



# Fundamentos de **NODE**

Node.js

# Objetivos de Aprendizagem

---

**Ao final deste módulo,  
esperamos que você seja capaz de:**

- Entender os fundamentos de NodeJS.
- Conhecer a diferença entre front-end e back-end.
- Executar seus primeiros códigos com JavaScript/Node.
- Entender os fundamentos de estilos arquitetônicos como REST e SOAP.
- Identificar a arquitetura do Node.



# INSTA- LAÇÃO

<https://nodejs.org/es/>



➤ Executar:

node -v

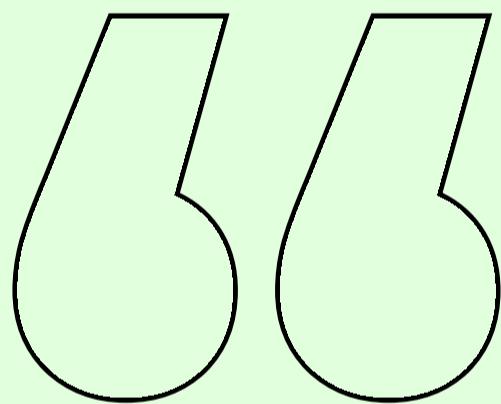
npm -v





# Introdução

**Node.js é um ambiente de execução JavaScript open source e multiplataforma.** É uma ferramenta popular para quase todo tipo de projeto. Node.js roda na engine (motor) V8, o coração do Google Chrome, fora do navegador. Isso permite que o Node.js seja muito performático.



*Uma aplicação Node.js roda em um único processo, sem criar uma nova thread para cada requisição. O Node.js provê uma série de primitivas assíncronas para I/O (input/output) em sua biblioteca nativa que previnem códigos JavaScript bloqueantes, e geralmente, bibliotecas em Node.js são escritas usando como padrão paradigmas não-bloqueantes, fazendo com que o comportamento de bloqueio seja uma exceção à regra.*

Quando o Node.js executa operações de I/O, como ler dados da rede, acessar um banco de dados ou o sistema de arquivos, em vez de bloquear a thread em execução e gastar ciclos de CPU esperando, o Node.js vai continuar com as operações quando a resposta retornar.

**Isso permite com que o Node.js lide com centenas de conexões paralelas em um único servidor.**



---

# Exemplo de uma aplicação Node - **Hello Word**

# Código em **NodeJS**:

```
const http = require('http')
const hostname =
'127.0.0.1'
const port = 3000
const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World\n')
})
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`)
})
```

- **Para executar esse trecho,**  
salve-o como server.js e rode com  
node server.js no seu terminal.

# Explicando o código:

- > Primeiro esse código importa o módulo http .
- > O método **createServer()** do http cria um novo servidor HTTP e o retorna.
- > É definido para o servidor, escutar em uma porta **port** e host name **host** específicos.
- > **Quando** o servidor **está pronto**, a função callback é chamada, nesse caso nos informando que o servidor está rodando.
- > Sempre que uma nova requisição é recebida, o evento de **request** é chamado, provendo dois objetos: uma requisição (objeto do tipo **http.IncomingMessage** ) e uma resposta (objeto do tipo **http.ServerResponse** ).
- > Esses 2 objetos são essenciais para manusear a chamada HTTP.



- O primeiro provê os detalhes da requisição. Nesse simples exemplo, ele não é utilizado, mas com ele você pode acessar os dados da **request** e as **headers**.
- O segundo é usado para retornar dados para quem o chamou.
- Nesse caso:

```
res.statusCode = 200
```

- Definimos a propriedade statusCode como 200, para indicar uma resposta bem sucedida.
- Definimos a **header** de **Content-Type** (tipo de conteúdo):

```
res.setHeader('Content-Type', 'text/plain')
```

- E fechamos a resposta, adicionando o conteúdo como um argumento do **end()**:

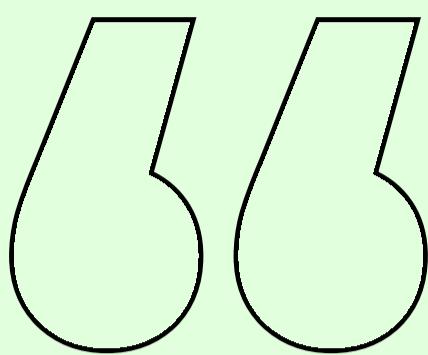
```
res.end('Hello World\n')
```



---

---

# Definição de **back-end** e **front-end**



*De uma forma simplista, podemos pensar nesses dois termos da tecnologia como um espetáculo de teatro, onde temos os bastidores, que aqui seria o back-end, e o palco onde o show acontece, que seria o front-end.*

Ou seja, o back-end resume-se a tudo o que está por trás do site, que o usuário/leitor não tem acesso direto e não consegue interagir. Já o front-end corresponde à parte em que os usuários conseguem ver e interagir, como cores, fontes, menus, imagens entre outras funcionalidades.

