "\$ npm uninstall <package_name>", and "\$ npm ls". Below this box, the terminal shows a list of installed packages in a grid-like format. The packages are: content-type/, cookie/, cookie-signature/, debug/, depd/, destroy/, ee-first/, encodeurl/, function-bind/, get-intrinsic/, has/, has-symbols/, http-errors/, iconv-lite/, inherits/, ipaddr.js/, 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/, and vary/. The prompt "david@coyote ~/dev/pw \$" is visible at the bottom left.

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

Outros comandos do NPM:

```
$ npm update <package_name>
$ npm uninstall <package_name>
$ npm ls
```

content-type/	function-bind/	ms/	setprototypeof/
cookie/	get-intrinsic/	negotiator/	side-channel/
cookie-signature/	has/	object-inspect/	statuses/
debug/	has-symbols/	on-finished/	toidentifier/
depd/	http-errors/	parseurl/	type-is/
destroy/	iconv-lite/	path-to-regexp/	unpipe/
ee-first/	inherits/	proxy-addr/	utils-merge/
encodeurl/	ipaddr.js/	qs/	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

│ │ ├── 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

│ ├── 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

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

Comandos do NPM:

`npm update <package_name>`

`npm install <package_name>`

function-bind/

ms/

setprototypeof/

get-intrinsic/

negotiator/

side-channel/

has/

object-inspect/

statuses/

has-symbols/

on-finished/

toidentifier/

http-errors/

parseurl/

type-is/

iconv-lite/

path-to-regexp/

unpipe/

inherits/

proxy-addr/

utils-merge/

addr.js/

qs/

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
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

```
├── ...
```

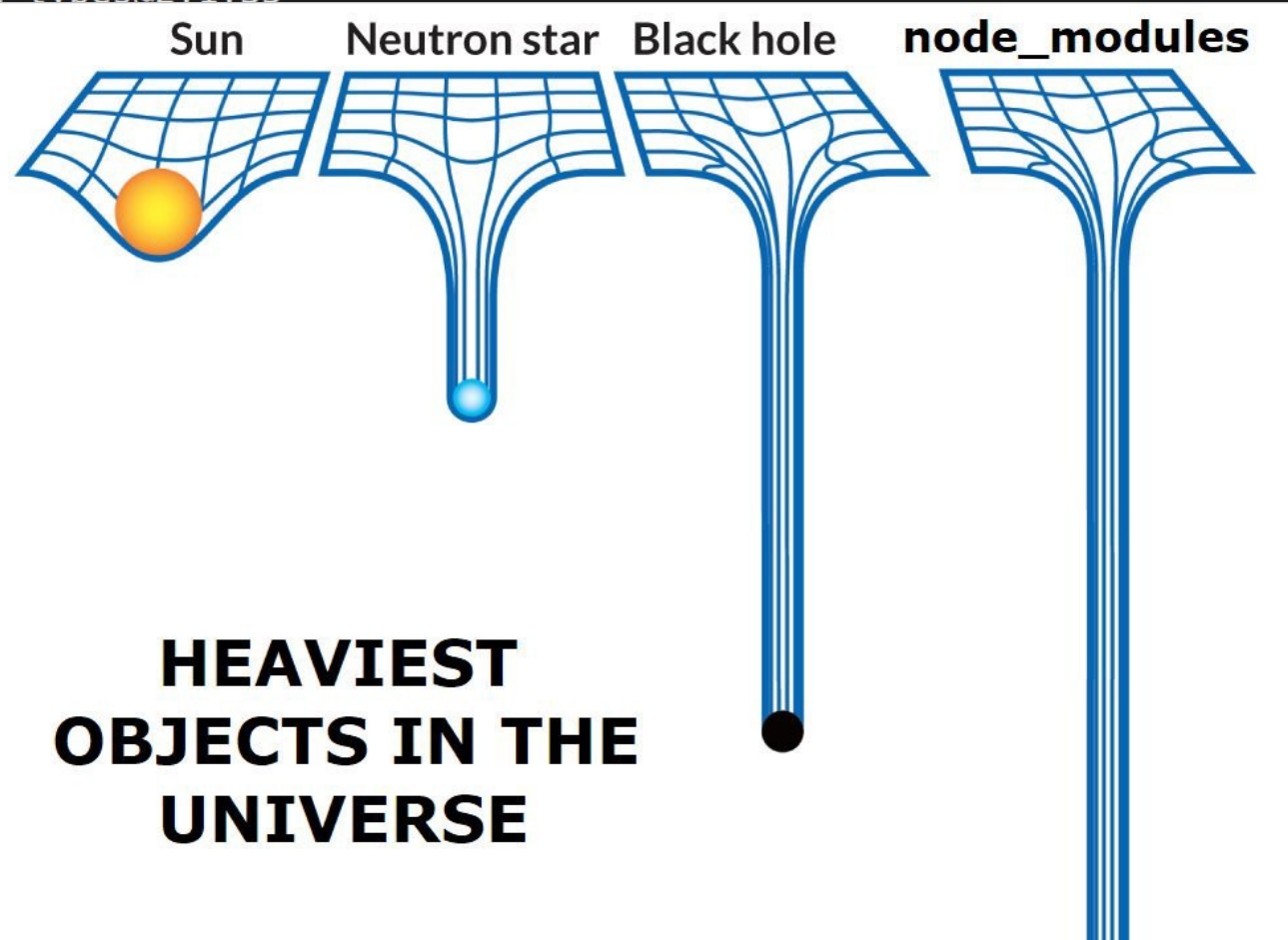
```
├── ...
```

```
├── ...
```

```
├── content-disposition@0.5.4
```

```
├── safe-buffer@5.2.1 deduped
```

```
├── content-type@1.0.5
```

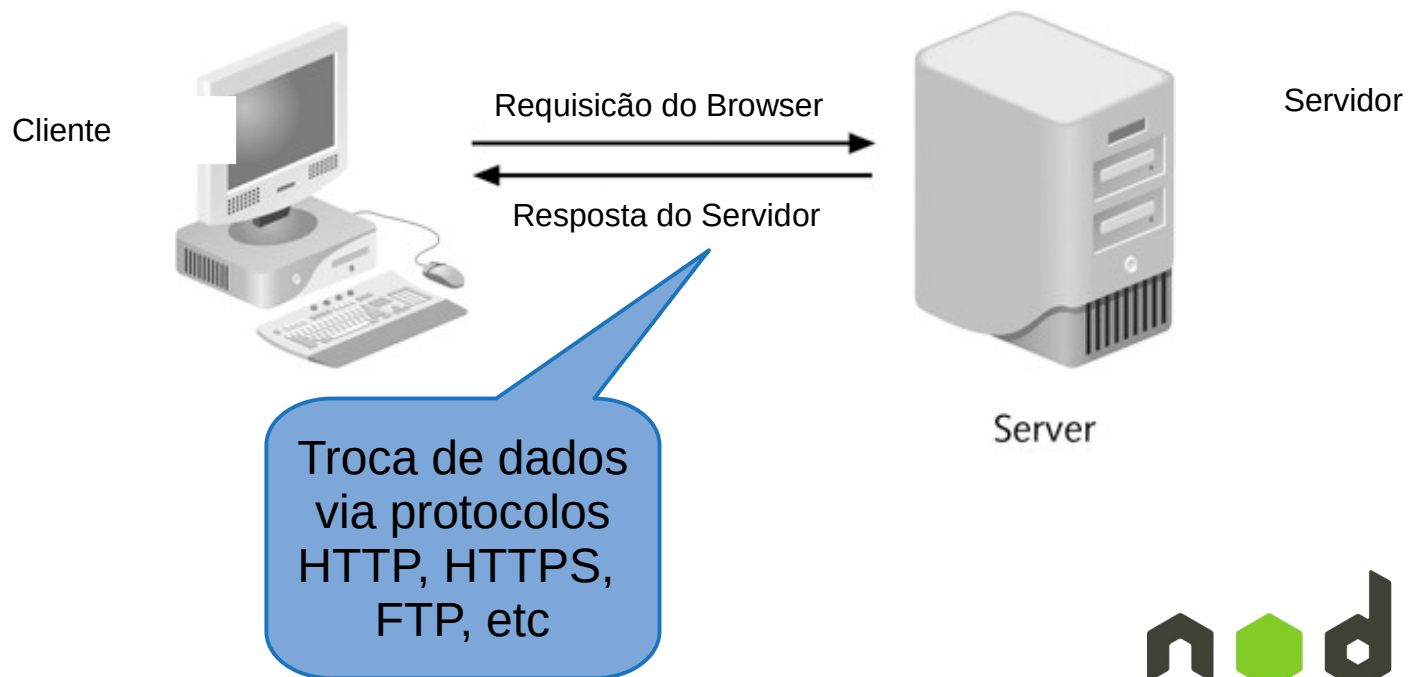




**Meu backup do diretório
node_modules!**

Servidores Web

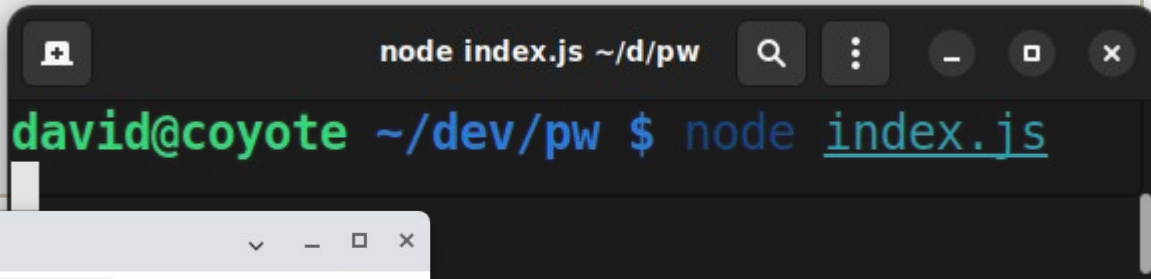
- O Node.JS pode ser usado para desenvolver todo tipo de aplicação, mas o seu principal uso é a criação de **Web Apps**
- Nesse contexto, um **servidor Web** é um sistema que responde a solicitações de clientes feitas pela World Wide Web