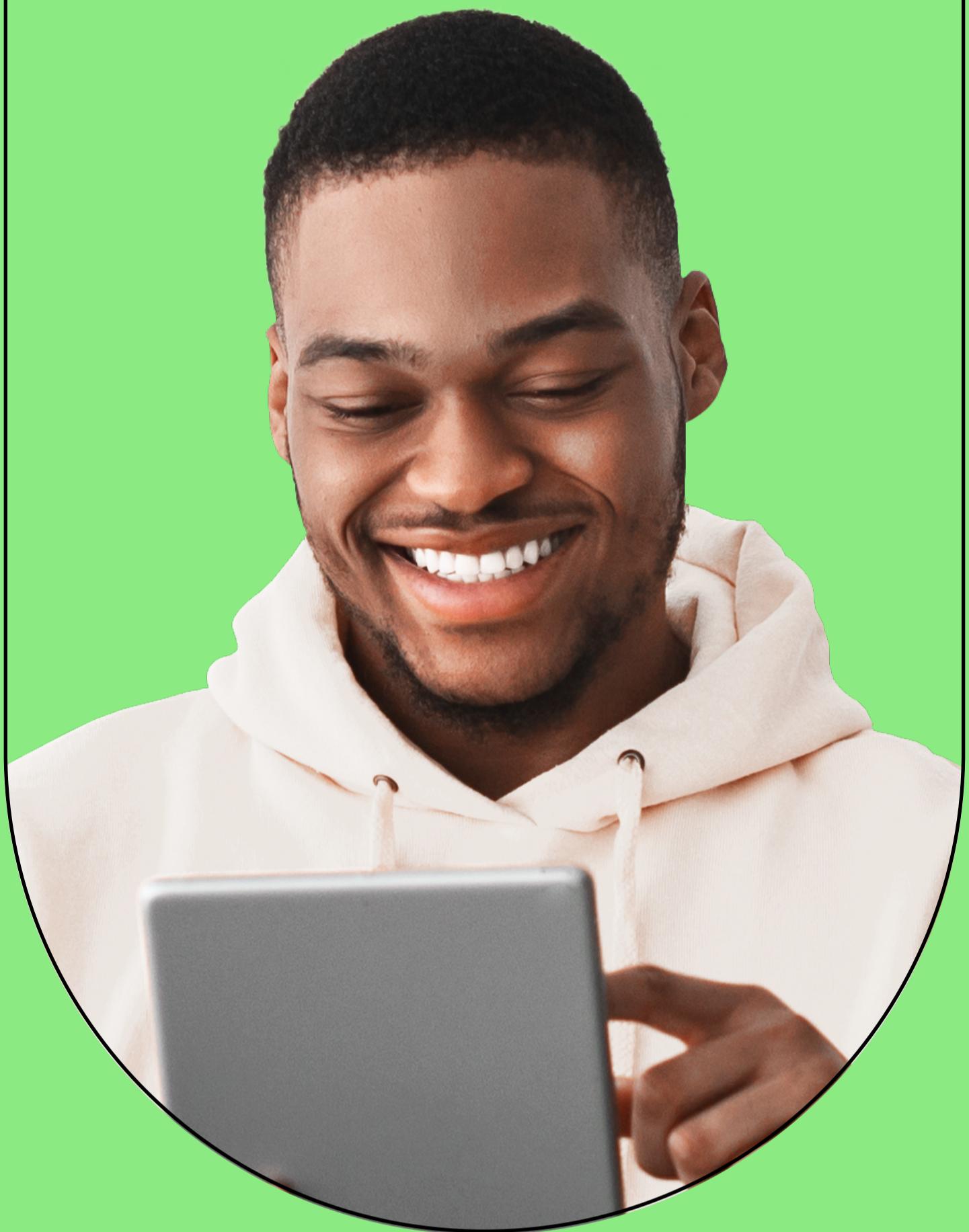
## Especificação do **back-end**

- > Em uma abordagem mais técnica, o back-end envolve servidor, banco de dados e aplicação.
- > Tudo isso é construído a partir de linguagens específicas de back-end, utilizadas para desenvolver a parte interna de um site. Algumas dessas linguagens são:
- > **PHP (Hypertext Preprocessor):** linguagem de script específica para o desenvolvimento de sites e aplicações web;
- > **Python:** lançada em 1991, é uma linguagem de alto nível utilizada para desktop, web, servidores e ciência de dados;
- > **Java:** uma das linguagens mais populares que engloba plataforma de software e linguagem de programação;
- > **JavaScript:** linguagem voltada para desenvolvimento web totalmente versátil.

# Especificações do front-end

- > Conhecido como “o lado do cliente”, o front-end é o responsável por toda a estrutura, design, conteúdo, comportamento, desempenho e capacidade de resposta de um site ou aplicação, ou seja, tudo o que é apresentado aos usuários para interação. Resumidamente, o front trabalha para criar a arquitetura que fornecerá uma boa experiência às pessoas.
- > É essa parte da programação que certifica se um site é responsivo e funciona perfeitamente em todas as telas de variados dispositivos.
- > O trabalho do desenvolvedor front-end também é baseado algumas linguagens principais. São elas:
- > **HTML (HyperText Markup Language):** utilizada para documentação e páginas web a partir de marcação de hipertexto;
- > **CSS (Cascading Style Sheets):** é uma linguagem de formatação de conteúdo, responsável pelo visual de um site; muito utilizada com HTML;
- > **JavaScript:** a linguagem também é utilizada para front-end, principalmente para criar dinamicidade nos sites.
- > É claro que o trabalho de um desenvolvedor back-end e front-end não se resume apenas às linguagens, existem ainda ferramentas de framework, bibliotecas, estruturas e softwares.





---

---

# Definição de **NODE**

# Especificações do front-end

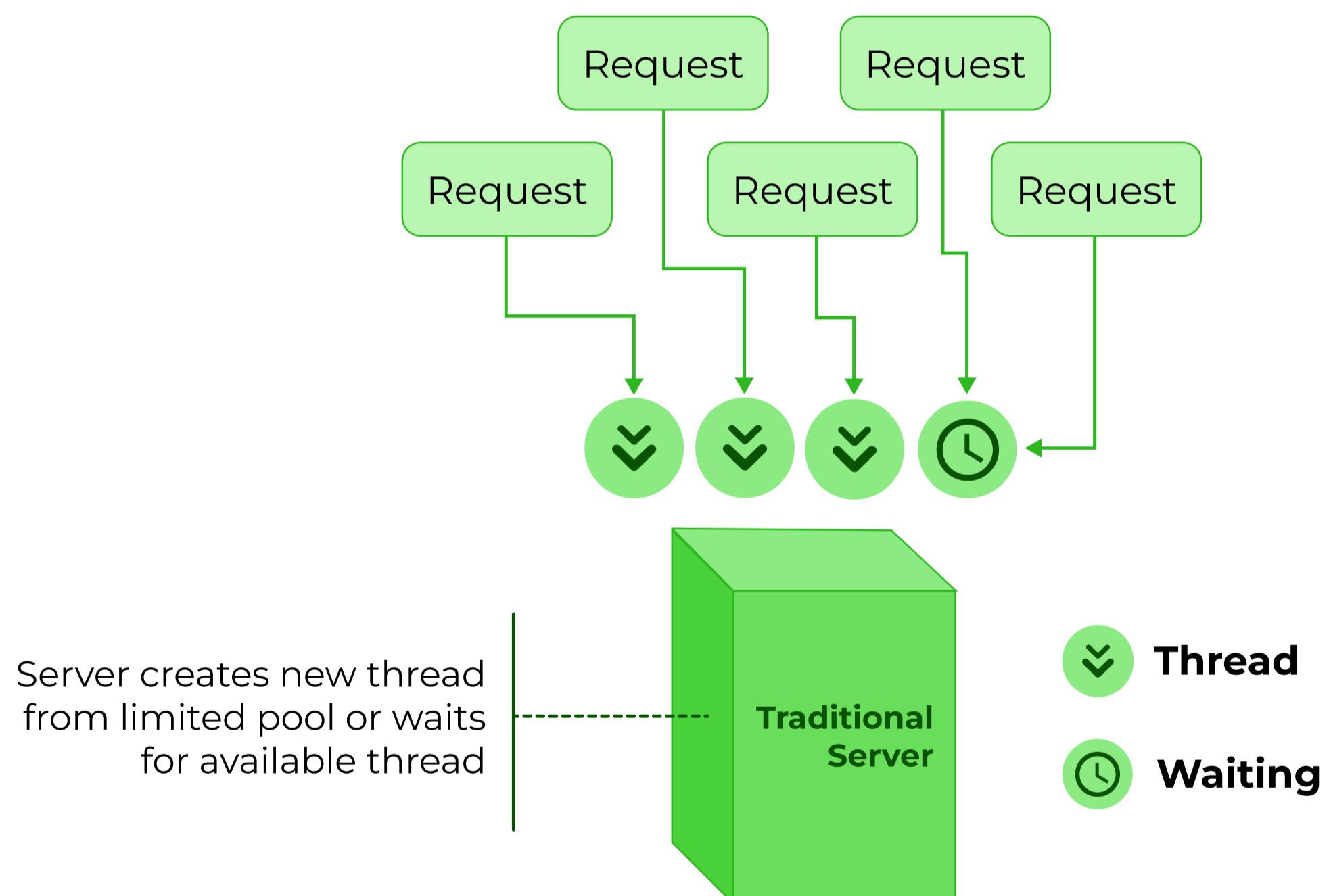
- > O Node.js pode ser definido como um ambiente de execução Javascript server-side.
- > Isso significa que com o Node.js é possível criar aplicações Javascript para rodar como uma aplicação standalone em uma máquina, não dependendo de um browser para a execução, como estamos acostumados.
- > Apesar de recente, o Node.js já é utilizado por grandes empresas no mercado de tecnologia, como Netflix, Uber e LinkedIn.
- > O principal motivo de sua adoção é a sua **alta capacidade de escala**. Além disso, sua arquitetura, flexibilidade e baixo custo, o tornam uma boa escolha para implementação de Microserviços e componentes da arquitetura Serverless. Inclusive, os principais fornecedores de produtos e serviços Cloud já têm suporte para desenvolvimento de soluções escaláveis utilizando o Node.js.

## Características

- > A principal característica que diferencia o Node.JS de outras tecnologias, como PHP, Java, C#, é o fato de sua execução ser **single-thread**.
- > Ou seja, apenas uma thread é responsável por executar o código Javascript da aplicação, enquanto que nas outras linguagens a execução é multi-thread.

# Mas o que isso significa na prática?

- › Em um servidor web utilizando linguagens tradicionais, para cada requisição recebida é criada uma nova thread para tratá-la. A cada requisição, serão demandados recursos computacionais (memória RAM, por exemplo) para a criação dessa nova thread. Uma vez que esses recursos são limitados, as threads não serão criadas infinitamente, e quando esse limite for atingido, as novas requisições terão que esperar a liberação desses recursos alocados para serem tratadas.
- › A figura abaixo mostra como funciona esse cenário em um servidor tradicional:



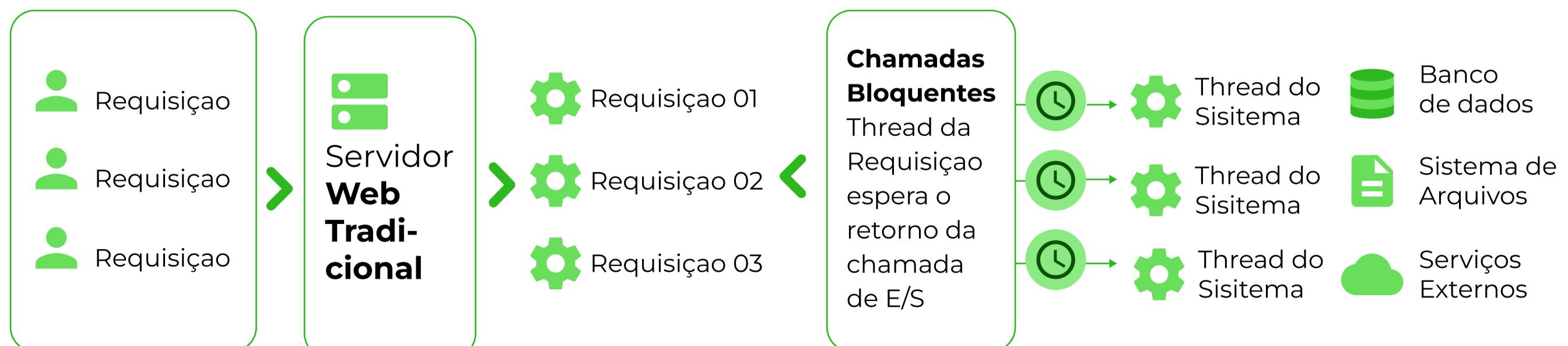
- › No modelo Node.js, apenas uma thread é responsável por tratar as requisições. Essa thread é chamada de **Event Loop**, e leva esse nome pois cada requisição é tratada como um evento.

- > O **Event Loop** fica em execução esperando novos eventos para tratar, e para cada requisição, um novo evento é criado.
- > Apesar de ser **single-threaded**, é possível tratar requisições concorrentes em um servidor Node.js. Enquanto o servidor tradicional utiliza o sistema multi-thread para tratar requisições concorrentes, o Node.js consegue o mesmo efeito através de chamadas de E/S (entrada e saída) não-bloqueantes.
- > Isso significa que as operações de entrada e saída (ex: acesso a banco de dados e leitura de arquivos do sistema) são assíncronas e não bloqueiam a thread. Diferentemente dos servidores tradicionais, a thread não fica esperando que essas operações sejam concluídas para continuar sua execução.
- > A figura abaixo representa a diferença de funcionamento de um servidor web tradicional e um Node.js:

## Modelo Node.js



## Modelo Tradicional



- No servidor Node.js, o Event Loop é a única thread que trata as requisições, enquanto que no modelo tradicional uma nova thread é criada para cada requisição.
- Enquanto o Event Loop delega uma operação de E/S para uma thread do sistema de forma assíncrona e continua tratando as outras requisições que aparecerem em sua pilha de eventos, as threads do modelo tradicional **esperam a conclusão** das operações de E/S, consumindo recursos computacionais durante todo esse período de espera.
- Apesar do Node.js ser single-threaded, sua arquitetura possibilita um número maior de requisições concorrentes sejam tratadas em comparação com o modelo tradicional, que é limitado devido ao alto consumo computacional pela criação e manutenção de threads a cada requisição.

## Vantagens de uso do Node.js

### Flexibilidade

- O **NPM** (Node Package Manager) é o gerenciador de pacotes do Node.js e também é o maior repositório de softwares do mundo. Isso faz do Node.js uma plataforma com potencial para ser utilizada em qualquer situação. O pacote mais conhecido se chama **Express.js** e é um framework completo para desenvolvimento de aplicações Web.

## Leveza

- Criar um ambiente Node.js e subir uma aplicação é uma tarefa que não exige muitos recursos computacionais em comparação com outras tecnologias mais tradicionais.
- Tanto sua leveza quanto flexibilidade fazem do Node.JS uma tecnologia indicada para a implementação de serviços e componentes de arquiteturas como a de microsserviços e serverless.

## Produtividade da equipe

- **Maior repositório do mundo:** O NPM fornece pacotes de código reusáveis e provavelmente aquela integração que você precisa fazer com outro sistema ou banco de dados já está implementado e disponível gratuitamente para instalar via NPM.
- **Mesma linguagem no frontend e backend:** Javascript é a linguagem padrão para desenvolvimento web client-side.



---

# Mas o que é **Node**?

- > **Node.js não é uma linguagem de programação.** Você programa utilizando a linguagem JavaScript, a mesma usada há décadas no client-side das aplicações web. Javascript é uma linguagem de scripting interpretada, embora seu uso com Node.js guarde semelhanças com linguagens compiladas, uma vez que máquina virtual V8 (veja mais adiante) faz etapas de pré-compilação e otimização antes do código entrar em operação.
- > **Node.js não é um framework Javascript.** Ele está mais para uma plataforma de aplicação, na qual você escreve seus programas com Javascript que serão compilados, otimizados e interpretados pela máquina virtual V8. Essa VM é a mesma que o Google utiliza para executar Javascript no browser Chrome, e foi a partir dela que o criador do Node.js, Ryan Dahl, criou o projeto. O resultado desse processo híbrido é entregue como código de máquina server-side, tornando o Node.js muito eficiente na sua execução e consumo de recursos.
- > **Node.js é uma tecnologia assíncrona que trabalha em uma única thread de execução.** Por assíncrona entenda que cada requisição ao Node.js não bloqueia o processo do mesmo, atendendo a um volume absurdamente grande de requisições ao mesmo tempo mesmo sendo single thread.