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
```

localhost:3333

localhost:3333

Instituto de Computação



Passagem de Parâmetro – ARGV

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

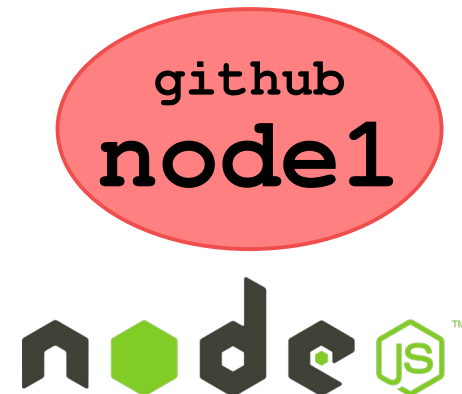
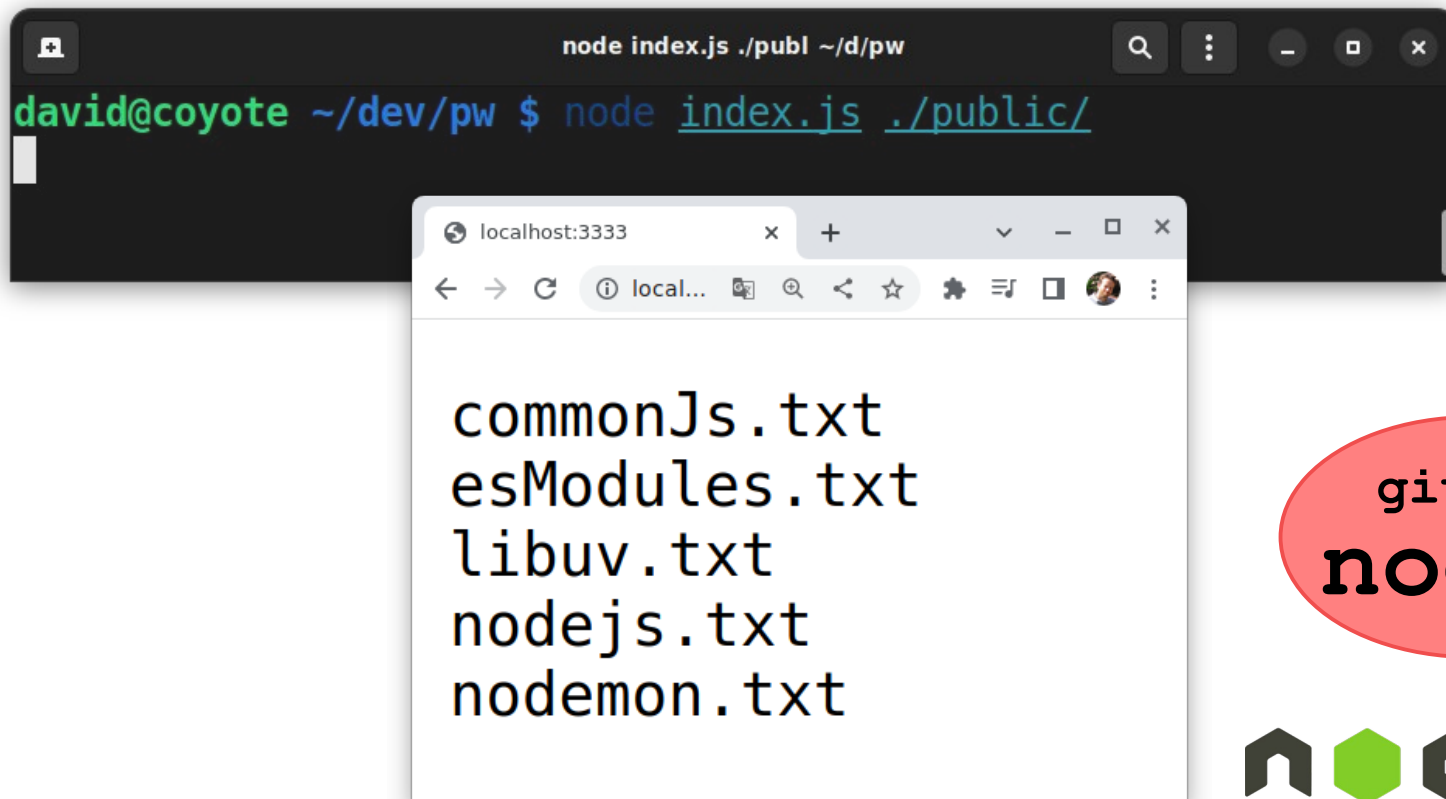
```
process.argv.forEach((val, index) => {  
  console.log(`${index}: ${val}`)  
})
```

A terminal window with a dark background and light text. The title bar shows a window icon, the path ~/d/pw, and standard window controls (search, list, zoom, close). The prompt is david@coyote ~/dev/pw. The command node index.js icomp ufam is entered. The output shows an array of arguments: 0: /usr/bin/node, 1: /home/david/dev/pw/index.js, 2: icomp, 3: ufam. The prompt is now david@coyote ~/dev/pw \$.

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



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:

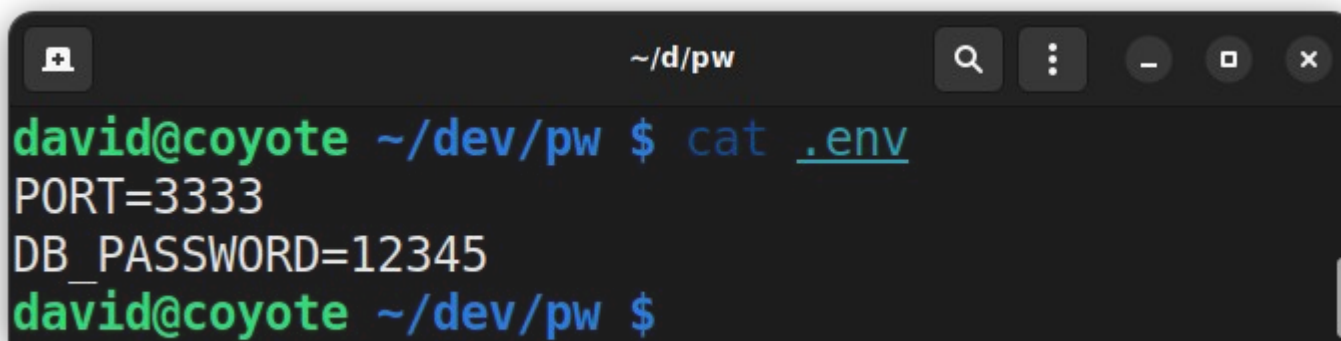
```
process.env.DB_PASSWORD=12345
```

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



Variáveis de Ambiente

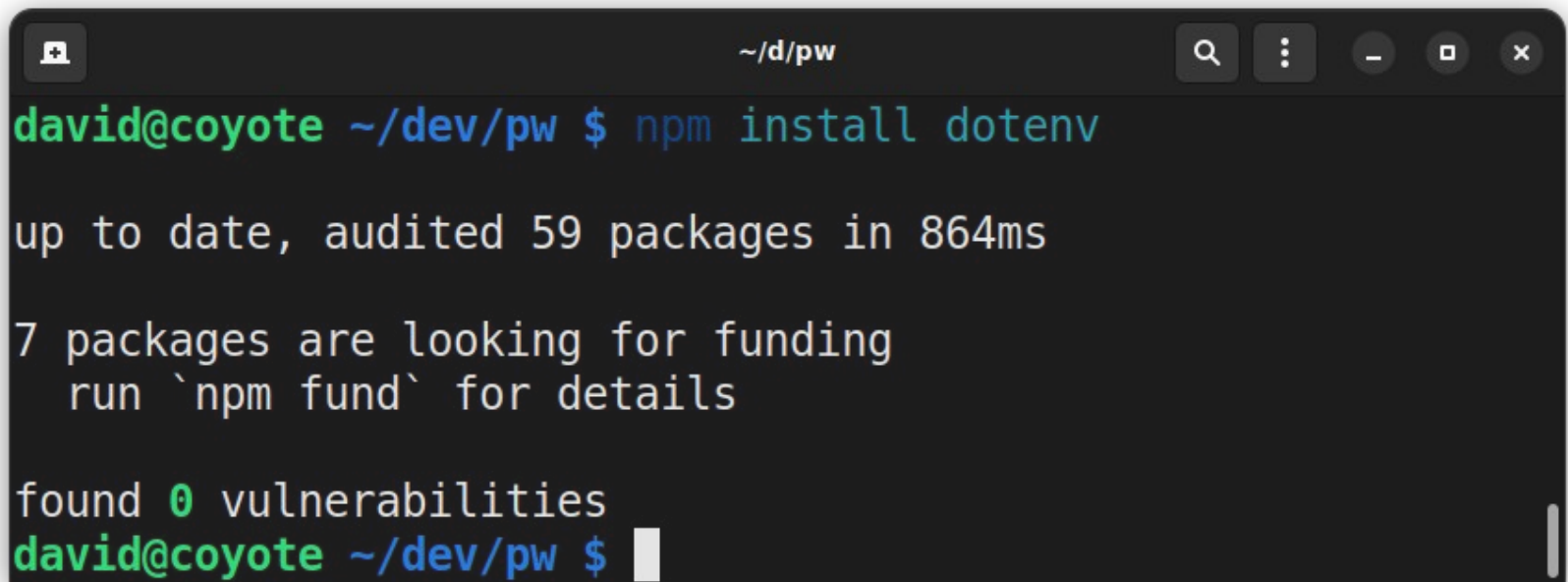
- Uma opção para definir as variáveis de ambiente é através de arquivos não versionados no diretório raiz da aplicação
 - Exemplos de nomes para esses arquivos são **.env**, **.env.development**, **.env.production**

A terminal window with a dark background. The title bar shows a window icon, the path ~/d/pw, and standard window controls (search, list, zoom, close). The prompt is david@coyote ~/dev/pw. The command cat .env is entered, and the output shows PORT=3333 and DB_PASSWORD=12345. The prompt returns to david@coyote ~/dev/pw.

```
~/d/pw
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**

```
# Arquivo .env  
PORT=3333
```

```
const http = require('http');  
require('dotenv').config();  
  
const PORT = process.env.PORT ?? 8080;  
  
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**

```
# Arquivo .env  
PORT=3333
```

```
const http = require('http');  
require('dotenv').config();  
  
const PORT = process.env.PORT ?? 8080;  
  
const server = http.createServer(function (req, res) {
```

Instituto de Computação