---

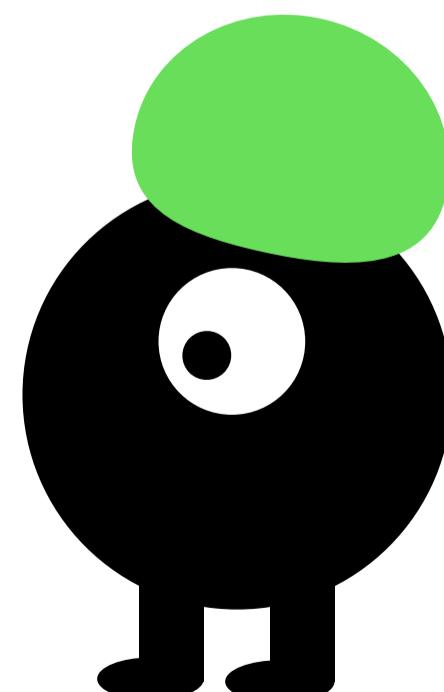
# Benchmark Node

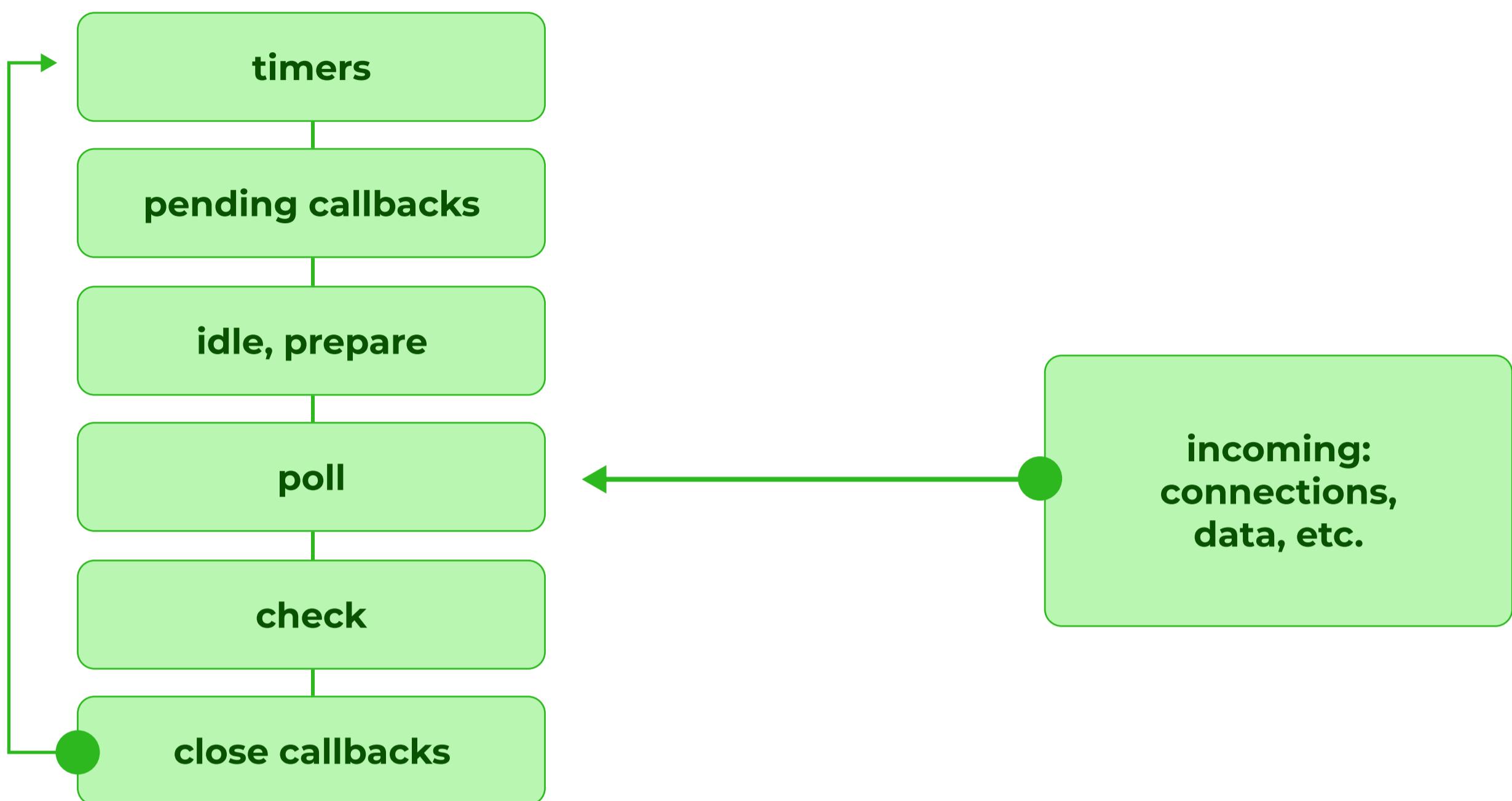
- Estamos vivendo em um admirável mundo novo. Um mundo cheio de JavaScript.
- Nos últimos anos, o JavaScript dominou a web, conquistando toda a indústria. Após a introdução do Node.js, a comunidade JavaScript foi capaz de utilizar a simplicidade e a dinamicidade da linguagem para ser a única linguagem para fazer tudo, lidando com o lado do servidor, o lado do cliente e até ousou e reivindicou uma posição para aprendizado de máquina.
- Novos conceitos foram introduzidos que nunca existiam antes, como **funções arrow e promessas**.
- Mas o JavaScript mudou drasticamente como linguagem nos últimos anos.

## Promises

Isso nos leva à pergunta: por que os retornos de chamada e as promessas foram introduzidos de qualquer maneira? Por que não podemos simplesmente escrever código executado sequencialmente em JavaScript?

Bem,  
teoricamente você pode.  
**Mas você deveria?**





- O problema de escrever JavaScript é que a linguagem em si é single threaded. Isso significa que você não pode executar mais de um único procedimento por vez, ao contrário de outras linguagens, como Go ou Ruby, que têm a capacidade de gerar threads e executar vários procedimentos ao mesmo tempo, seja em threads de kernel ou em threads de processo .
- Para executar o código, o JavaScript depende de um procedimento chamado loop de eventos, que é composto de vários estágios. O processo JavaScript passa por cada etapa e, no final, começa tudo de novo.
- Mas o JavaScript tem algo na manga para combater o problema de bloqueio. retornos de chamada de E/S (Entrada e saída).
- A maioria dos casos de uso da vida real que exigem que criemos um thread é o fato de que estamos solicitando alguma ação pela qual a linguagem não é responsável - por exemplo, solicitar uma busca de alguns dados do banco de dados.