```
index.js ~/d/pw  
ev/pw $ node index.js
```

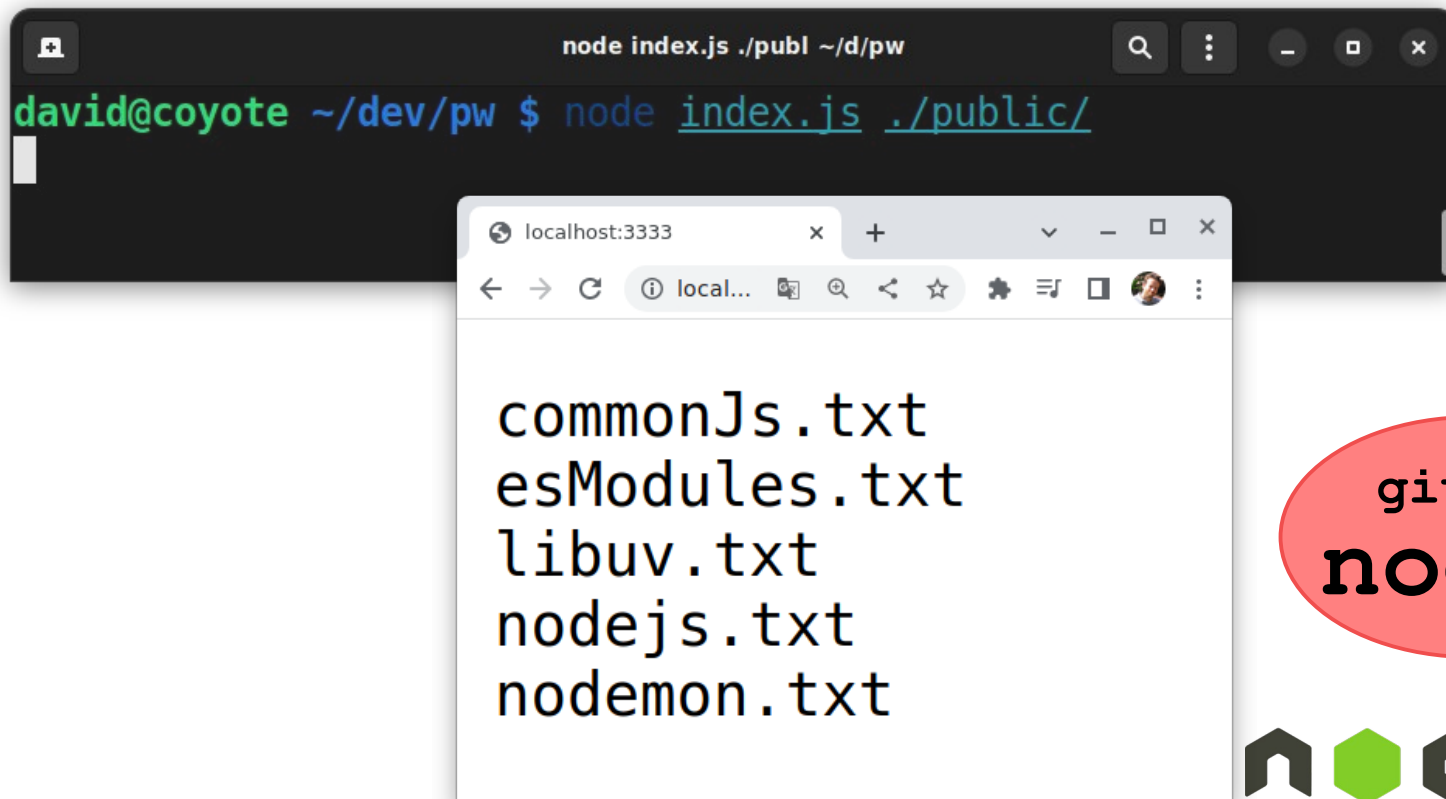
Variáveis de Ambiente

- 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



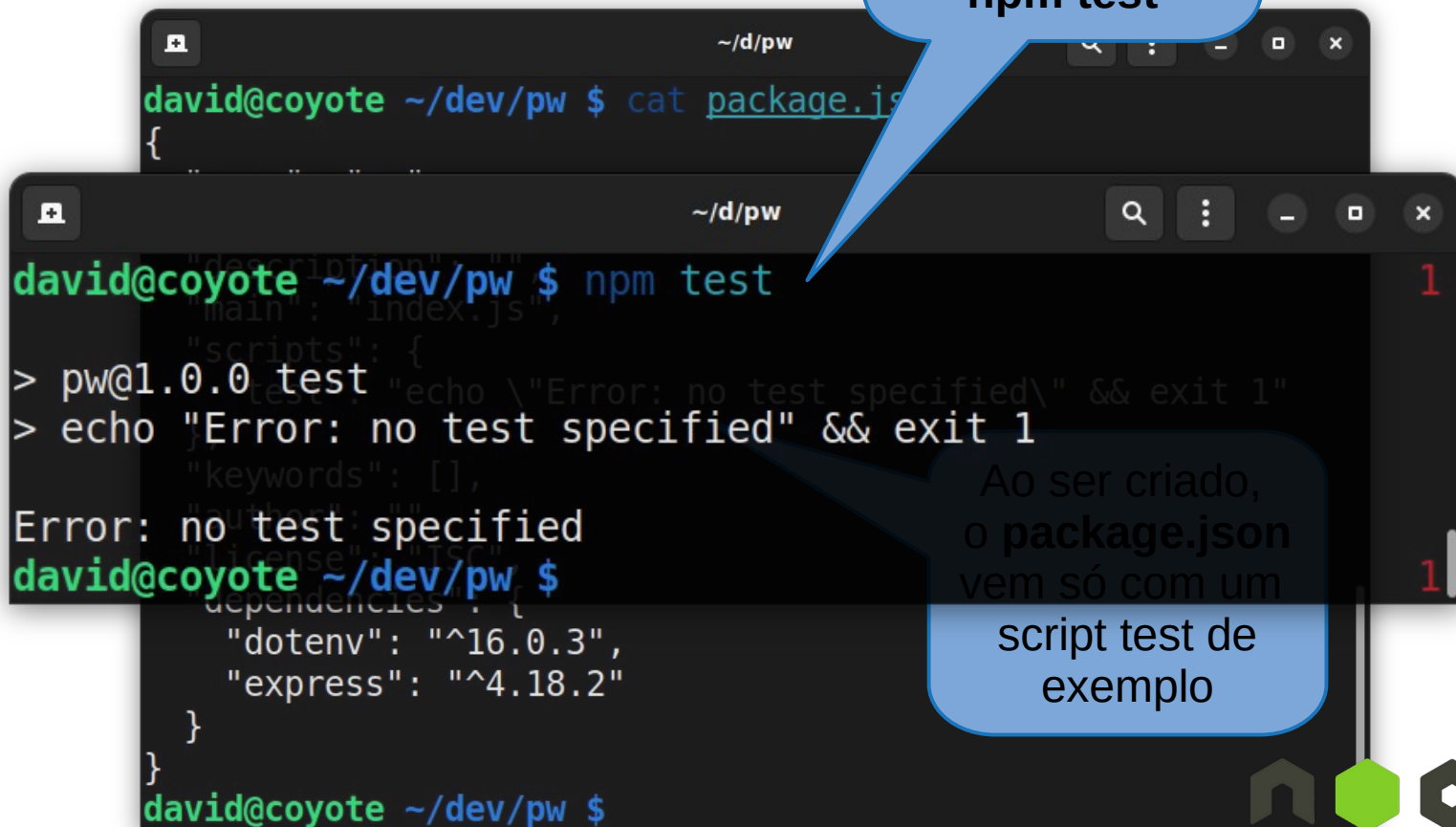
```
~ /d/pw
david@coyote ~/dev/pw $ cat package.json
{
  "name": "pw",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "dotenv": "^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. Para executar o script de exemplo, podemos usar o comando `npm test`.



```
~ /d/pw
david@coyote ~/dev/pw $ cat package.json
{
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  }
}

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 $
```

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. Podemos usar o comando `npm run` para executar um script na app.

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

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.

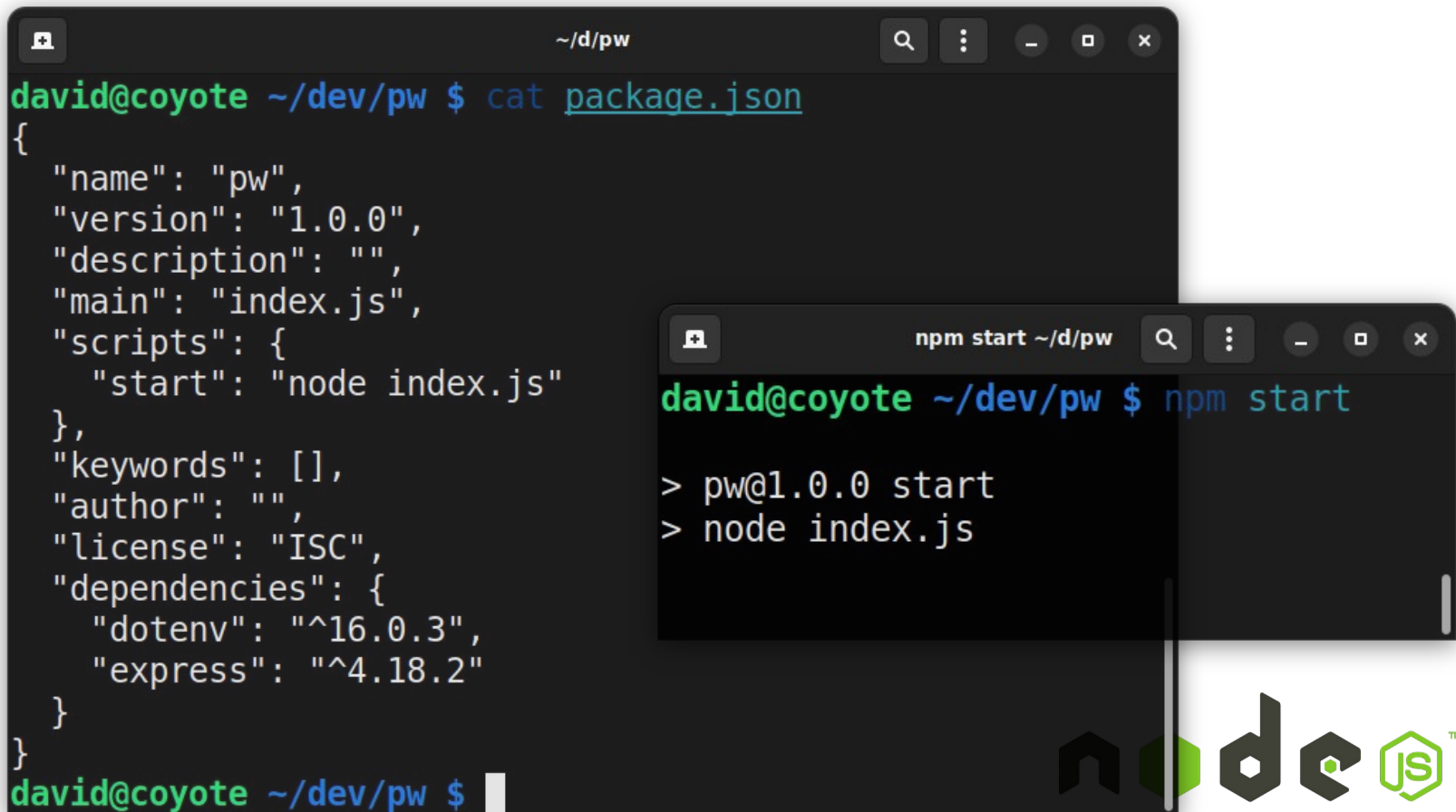
O `package.json` vem só com um script `test` de exemplo

```
~ /d/pw
david@coyote ~/dev/pw $ cat package.json
{
  "name": "coyote",
  "version": "1.0.0",
  "description": "coyote",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "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




The image shows two terminal windows. The background window displays the contents of package.json, which includes a 'scripts' section with a 'start' script. The foreground window shows the execution of 'npm start', which runs the 'start' script defined in package.json.

```
~/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 $

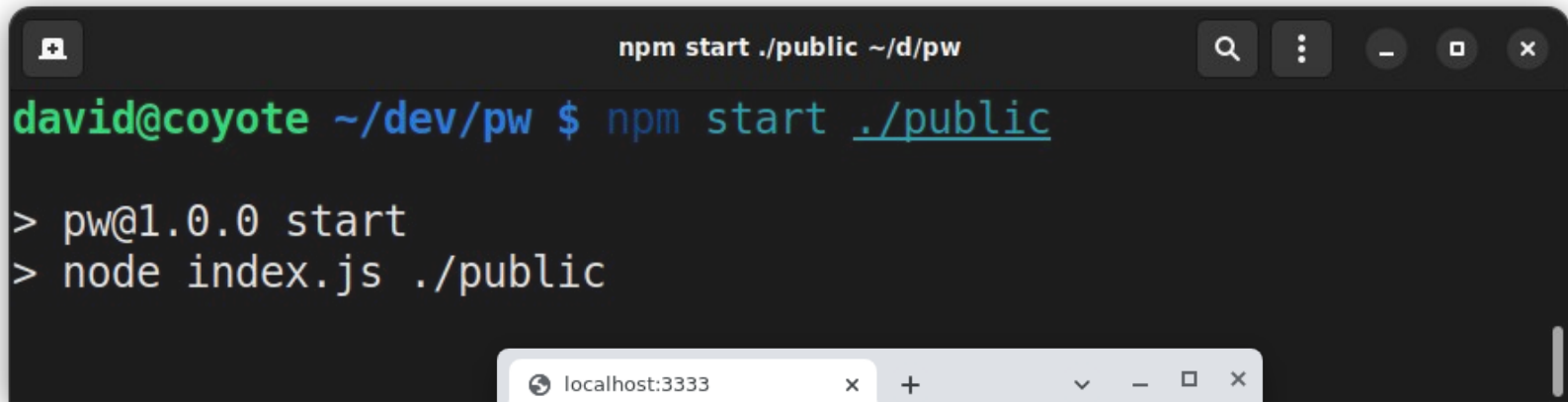
npm start ~/d/pw
david@coyote ~/dev/pw $ npm start

> pw@1.0.0 start
> node index.js
```

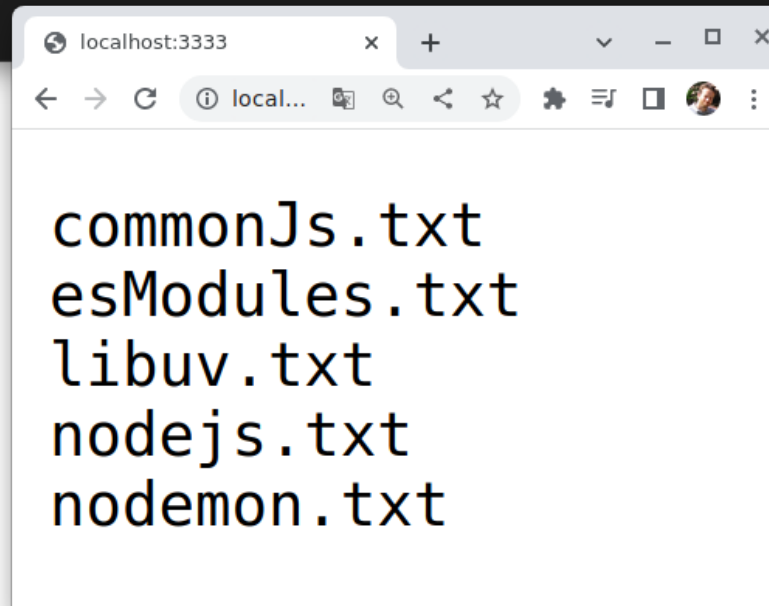


Exercício I – Parte 3

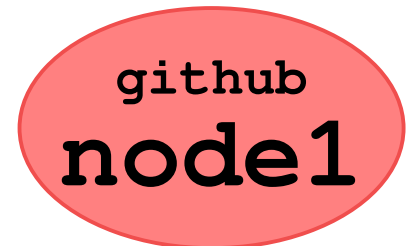
- Crie um script **npm start** para a sua aplicação.



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



```
localhost:3333  
commonJs.txt  
esModules.txt  
libuv.txt  
nodejs.txt  
nodemon.txt
```



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: reinicializar 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

```
~ /d/pw
david@coyote ~/dev/pw $ npm install nodemon --save-dev
added 32 packages, and audited 91 packages in 1s
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 ambientes de produção



Nodemon

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

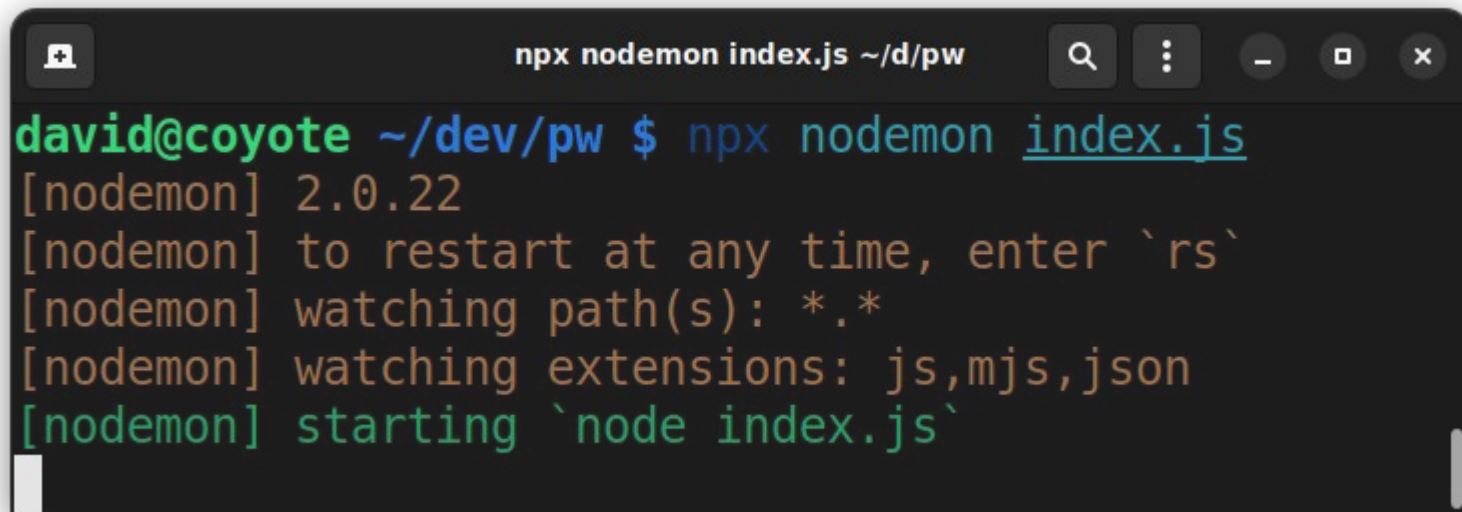
```
~/d/p/n/.bin
david@coyote ~/dev/pw/node_modules/.bin $ ls -la
total 8
drwxrwxr-x  2 david david 4096 mai 14 07:11 ./
drwxrwxr-x 89 david david 4096 mai 14 07:11 ../
lrwxrwxrwx  1 david david  14 mai 11 09:49 mime -> ../mime/cli.js*
lrwxrwxrwx  1 david david  25 mai 14 07:11 nodemon -> ../nodemon/bin/nodemon.js*
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

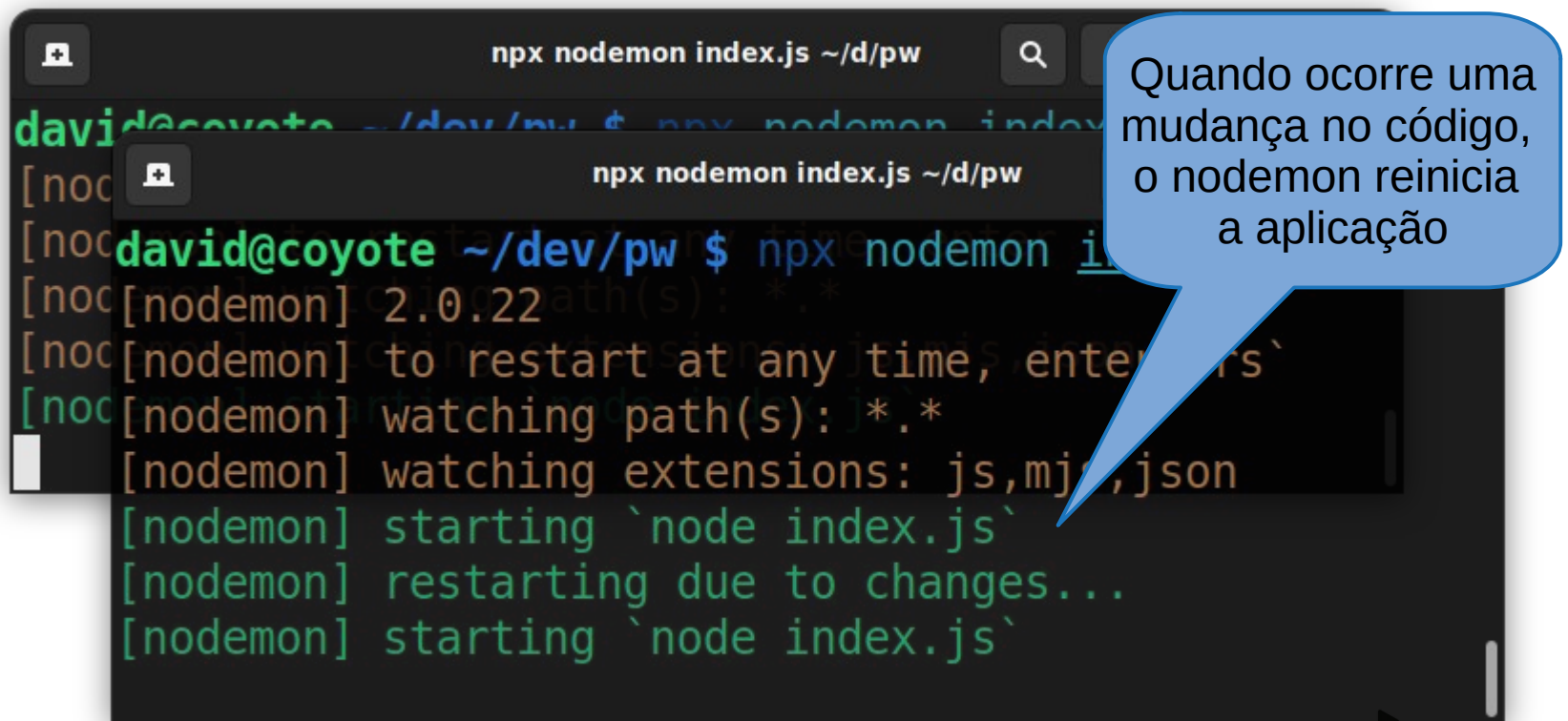
- Para inicializar a aplicação através do nodemon, podemos usar o comando **`npx nodemon index.js`**

A terminal window with a dark background. The title bar at the top reads 'npx nodemon index.js ~/d/pw'. The terminal text shows a user prompt 'david@coyote ~/dev/pw \$' followed by the command 'npx nodemon index.js'. The output shows '[nodemon] 2.0.22', '[nodemon] to restart at any time, enter `rs`', '[nodemon] watching path(s): *.*', '[nodemon] watching extensions: js,mjs,json', and '[nodemon] starting `node index.js`'.

```
npx nodemon index.js ~/d/pw
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**



The image shows a terminal window with the command `npx nodemon index.js ~/d/pw` executed. The output shows the nodemon process starting, watching for changes, and restarting the application. A blue speech bubble points to the terminal output, explaining that nodemon restarts the application when there is a change in the code.

```
npx nodemon index.js ~/d/pw
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] restarting due to changes...
[nodemon] starting `node index.js`
```