- Em linguagens multithread, a thread que criou a solicitação simplesmente trava ou espera pela resposta do banco de dados. Isso é apenas um desperdício de recursos. Também sobrecarrega o desenvolvedor ao escolher o número correto de threads em um pool de threads. Isso é para evitar vazamentos de memória e a alocação de muitos recursos quando o aplicativo está em alta demanda.
- O JavaScript se destaca em uma coisa mais do que em qualquer outro fator, lidar com operações de E/S. JavaScript permite que você chame uma operação de E/S, como solicitar dados de um banco de dados, ler um arquivo na memória, gravar um arquivo em disco, executar um comando shell, etc. Quando a operação for concluída, você executa uma **chamada de retorno**. Ou no caso de promessas, você **resolve** a promessa com o resultado ou a **rejeita** com um erro.
- A comunidade de JavaScript sempre nos aconselha a nunca usar código síncrono ao fazer operações de E/S. A razão bem conhecida para isso é que NÃO queremos impedir que nosso código execute outras tarefas. Como é single-thread, se tivermos um pedaço de código que lê um arquivo de forma síncrona, o código bloqueará todo o processo até que a leitura seja concluída. Em vez disso, se dependermos de código assíncrono, podemos fazer várias operações de E/S e lidar com a resposta de cada operação individualmente quando ela for concluída. Nada de bloqueio.
- Mas certamente em um ambiente onde **não nos importamos em lidar com muitos processos**, usar código **síncrono e assíncrono** não faz diferença alguma, certo?

- O teste que vamos executar terá como objetivo nos fornecer benchmarks sobre a rapidez com que o código sincronizado e assíncrono é executado e se há uma diferença no desempenho.
- Decidi escolher a leitura de um arquivo como a operação de E/S a ser testada.

## Exemplo

- Veja essa função que escreverá um arquivo aleatório preenchido com bytes aleatórios gerados com o módulo **Node.js Crypto**:

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

fs.writeFileSync( "./test.txt", crypto.randomBytes(2048)
  .toString('base64') )

const fs = require('fs');

process.on( 'unhandledRejection', (err) =>{
  console.error(err);
})

function synchronous() {
  console.time("sync");
  fs.readFileSync("./test.txt")
  console.timeEnd("sync")
}

async function asynchronous() {
  console.time("async");
  let p0 = fs.promises.readFile("./test.txt");
  await Promise.all([p0])
  console.timeEnd("async")
}

synchronous()
asynchronous()
```

- A execução do código anterior resultou nos seguintes resultados:

Run#	Sync	Async	Async/Sync	Razão
1	0.278ms	3.829ms	13.773	
2	0.335ms	3.801ms	11.346	
3	0.403ms	4.498ms	11.161	

- Bom, isso foi inesperado.
- As expectativas iniciais eram de que eles deveriam levar o mesmo tempo.
- Bem, que tal adicionarmos outro arquivo e lermos 2 arquivos em vez de 1?

## Segue código atualizado:

```
function synchronous(){
  console.time("sync");
  fs.readFileSync("./ test.txt")
  fs.readFileSync("./ test2.txt")
  console.time End("sync")
}

async function asynchronous(){
  console.time("async");
  let p0=fs.promises.readFile("./ test.txt");
  let p1=fs.promises.readFile("./ test2.txt");
  await Promise.all([p0,p1])
  console.time End("async")
}
```

- Simplesmente adicionei outra leitura para cada uma delas, e nas **promises**, fiquei aguardando as promessas de leitura que deveriam estar rodando em paralelo. Estes foram os resultados:

<b>Run#</b>	<b>Sync</b>	<b>Async</b>	<b>Async/Sync</b>	<b>Razão</b>
<b>1</b>	1.659ms	6.895ms	4.156	
<b>2</b>	0.323ms	4.048ms	12.533	
<b>3</b>	0.324ms	4.017ms	12.398	
<b>4</b>	0.333ms	4.271ms	12.826	

- A primeira tem valores completamente diferentes das 3 execuções que se seguem. Meu palpite é que está relacionado ao compilador JavaScript JIT que otimiza o código em cada execução.
- Então, as coisas não parecem tão boas para funções assíncronas. Talvez se tornarmos as coisas mais dinâmicas e talvez estressarmos um pouco mais o aplicativo, possamos obter um resultado diferente.
- Então, o próximo teste envolve escrever 100 arquivos diferentes e depois ler todos eles.
- Primeiro, modificar o código para escrever 100 arquivos antes da execução do teste. Os arquivos são diferentes em cada execução, embora mantendo quase o mesmo tamanho.

```

let filePaths = [];

function writeFile() {
  let filePath = `./files/${crypto.randomBytes(6)
    .toString('hex')}.txt`
  fs.writeFileSync( filePath, crypto.randomBytes(2048)
    .toString('base64'))
  filePaths.push(filePath)
}

function synchronous() {
  console.time("sync");
  /*fs.readFileSync("./test.txt")
  fs.readFileSync("./test2.txt") */
  filePaths.forEach((filePath) =>{
    fs.readFileSync(filePath)
  })
  console.timeEnd("sync")
}

async function asynchronous() {
  console.time("async");
  /* let p0 = fs.promises.readFile("./test.txt");
  let p1 = fs.promises.readFile("./test2.txt"); */
  // await Promise.all([p0,p1])
  let promiseArray = [];
  filePaths.forEach((filePath) =>{
    promiseArray.push(fs.promises.readFile(filePath))
  })
  await Promise.all(promiseArray)
  console.timeEnd("async")
}

```

## E para limpeza e execução:

```

let oldFiles = fs.readdirSync("./files")
oldFiles.forEach((file) =>{
  fs.unlinkSync("./files/" + file)
})
if (!fs.existsSync("./files")){
  fs.mkdirSync("./files")
}

for (let index = 0; index < 100; index++) {
  writeFile()
}

synchronous()
asynchronous()

```



- E rodando o código, segue a tabela de resultados:

<b>Run#</b>	<b>Sync</b>	<b>Async</b>	<b>Async/Sync</b>	<b>Razão</b>
<b>1</b>	4.999ms	12.890ms	2.579	
<b>2</b>	5.077ms	16.267ms	3.204	
<b>3</b>	5.241ms	14.571ms	2.780	
<b>4</b>	5.086ms	16.334ms	3.213	

➤ Isso indica que com o aumento da demanda ou da simultaneidade, as promessas de overhead começam a fazer sentido. Para fins de elaboração, se estivermos executando um servidor da Web que deve executar centenas ou talvez milhares de solicitações por segundo por servidor, a execução de operações de E/S usando sincronização começará a perder seu benefício rapidamente.

➤ Apenas por uma questão de experimentação, vamos ver se é realmente um problema com as próprias promessas ou é outra coisa.

➤ Para isso, uma função que irá calcular o tempo para resolver uma promessa que não faz absolutamente nada e outra que resolve 100 promessas vazias.

➤ Segue o código abaixo:

```
function promiseRun() {  
    console.time("promise run");  
    return new Promise((resolve) => resolve())  
        .then(() => console.timeEnd("promise run"))  
}  
  
function hundrededPromiseRuns(){  
    let promiseArray = [];  
    console.time("100 promises")  
    for(let i = 0; i < 100; i++) {  
        promiseArray.push(new Promise((resolve) => resolve()))  
    }  
    return Promise.all(promiseArray)  
        .then(() => console.timeEnd("100 promises"))  
}  
  
promiseRun()  
hundered PromiseRuns()
```

Run#	Single Promises	100 Promises
1	1.651ms	3.293ms
2	0.758ms	2.575ms
3	0.814ms	3.127ms
4	0.788ms	2.623ms

- Interessante.
- Parece que as promises não são a principal causa do atraso, o que faz supor que a fonte do atraso são os threads do kernel fazendo a leitura real. Isso pode levar um pouco mais de experimentação para chegar a uma conclusão decisiva sobre a principal razão por trás do atraso.

# Uma palavra final

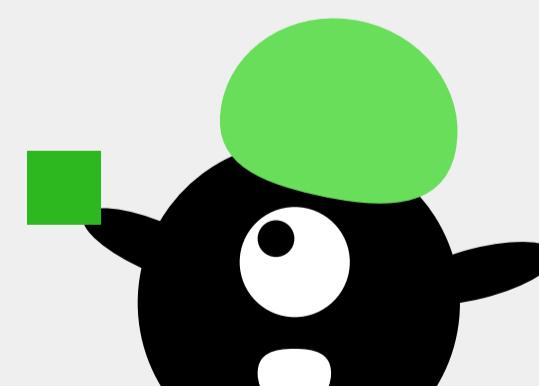
- › **Então você deve usar promises ou não?**
- › Uma opinião seria a seguinte:

Se você estiver escrevendo um script que será executado em uma única máquina com um fluxo específico acionado por um pipeline ou um único usuário, use o código de **sincronização**.

Se você estiver escrevendo um servidor da Web que será responsável por lidar com muito tráfego e solicitações, a sobrecarga que vem da execução **assíncrona** superará o desempenho do código de sincronização.

**Segue o link** para documentação oficial do NODEJS:

Node 





---

# NPM

# O que é e como funciona o NPM?

- O npm é o Gerenciador de Pacotes do Node (Node Package Manager) que vem junto com ele e que é muito útil no desenvolvimento Node. Por anos, o Node tem sido amplamente usado por desenvolvedores JavaScript para compartilhar ferramentas, instalar vários módulos e gerenciar suas dependências.

## O npm funciona baseado nesses dois ofícios:

- Ele é um repositório amplamente usado para a publicação de projetos Node.js de código aberto (open-source). Isso significa que ele é uma plataforma online onde qualquer pessoa pode publicar e compartilhar ferramentas escritas em JavaScript.
- O npm é uma ferramenta de linha de comando que ajuda a interagir com plataformas online, como navegadores e servidores. Essa utilidade auxilia na instalação e desinstalação de pacotes, gerenciamento da versões e gerenciamento de dependências necessárias para executar um projeto.
- Para usá-lo, você precisa instalar o node.js – visto que, eles são empacotados juntos.
- Para usar os pacotes, seu projeto deve conter um arquivo chamado de package.json . Dentro do pacote, você encontrará metadados específicos para os projetos.

- Os metadados mostram alguns aspectos do projeto na seguinte ordem:
  - O nome do projeto;
  - A versão inicial;
  - A descrição;
  - O ponto de entrada;
- Continuando: os metadados mostram alguns aspectos do projeto na seguinte ordem:
  - Os comandos de teste;
  - O repositório git;
  - As palavras-chave;
  - A licença;
  - As dependências;
  - As dependências do desenvolvedor (devDependencies).
- Os metadados ajudam a identificar o projeto e agem como uma base para que os usuários obtenham as informações sobre ele.

## Exemplo:

```
{
  "name": "gama-academy-npm",
  "version": "1.0.0",
  "description": "npm guide for beginner",
  "main": "beginner-npm.js",
  "scripts": {
    "test": "echo\\\"Error: no test specified\\\"&& exit 1"
  },
  "keywords": [
    "npm",
    "example",
    "basic"
  ],
  "author": "Gama Academy",
  "license": "MIT",
  "dependencies": {
    "express": "^4.16.4"
  }
}
```





# Explicação:

- › O nome é gama-academy-npm;
  - › A versão é 1.0.0;
  - › A descrição informa npm guide for beginner;
  - › O ponto de entrada do projeto ou o arquivo principal é beginner-npm.js;
  - › As palavras chave ou tags para encontrar o projeto no repositório são npm, example e basic;
  - › O autor do projeto é Gama Academy;
  - › Este projeto está licenciado sob o MIT;
  - › As dependências ou outros módulos que esse módulo usa são express 4.16.4.
- 



---

# Começar um Projeto com **NPM**

- Se você já tiver o Node e o npm e deseja começar a construir seu projeto, execute o comando **npm init**. Com isso, você dará procedimento à inicialização do seu projeto.

```
npm init
```

- Digite **yes** e pressione Enter para confirmar, salvando, assim, o package.json. Você sempre pode modificá-lo depois, editar o arquivo diretamente ou executar **npm init** novamente.
- Após esse comando, entre na pasta do projeto que você acabou de criar escrevendo esse comando no terminal:

```
cd nome_da_pasta_do_seu_projeto
```

- Esse comando funciona como uma ferramenta para criar o arquivo **package.json** de um projeto.
- Depois de executar as etapas do **npm init**, um arquivo **package.json** será gerado e colocado no diretório atual.

# Instalar Módulos npm

- Um pacote em node.js contém todos os arquivos que você precisa para um módulo. Os módulos são bibliotecas JavaScript que você pode incluir no seu projeto.
- Instalar módulos é uma das coisas mais básicas que você deve aprender a fazer quando começar a usar o gerenciador de pacote Node. Segue um comando para instalar um módulo no diretório atual:

```
npm install <module>
```

```
npm i <module>
```

- No comando acima, substitua o <module> pelo nome do módulo que você quer instalar.
- Por exemplo, se você quer instalar o módulo Express – o mais usado e mais bem conhecido framework web node.js – você pode executar o seguinte comando:

```
npm install express
```

- O comando acima irá instalar o módulo express em /node\_modules no diretório atual.
- Sempre que você instalar um módulo do npm, ele será instalado na pasta node\_modules .



---

# O que é JavaScript?

# Definição de alto nível

- JavaScript é uma linguagem de programação que permite a você implementar itens complexos em páginas web — toda vez que uma página da web faz mais do que simplesmente mostrar a você informação estática — mostrando conteúdo que se atualiza em um intervalo de tempo, mapas interativos ou gráficos 2D/3D animados, etc. — você pode apostar que o JavaScript provavelmente está envolvido.
- É a terceira camada do bolo das tecnologias padrões da web, duas das quais são o HTML e CSS.



- **HTML** é a linguagem de marcação que nós usamos para estruturar e dar significado para o nosso conteúdo web. Por exemplo, definindo parágrafos, cabeçalhos, tabelas de conteúdo, ou inserindo imagens e vídeos na página.
- **CSS** é uma linguagem de regras de estilo que nós usamos para aplicar estilo ao nosso conteúdo HTML. Por exemplo, definindo cores de fundo e fontes, e posicionando nosso conteúdo em múltiplas colunas.
- **JavaScript** é uma linguagem de programação que permite a você criar conteúdo que se atualiza dinamicamente, controlar mídias, imagens animadas, e tudo o mais que há de interessante. Ok, não tudo, mas é maravilhoso o que você pode efetuar com algumas linhas de código JavaScript.

- As três camadas ficam muito bem uma em cima da outra. Vamos exemplificar com um simples bloco de texto. Nós podemos marcá-lo usando HTML para dar estrutura e propósito:

```
<p>Jogador 1: Chris /p>
```

- Nós podemos adicionar um pouco de CSS na mistura, para deixar nosso parágrafo um pouco mais atraente:

```
p{
  font-family: 'helvetica neue', helvetica, sans-serif;
  letter-spacing: 1px;
  text-transform: uppercase;
  text-align: center;
  border: 2px solid rgba(0,0,200,0.6);
  background: rgba(0,0,200,0.3);
  color: rgba(0,0,200,0.6);
  box-shadow: 1px 1px 2px rgba(0,0,200,0.4);
  border-radius: 10px;
  padding: 3px 10px;
  display: inline-block;
  cursor: pointer;
}
```

PLAYER: CHRIS

- E finalmente, nós podemos adicionar JavaScript para implementar um comportamento dinâmico:

```
const para = document.querySelector('p');

para.addEventListener('click', atualizarNome);

function atualizarNome(){
  var nome = prompt('Insira um novo nome');
  para.textContent='Jogador1: ' + nome;
}
```

JOGADOR 1: MEU NOME



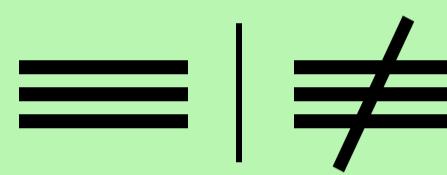
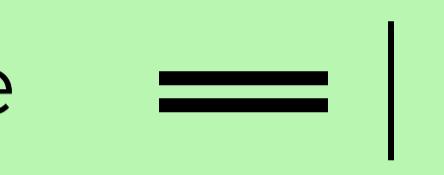
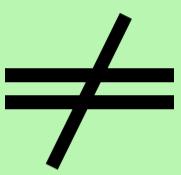


---

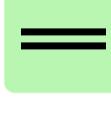
Algumas boas  
práticas de  
**desenvolvimento**  
**em JavaScript**

# Use ao invés de

-  O JavaScript usa dois tipos diferentes de operadores de igualdade:

 |  e  | 

Considera-se boa prática, sempre usar a primeira dupla, ao fazer comparações.

-  Se dois operandos tem o mesmo tipo e valor, o  retornar verdadeiro e  retornar falso.
-  Entretanto, ao usarmos  e , você acabará se deparando com problemas, ao trabalhar com tipos diferentes. Nesses casos, eles farão uma conversão dos valores, sem sucesso.





# Eval - Ruim

- Para aqueles não familiarizados, a função eval dá-nos acesso ao compilador do JavaScript. Essencialmente, podemos executar o resultado de uma cadeia de caracteres passando-a como parâmetro para a função eval .
- Além de diminuir substancialmente a performance do seu script, isso também gera um grande risco à segurança da aplicação, uma vez que dá muito poder ao que foi passado como texto. **Evite!**



# Não Use Abreviações

- › Tecnicamente, você não precisa usar chaves ou ponto-e-vírgulas. A maioria dos navegadores interpretará corretamente o trecho abaixo:

```
if (someVariableExists)
    x = false
```

Entretanto, considere o seguinte:

```
if (someVariableExists)
    x = false
anotherFunctionCall()
```

- › Alguém pode achar que o código acima equivale ao do trecho abaixo:

```
if (someVariableExists) {
    x = false;
    anotherFunctionCall()
}
```

- › Infelizmente, ele estaria errado. Na verdade, ele significaria:

```
if (someVariableExists) {
    x = false
}
anotherFunctionCall()
```

- › Como você perceberá, a indentação imita o funcionamento das chaves. Não é preciso dizer que, essa prática, é terrível, e deve ser evitada a todo custo. A única vez que as chaves podem ser omitidos é nas instruções de linha única e, ainda assim, isso também é bastante debatido.

```
if (2 + 2 === 4) return 'nicely done'
```



# Declare Variáveis, Fora da Instrução For

- Ao executar instruções for muito grandes, não faça com que o motor JavaScript tenha de trabalhar mais que ele realmente precisa. **Por exemplo:**

## RUIM

```
for (let i = 0; i < someArray.length; i++) {  
  let container = document.getElementById('container')  
  container.innerHTML += 'my number: ' + i  
  console.log(i)  
}
```

- Perceba como precisamos determinar o tamanho do vetor a cada iteração e como temos de atravessar a DOM para encontrar o elemento container todas as vezes – altamente ineficiente!

## BOM

```
let container = document.getElementById('container')  
for (let i = 0, len = someArray.length; i < len; i++) {  
  container.innerHTML += 'my number: ' + i  
  console.log(i)  
}
```



# Reduza as Variáveis Globais



Ao reduzir a quantidade de variáveis globais a um único nome, você reduz, significantemente, a chance de más interações com outras aplicações, widgets ou bibliotecas.

## RUIM

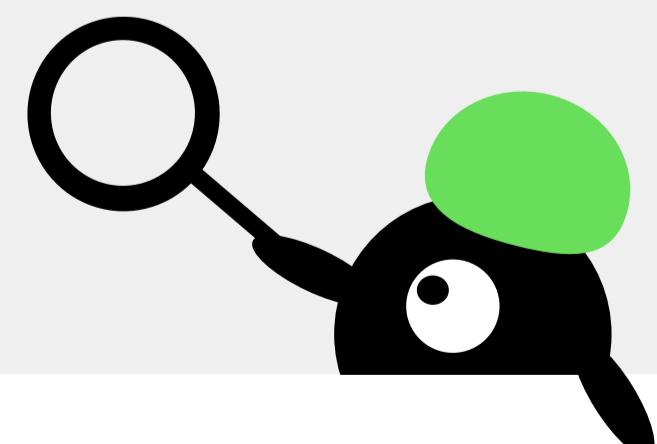
```
let name = 'Jeffrey'  
let lastName = 'Way'  
  