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

Nodemon

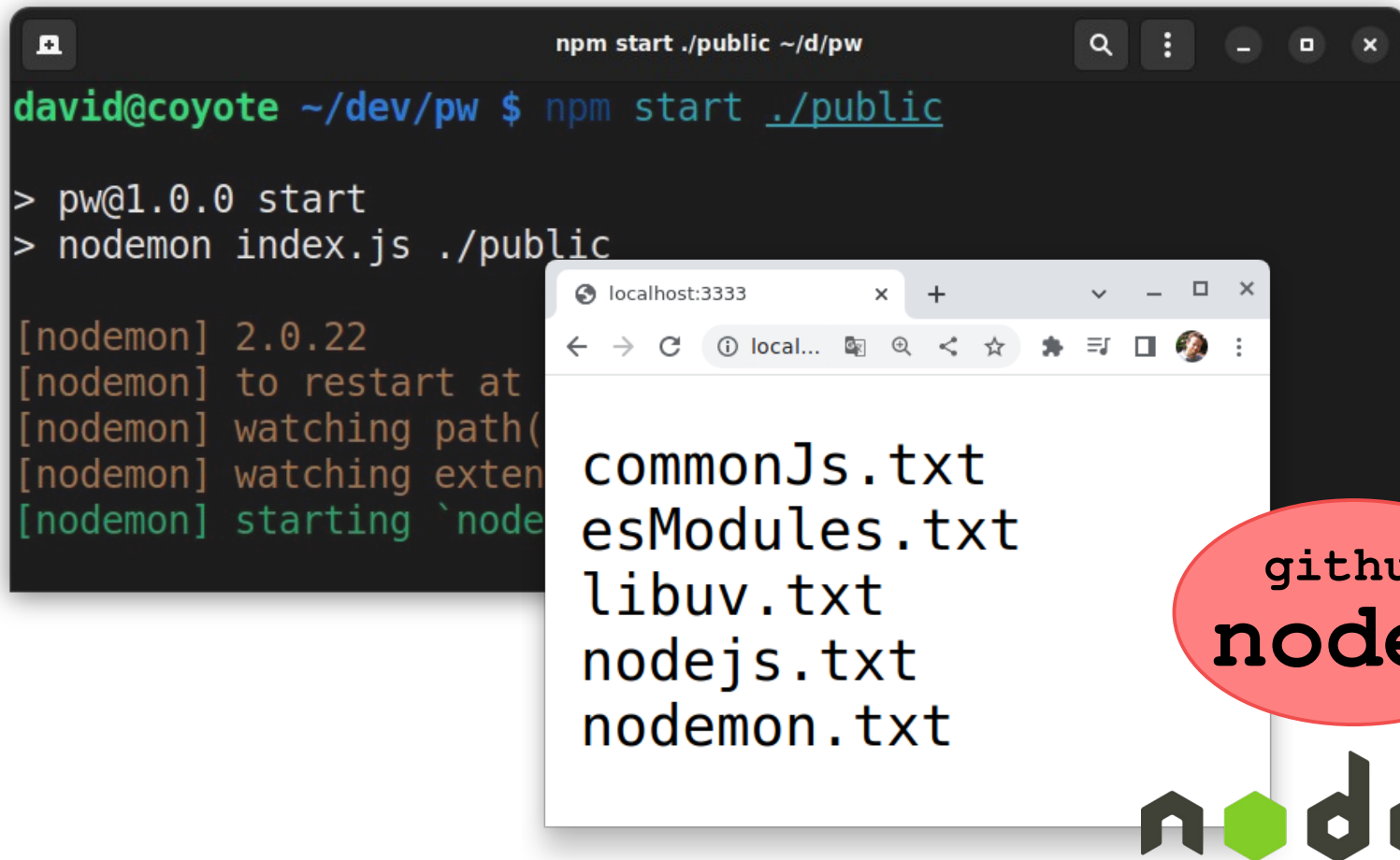
- Podemos editar o **package.json** e criar um script para ambiente de desenvolvimento e outro para produção

```
{
  "name": "pw",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "npm run dev",
    "start:prod": "node src/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



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

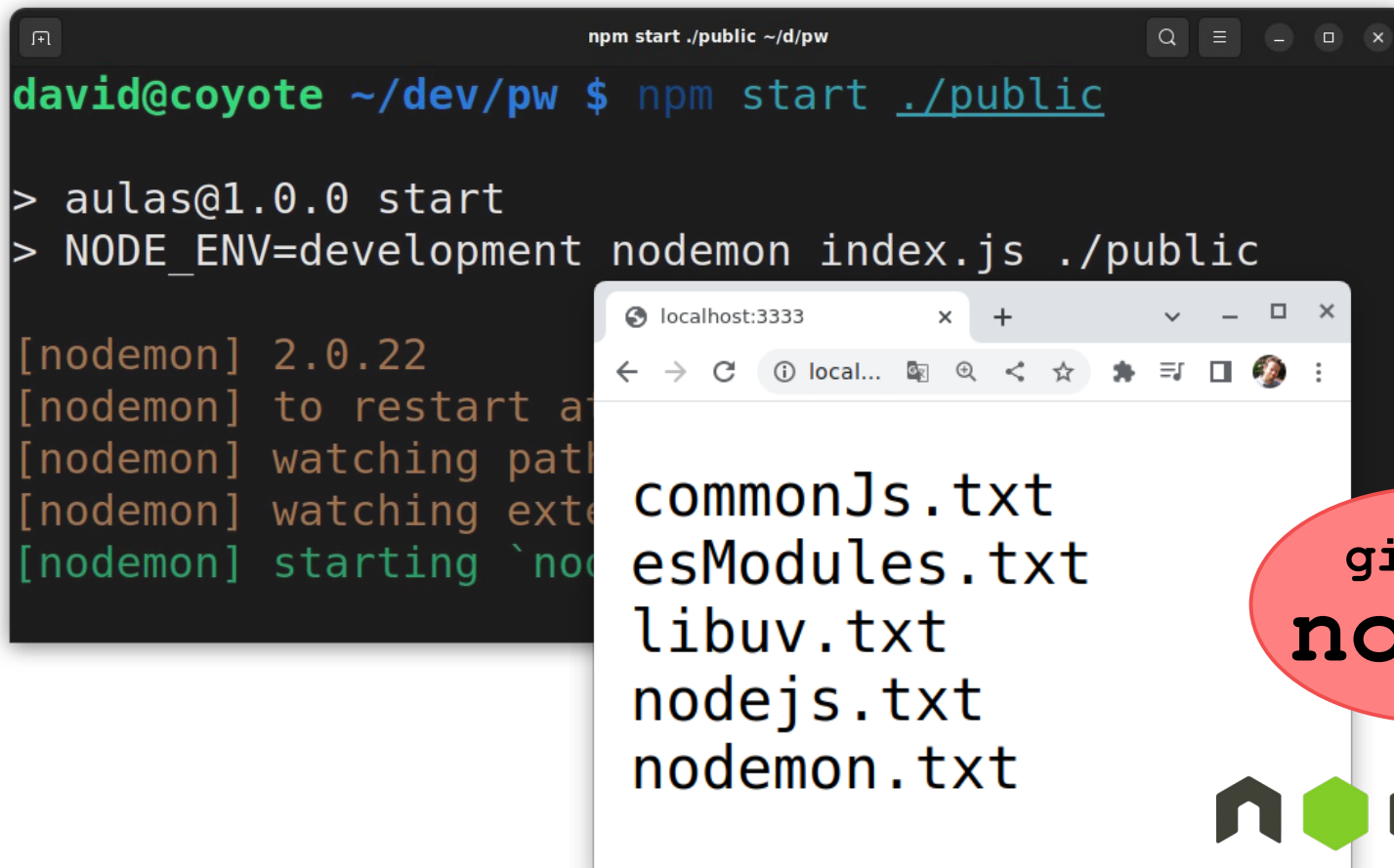
- Para selecionar o arquivo de configurações correto de acordo com o seu ambiente, podemos usar o seguinte código:

```
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



The image shows a terminal window and a web browser window. The terminal window displays the command `npm start ./public` being executed, which runs the script `aulas@1.0.0 start` using `NODE_ENV=development nodemon index.js ./public`. The terminal output shows Nodemon 2.0.22 watching for file changes and starting the application. The browser window shows the local host `localhost:3333` displaying a list of files: `commonJs.txt`, `esModules.txt`, `libuv.txt`, `nodejs.txt`, and `nodemon.txt`.

```
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 a
[nodemon] watching path
[nodemon] watching exte
[nodemon] starting `no
```

localhost:3333

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

github
node1

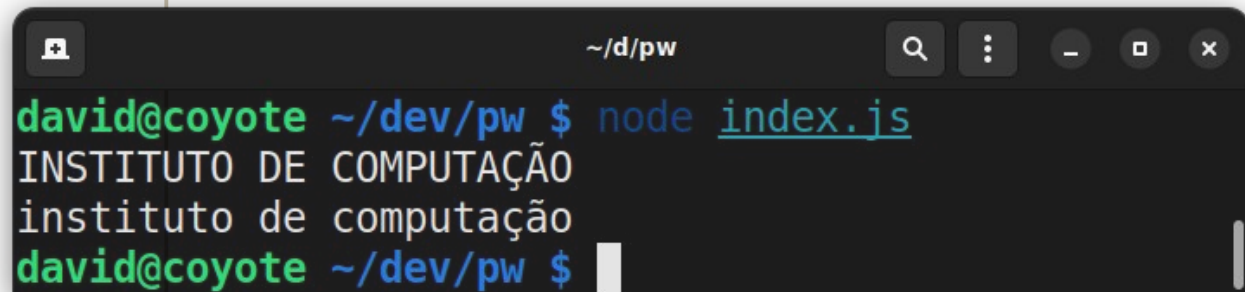
node JS

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 text. The title bar shows the path ~/d/pw. The prompt is david@coyote. The command node index.js has been executed, resulting in two lines of output: INSTITUTO DE COMPUTAÇÃO and instituto de computação. The prompt is now david@coyote ~/dev/pw \$.

```
~/d/pw  
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";
```

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

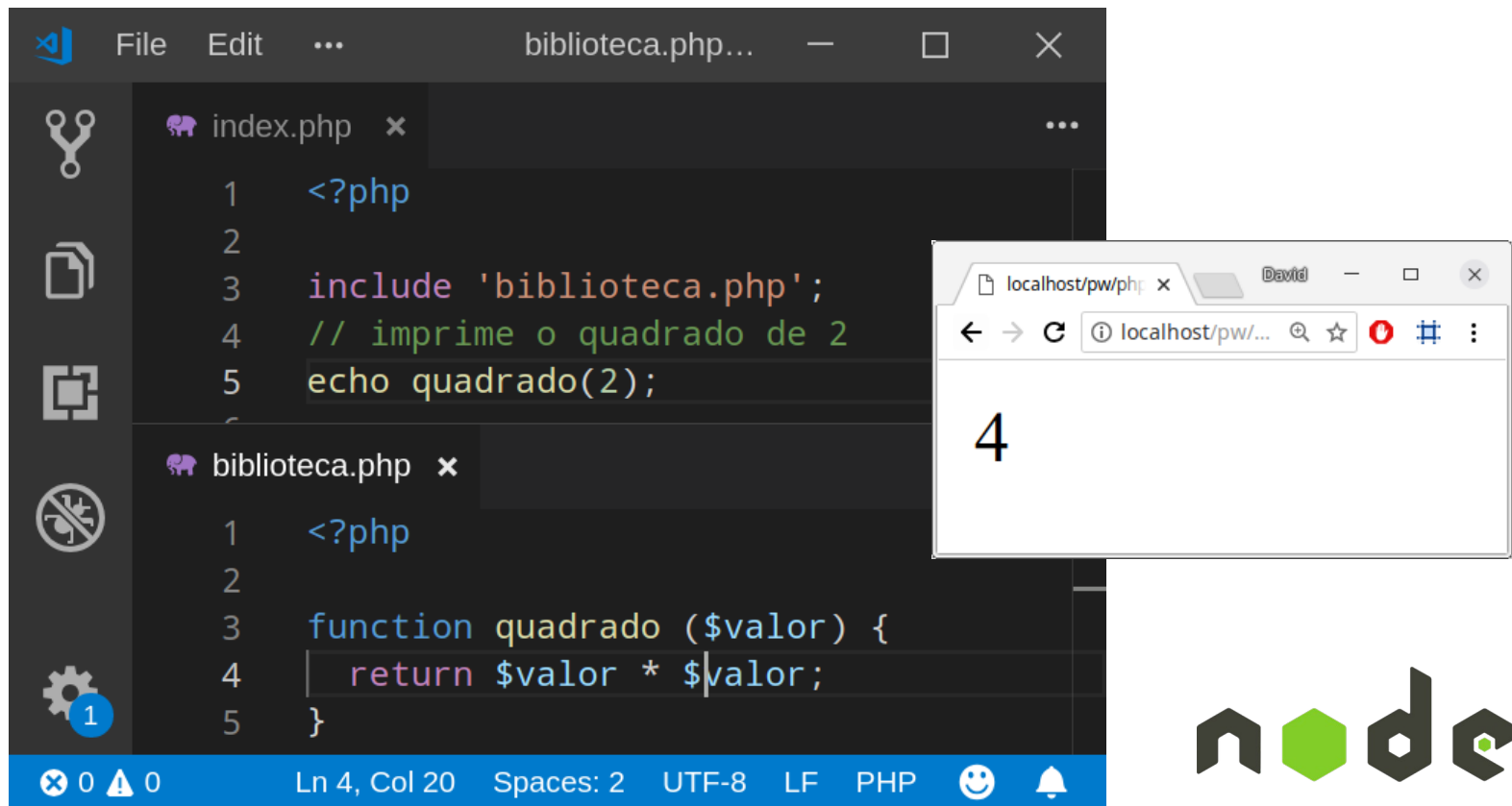
Note que é preciso usar o objeto **module.exports** para definir que variáveis e funções poderão ser acessadas ou executadas por quem importar o módulo.