function doSomething() { ... }  
  
console.log(name) // Jeffrey -- or window.name
```

## BOM

```
let DudeNameSpace = {  
  name: 'Jeffrey',  
  lastName: 'Way',  
  doSomething: function() { ... }  
}  
console.log(DudeNameSpace.name) // Jeffrey
```

Link para documentação oficial  
sobre a linguagem **JavaScript**:

**JavaScript**





---

# TypeScript

# O que é TypeScript?

- TypeScript é um superset para JavaScript, ou um conjunto adicional de instruções, keywords e estruturas, criado pela Microsoft.
- Ou seja, você ainda estará programando JavaScript, mas com “super poderes”.
- E que super poderes são essas?
- O primeiro, mais direto e mais simples é a tipagem estática, muito comum em linguagens como Java, C e derivados, mas algo que passa longe do JavaScript geralmente.
- Mas TypeScript vai muito além disso, adicionando várias outras validações, oferecendo suporte a Generics, interfaces e muito mais, além de estender enormemente as possibilidades do uso de classes em JS.
- Note que todos estes benefícios são em tempo de codificação, pois em produção, você terá o bom e velho JS sendo transpilado a partir do TypeScript novamente.



# Instalando e testando TypeScript

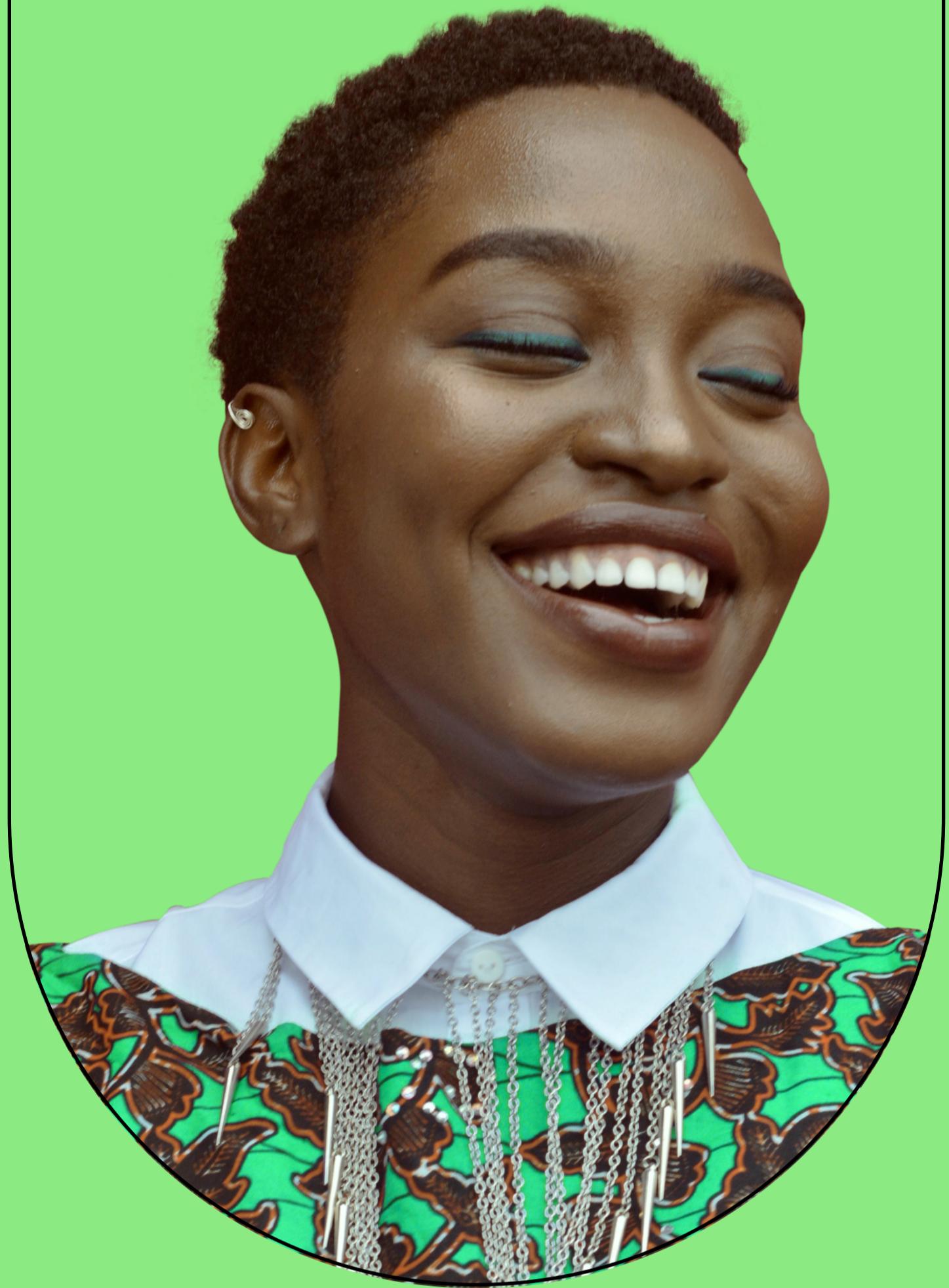
- O primeiro passo é você instalar o compilador do TypeScript na sua máquina. Claro, aqui considerando que você já tem o Node.js instalado.
- Você pode fazer isso facilmente instalando a dependência do mesmo de maneira global via NPM (não esqueça que você precisa ter permissões de administrador).

```
npm install -g typescript
```

- Para verificar se foi instalado corretamente, você pode criar um arquivo qualquer com a extensão .ts (de TypeScript) e tentar compilá-lo usando o utilitário tsc , como abaixo:

```
tsc app.ts
```





---

# TypeScript com exemplos práticos

# Tipagem

- Para começarmos a usar TypeScript, vamos fazer alguns exemplos práticos bem simples.
- Primeiro, crie um arquivo app.ts e insira um código JS como abaixo:

```
//app.ts
function somar(num1, num2){
    return num1 + num2
}

console.log(somar(1,2))
console.log(somar('1', '2'))
```

- Agora, vamos colocar a tipagem nos parâmetros desta função somar. Para adicionar tipagem a um parâmetro é bem simples, basta usar **:tipo** depois do nome da variável, como abaixo.

```
//app.ts
function somar(num1: number, num2: number){
    return num1 + num2
}

console.log(somar(1, 2))
console.log(somar('1', '2'))
```

- No momento que você fizer isso, a sua ferramenta de editor de código já vai reclamar do uso errado na segunda chamada do somar. E, se mesmo assim, você quiser tentar compilar, vai dar erro no console.



# Retornos e inferência de tipos

- Além dos parâmetros, você pode usar tipagem no retorno de funções, sendo tão simples quanto no primeiro exemplo, bastando adicionar :tipo ao final da assinatura da função e antes de abrir chaves.

```
function dividir(num1: number, num2: number): number{  
    return num1 / num2  
}
```

- No exemplo acima, além dos parâmetros terem de ser numéricos, o retorno também o será.

# Typecasting e configuração

- Em alguns casos a inferência de tipos não vai lhe ajudar.
- Seja porque o tipo retornado por uma função é muito genérico, seja porque a função não tem tipo de retorno definido.
- Em ambos os casos, se você sabe o que será retornado, você pode usar typecasting ou conversão de tipo.

- Para fazer isso, você usa o operador 'as' logo depois da chamada da function, citando o tipo a ser convertido logo a sequência.

```
function multiplicar(num1, num2) {  
    return num1 * num2  
}  
  
const resultado2 = multiplicar(2, 3) as number
```

- Isso garantirá que **resultado2**, será um number . Claro, se a conversão for possível, caso contrário dará erro, então tome cuidado com esse recurso.
- Para acionarmos mais alguns poderes do TS temos de adicionar um arquivo de configuração em nosso projeto. Você faz isso usando o comando abaixo na pasta do projeto:

```
tsc -init
```

- Com isso, será criado um arquivo tsconfig.json , como abaixo.

```
}  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "strict": true,  
    "esModuleInterop": true,  
    "skipLibCheck": true,  
    "forceConsistentCasingInFileNames": true  
}
```

# NULL

- “**strict**”: **true** , que é altamente recomendada que seja deixada assim, mas que ao mesmo tempo é a mais xarope de todas, isso porque ela reforça regras de segurança no código, como o cuidado a variáveis nulas e a tipos não especificados.
- Se você tiver algum trecho de código que possa retornar null e com isso estourar uma exceção em produção, o compilador do TS não vai deixar você avançar antes de implementar um teste para ver se o valor não é null primeiro.
- Outra situação em que o **strict** mode vai te incomodar é com a declaração de tipos que deixará de ser opcional e passará a ser obrigatória. Caso você, por algum motivo, ainda queira que uma variável, parâmetro ou retorno de função tenha tipagem dinâmica, você pode usar o tipo ‘**any**’ para ele.

Link para documentação oficial  
do **TypeScript**:

[TypeScript](#)