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



Módulos do NodeJS

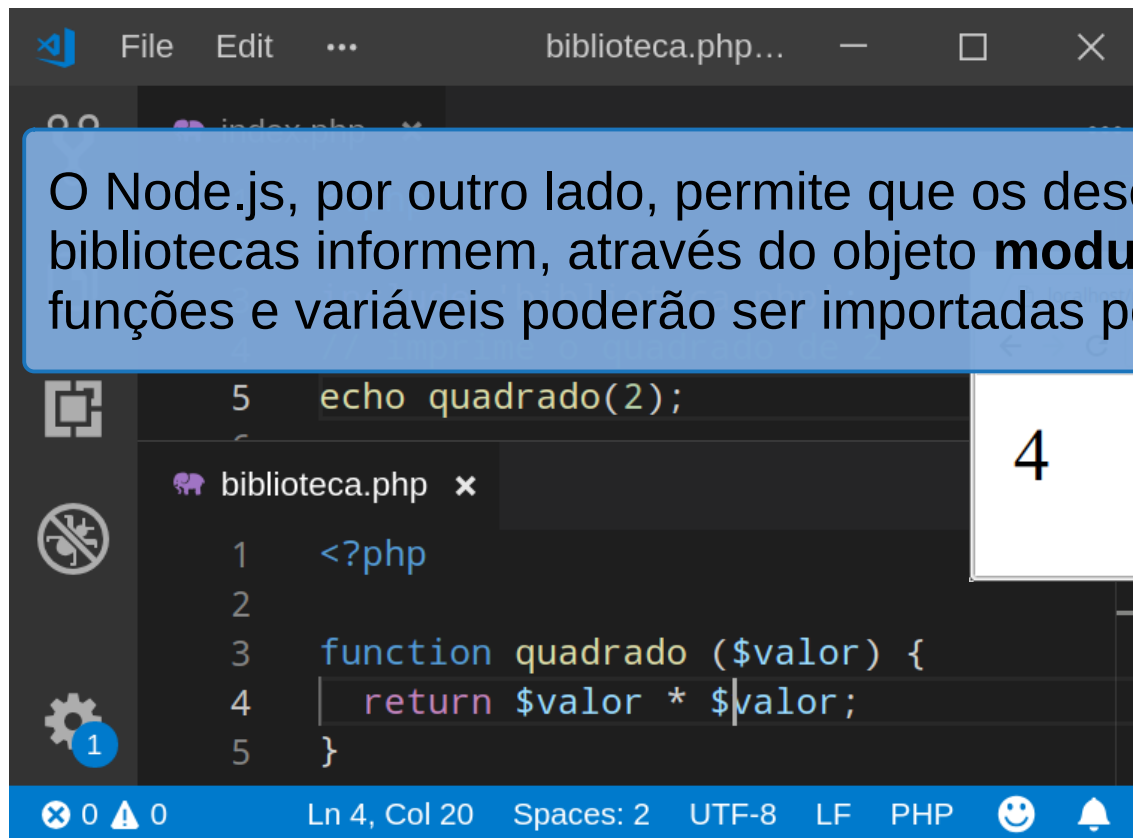
- Em linguagens como PHP e Ruby, todas as variáveis e funções declaradas nas bibliotecas passam a pertencer ao escopo global das aplicações que as importam



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



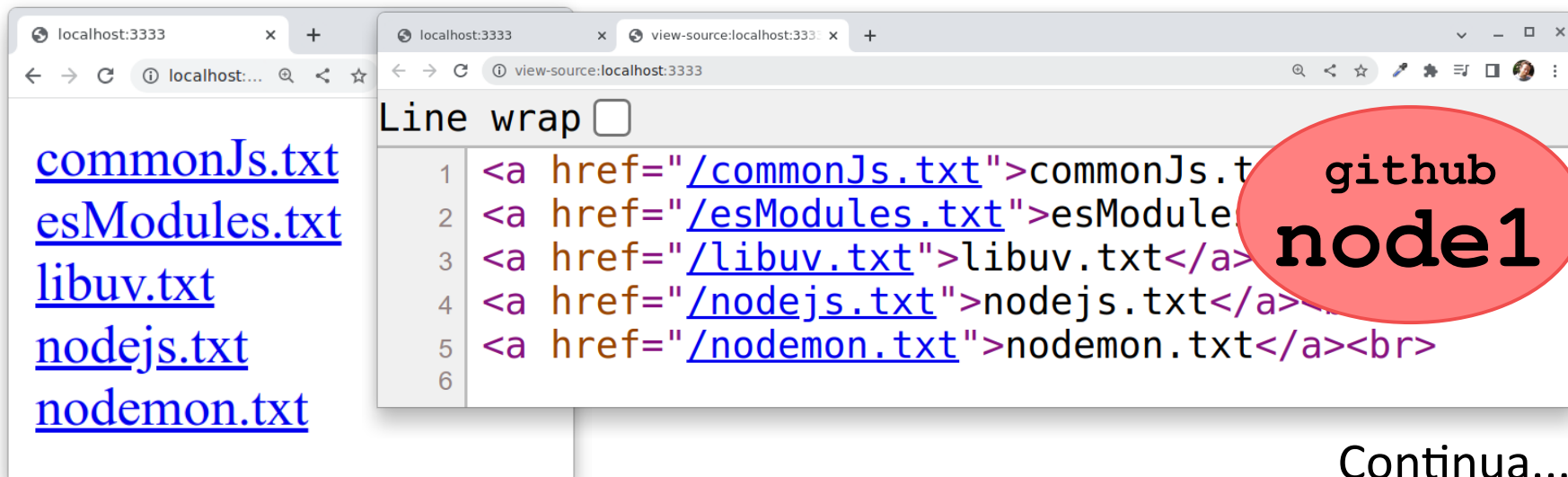
```
File Edit ... biblioteca.php...  
biblioteca.php x  
1 <?php  
2  
3 function quadrado ($valor) {  
4     return $valor * $valor;  
5 }  
5 echo quadrado(2);  
4
```



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 util.js separado.

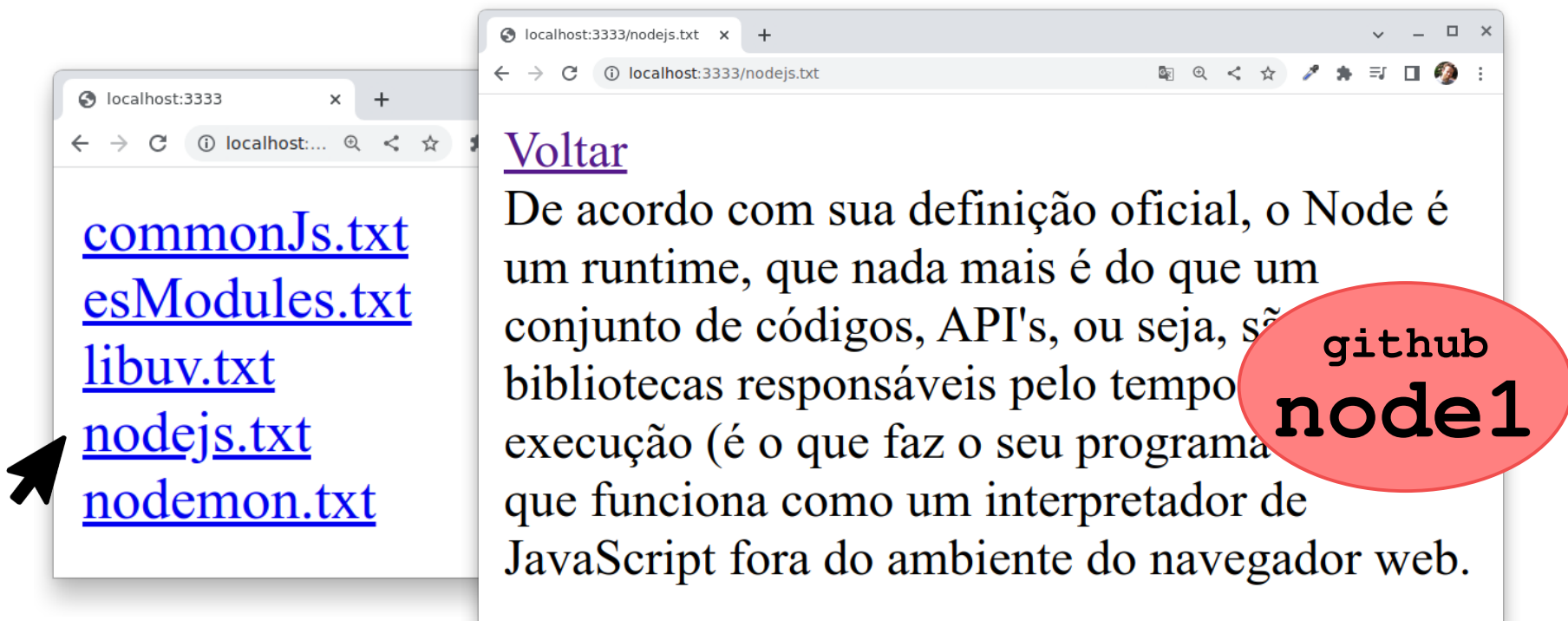
```
function createLink(filename) {  
  return `}
```



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.



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 modularização 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**



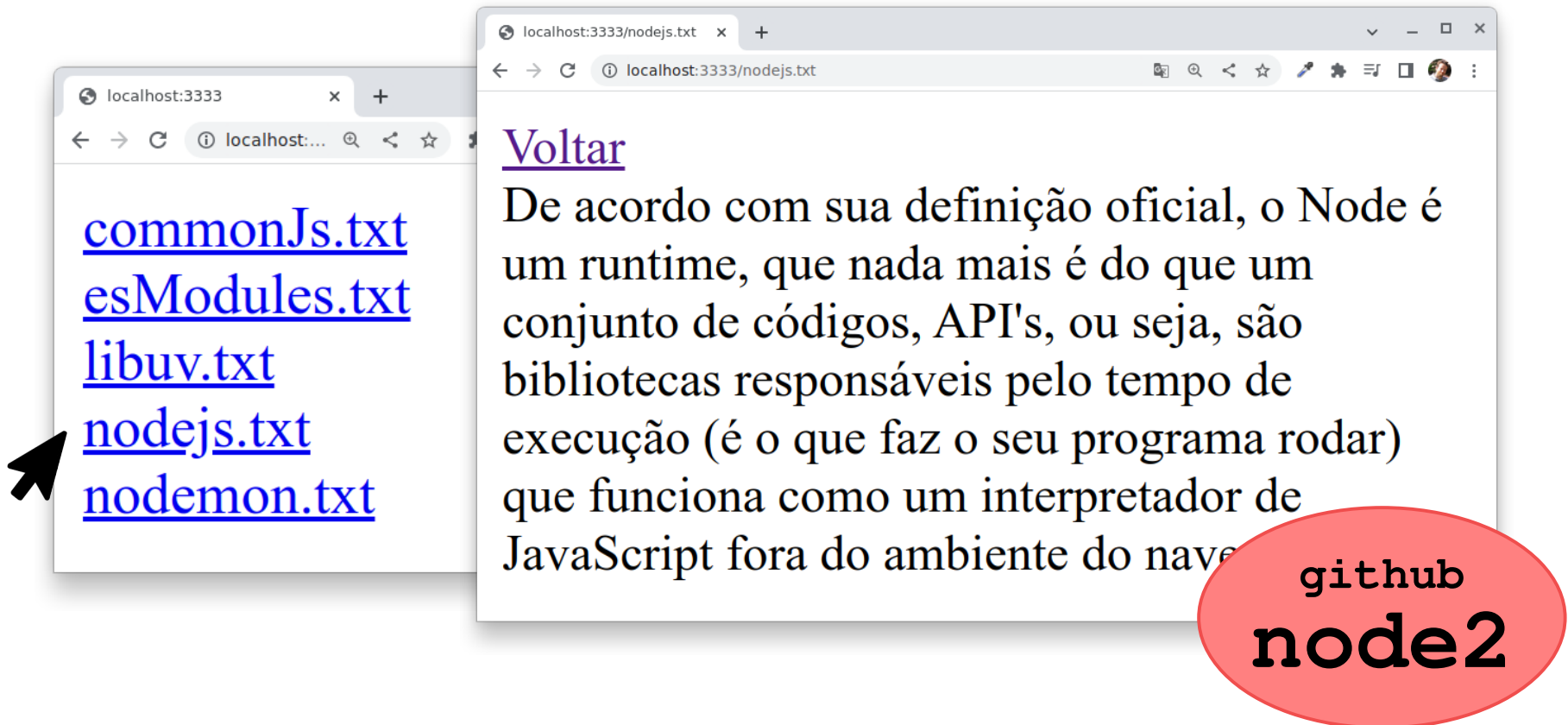
CommonJs vs ES modules

- 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",  
  // ...  
}
```

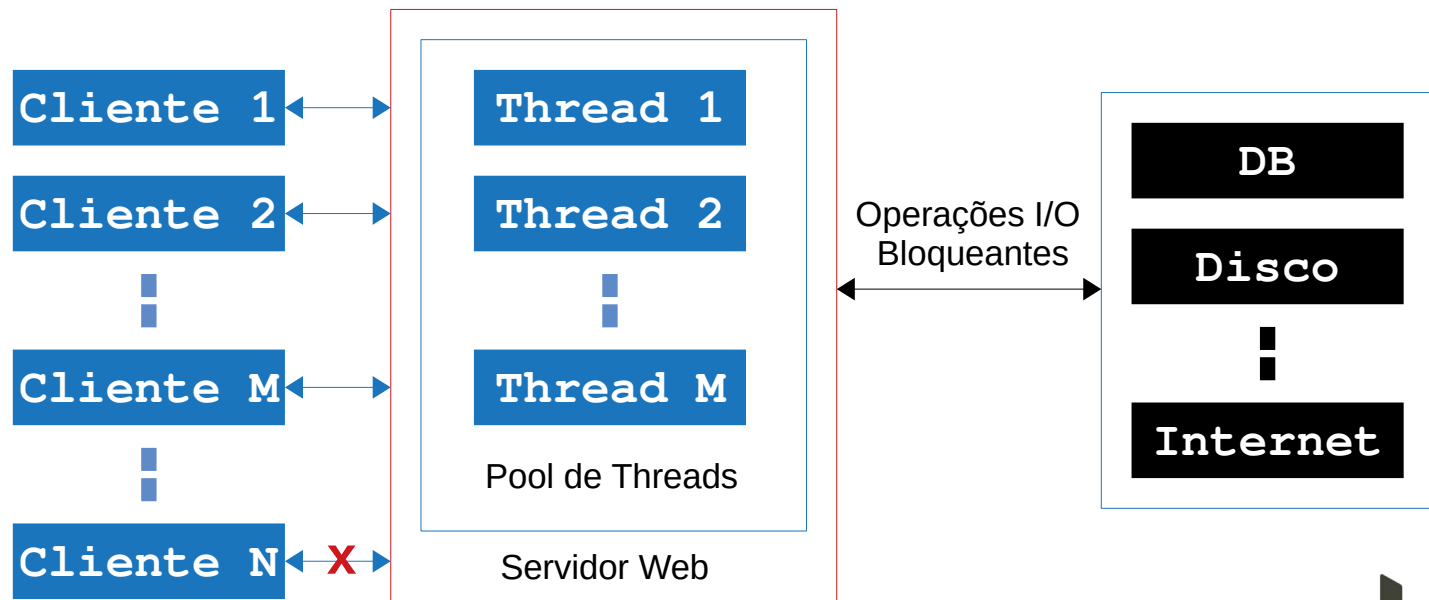
Exercício II

- Refaça o exercício anterior usando ES modules.



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

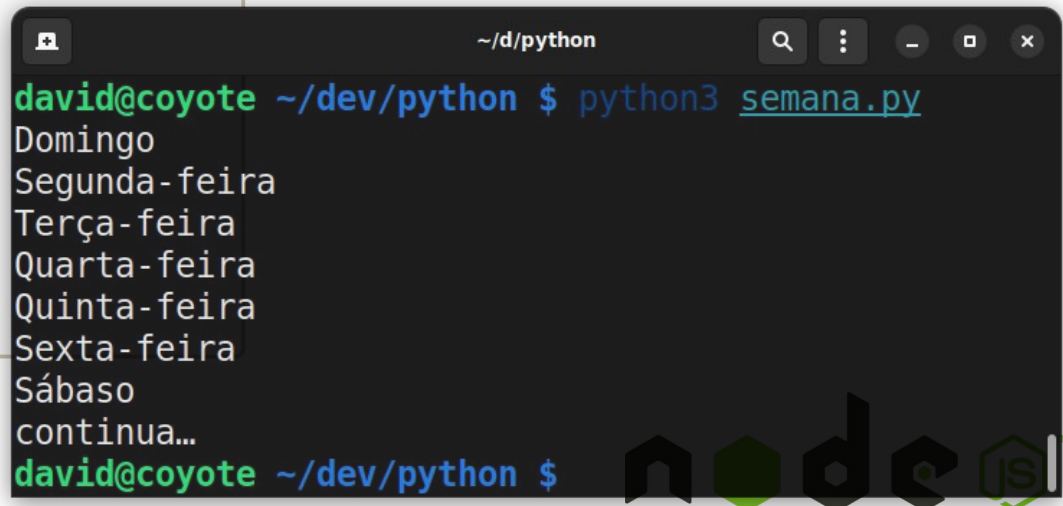
```
#!/usr/bin/python3
```

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

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

```
dias.close()
```

Operação
de I/O
bloqueante



A terminal window titled '~d/python' showing the execution of a Python script. The prompt is 'david@coyote ~/dev/python \$'. The command 'python3 semana.py' has been executed. The output of the script is displayed line by line: 'Domingo', 'Segunda-feira', 'Terça-feira', 'Quarta-feira', 'Quinta-feira', 'Sexta-feira', 'Sábado', and 'continua...'. The prompt is now 'david@coyote ~/dev/python \$'. At the bottom right of the terminal window, there is a logo for 'nodejs'.

```
~/d/python  
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

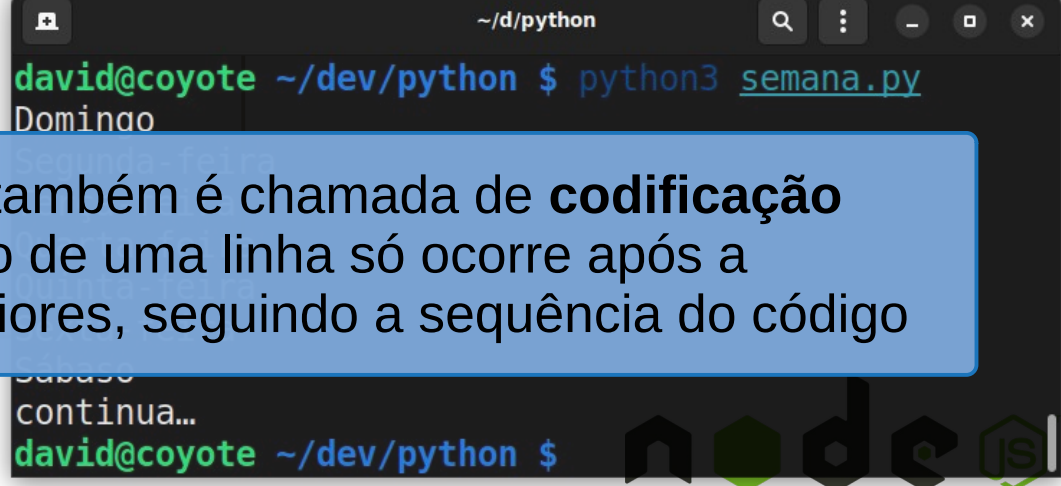
- Nessa abordagem multi-thread tradicional, todas as operações de I/O são **bloqueantes**
 - Por exemplo, no programa Python abaixo, todo o código deve esperar a leitura de arquivo que ocorre na linha 3

```
#!/usr/bin/python3
```

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

```
print (conteudo)
```

Operação
de I/O
bloqueante



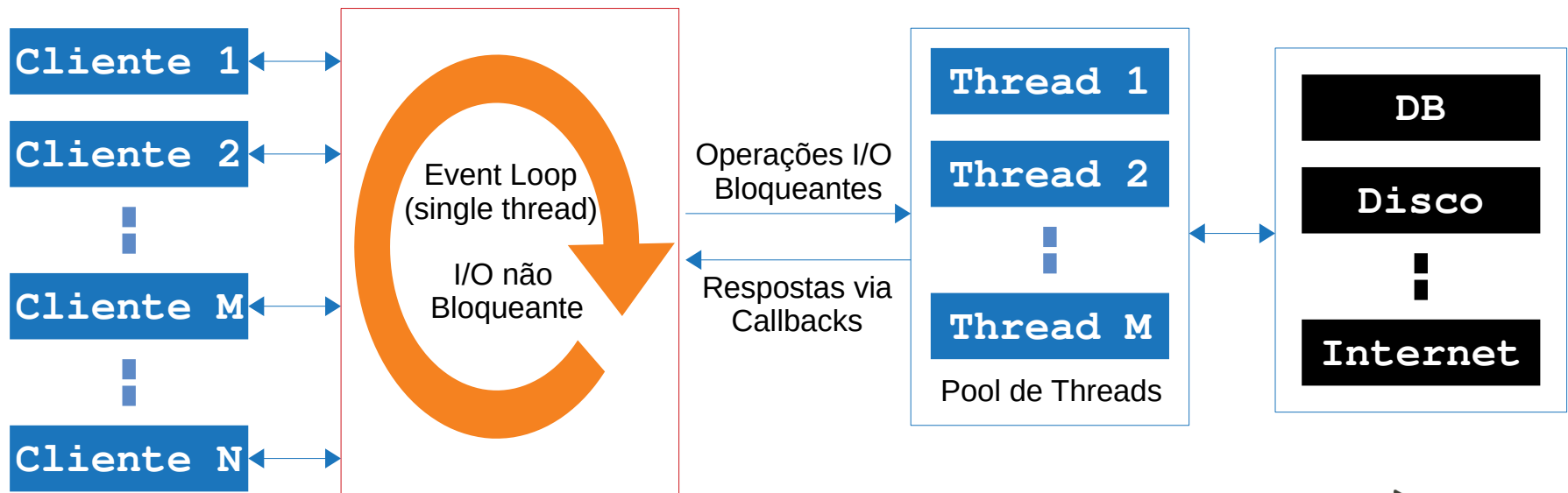
A terminal window titled '~d/python' showing the execution of a Python script. The prompt is 'david@coyote ~/dev/python \$'. The command 'python3 semana.py' has been executed. The output shows 'Domingo' on the first line and 'continua...' on the second line. The prompt is now 'david@coyote ~/dev/python \$'.

```
~d/python  
david@coyote ~/dev/python $ python3 semana.py  
Domingo  
continua...  
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

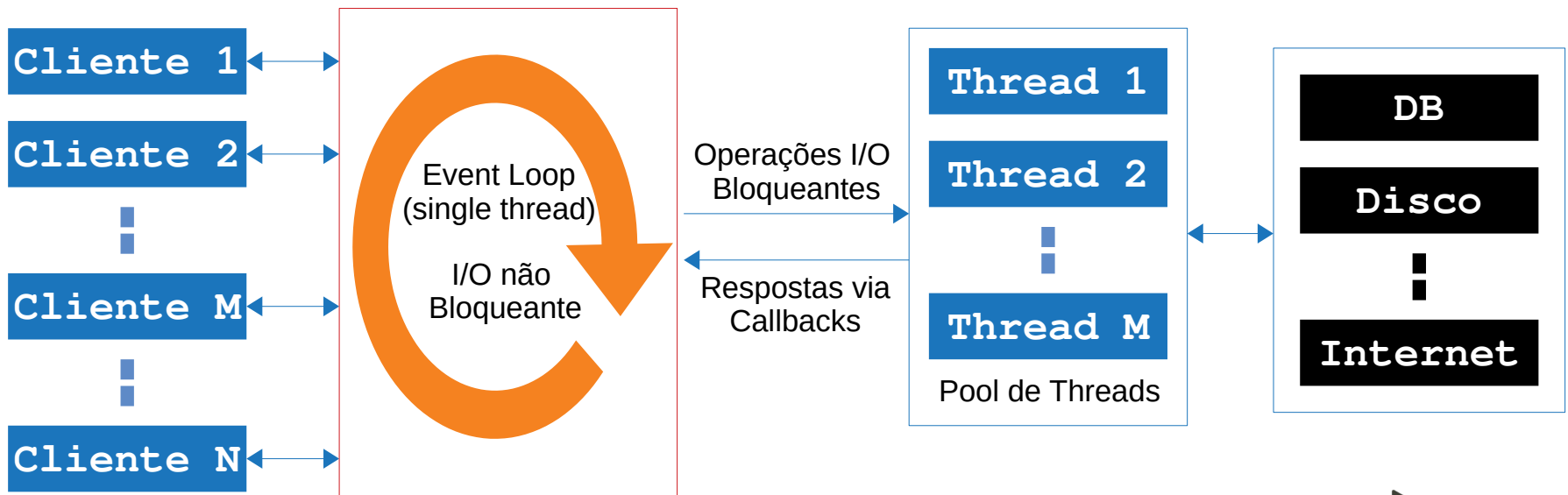
Single Thread Event Loop

- 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

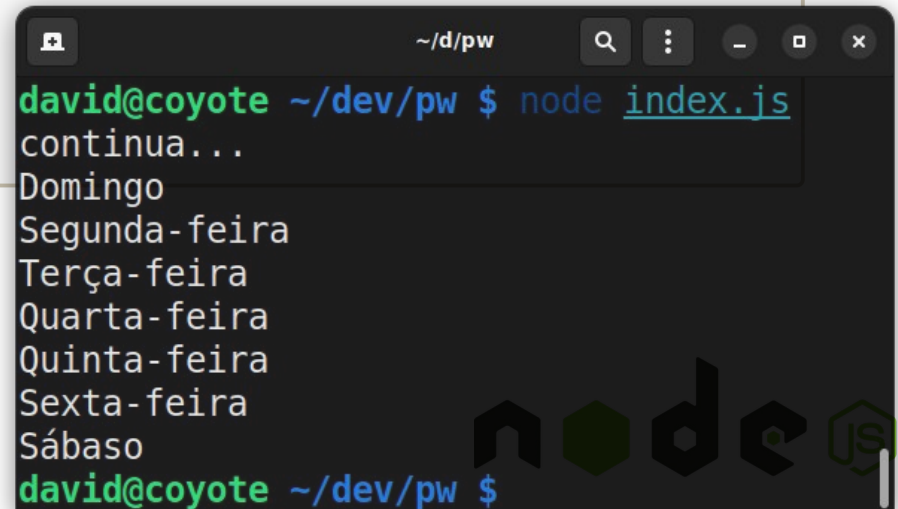
- No Node.js, apenas uma thread responde por todas as requisições dos usuários – essa thread é chamada de Event Loop
- O Node.js é um ambiente de execução **single thread**, que no background usa múltiplas threads para executar códigos bloqueantes de I/O



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...");
```



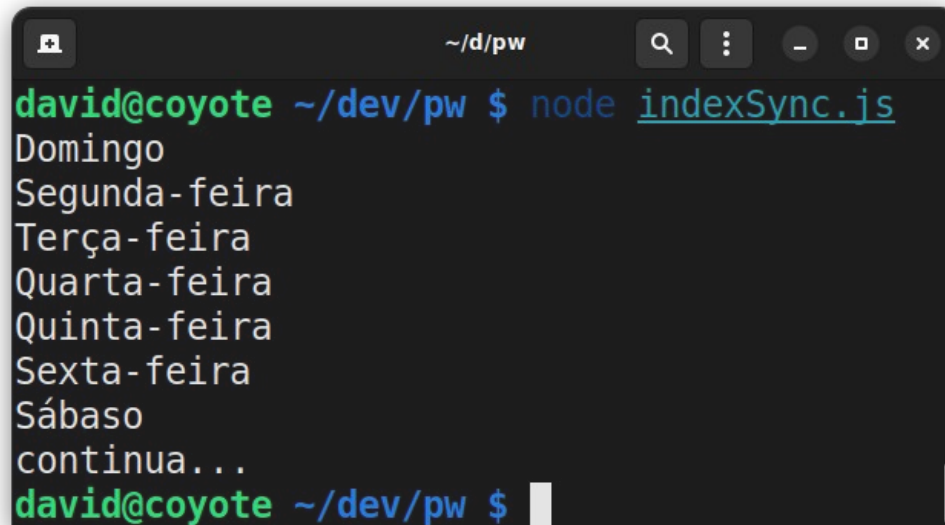
A terminal window with a dark background and light-colored text. The window title is '~d/pw'. The prompt is 'david@coyote ~/dev/pw \$'. The command 'node index.js' has been executed. The output of the script is displayed line by line: 'continua...', 'Domingo', 'Segunda-feira', 'Terça-feira', 'Quarta-feira', 'Quinta-feira', 'Sexta-feira', and 'Sábado'. The prompt 'david@coyote ~/dev/pw \$' is visible at the bottom.

```
~/d/pw  
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 with a dark background. The title bar shows the path ~/d/pw and standard window controls. The prompt is david@coyote ~/dev/pw. The command node indexSync.js has been executed. The output is a list of days of the week: Domingo, Segunda-feira, Terça-feira, Quarta-feira, Quinta-feira, Sexta-feira, Sábado, and continua... The prompt is now david@coyote ~/dev/pw \$.

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




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 pela comunidade de **callback hell** ou **código hadouken**

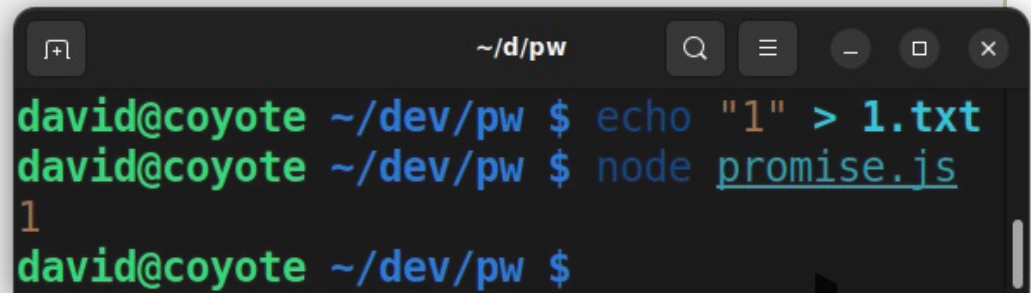
Promises

- 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 with a dark background and light text. The title bar shows the path ~/d/pw. The prompt is david@coyote. The first command is echo "1" > 1.txt, which creates a file named 1.txt containing the number 1. The second command is node promise.js, which runs the JavaScript file. The output is 1, which is the value resolved by the promise in the code block to the left. The prompt returns to david@coyote ~~/dev/pw \$.

```
~/d/pw
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 promissas 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 with a dark background. The title bar shows the path ~/d/pw. The prompt is david@coyote. The command node promise.js has been executed. The output shows two lines: 1 and 2, each on a new line. The prompt is now david@coyote ~/dev/pw \$.



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

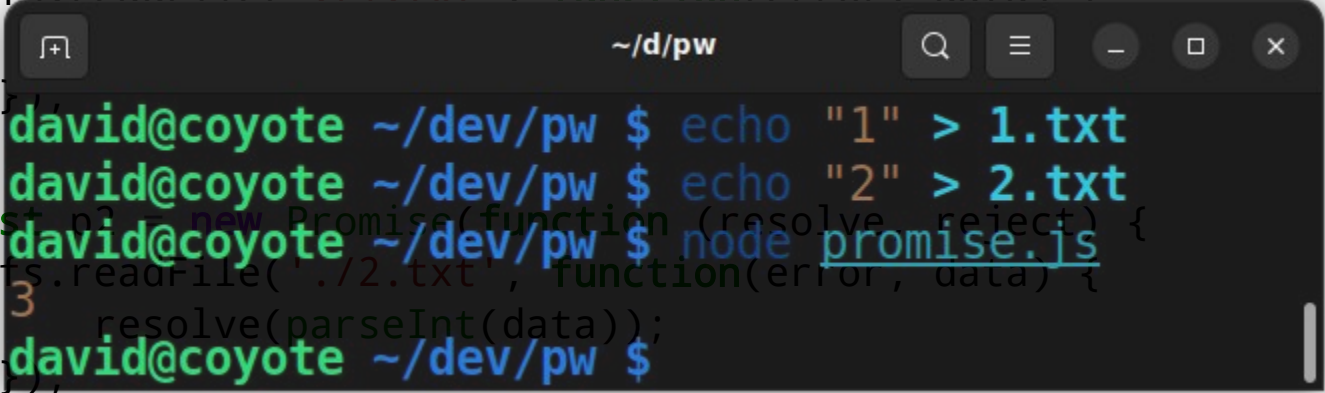
- O método estático **Promise.all()** pode ser usado para aguardar a resolução de um conjunto de **promises**

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

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

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

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



The terminal screenshot shows the following commands and output:

```
~/d/pw
david@coyote ~/dev/pw $ echo "1" > 1.txt
david@coyote ~/dev/pw $ echo "2" > 2.txt
david@coyote ~/dev/pw $ node promise.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 **Promise.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 **Promise.then()** são os modificadores **async** e **await**


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

```
function readFile (filename) {
```

```
  return new Promise((resolve, reject) => {  
    fs.readFile(filename, 'utf8', (err, data) => {  
      if (err) reject(err);  
      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');
```



```
~ /d/pw  
david@coyote ~/dev/pw $ node asyncawait.js
```

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) {
```



```
}  
}  
b  
async function calculaValor() {  
  const valor1 = await readFile("1.txt")  
  const valor2 = await readFile("2.txt")  
  return valor1 + valor2  
}  
david@coyote ~/dev/pw $
```

```
console.log('a')
```

```
console.log('b')
```

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

```
console.log('c')
```

