



# Node JS

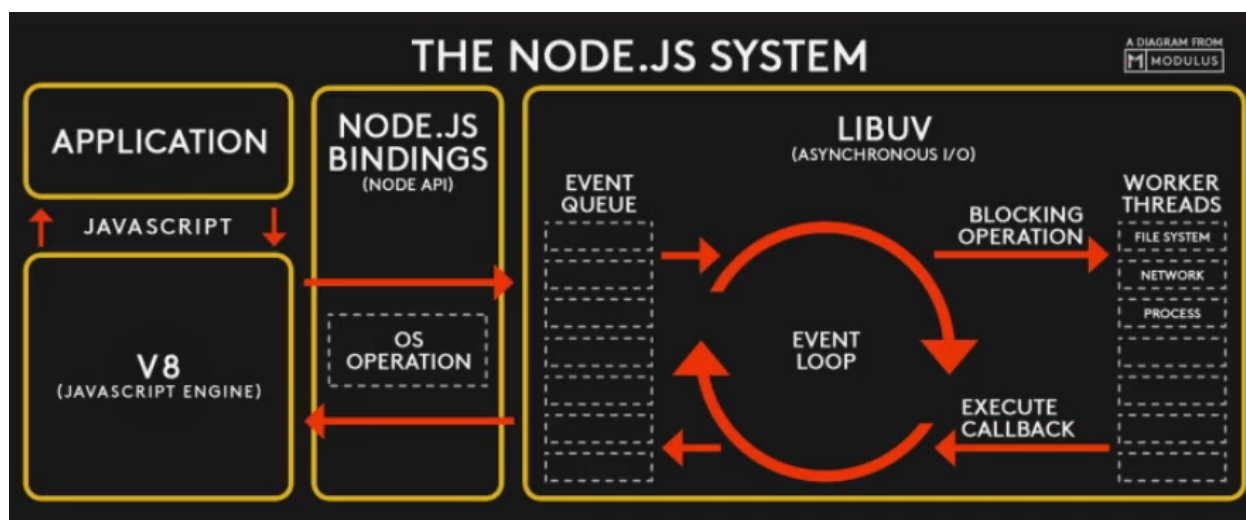
Runtime Environment de Javascript no computador.

É útil para a criação de backend e front-end.

Permite que requests sejam feitas de forma assíncrona, não bloqueia o “garçom” e permite várias requests simultâneas.

O **Event Loop do JS** permite que a sua única thread “garçom” faça requests mas não seja bloqueado.

Assincronia e funcionamento do Node.



Operações assíncronas esperam a event queue terminar pra serem executadas.

## Módulos

São objetos/arrays/variáveis criadas em outros arquivos .js e exportadas para uso.

Já existem alguns objetos dentro do node que podemos utilizá-lo. Como o objeto `module`

Para importar um modulo usamos a função `require("modulo")` e guardamos em uma variável.

Se quisermos exportar um módulo usamos `module.export = O` que eu quero exportar.

## Argv e flags

## Criar módulo.

`npm init` → Cria o pacote do projeto.

ou

`npm init -y` Para dar sim em tudo

Esse pacote é necessário para todos os projetos.

## Package.json:

É um arquivo JSON (Js Object Notation) que guarda informações sobre o seu pacote.

O seu módulo pode ser um amontoado de módulos, então cada aplicação tem o seu próprio módulo que puxa outros módulos.

## Baixar módulos:

`npm install nome-do-modulo` ou `npm i nome-do-modulo`

Isso cria uma pasta chamada **node-modules** e um arquivo **package-lock.json**.

## Node modules

é uma pasta com as dependencias dos módulos instalados, não é preciso compartilhá-lo, **uma vez que o ao instalar um módulo, ele é acrescentado ao seu proprio package.json**, e um simples `npm install` traz todo o **node\_modules** de volta.

## Package-lock.json

Não se mexe, ele aponta os diretorios de cada modulo pra que eles funcionem.

Pacotes utilizados somente na fase de desenvolvimento do programa podem ser instalados usando `npm install nome-do-modulo -D`

Instalar módulos globalmente: Instalar modulos no pc, e não no projeto.

`npm install nome do modulo -g`

## Gerenciar versão de pacotes.

`npm outdated` Lista os pacotes fora de versão

`npm install nomedopacote@versão` Instala uma versão específica do pacote

## Rodar scripts dos módulos

Cada package no seu package.json tem scripts de linux dentro, eles podem ser executados com `npm run nome do script`

## Desinstalar módulos

`npm uninstall nome-do-pacote`

## Módulo Process: Stdin e Stdout

No NodeJS existe um módulo chamado process, que referencia o processo que está rodando no terminal.

Esse `process` é um objeto, e possui os atributos stdin e stdout como objetos.

**Stdin:** objeto que armazena o input de dados. `process.stdin`

**Stdout:** objeto que gerencia o output de dados. `process.stdout`

Stdin e stdout possuem métodos.

`process.stdout.write ( "String" )` Escreve a string na tela.

É isso que o console.log faz debaixo dos panos

`process.stdin.on ( "data", funcao() )` Funcao assincrona que sempre que forem escritos dados, a função de callback rodará, como se fosse em um while.

Na verdade o método `.on` é faz parte de **todo objeto dentro de process**, incluindo ele mesmo.

O método `.on` é traduzido como: **Execute tal função enquanto receber tal evento.**

`.on ( "evento" , função callback )`

Existem vários tipos de evento, como "data".

Outros métodos de process podem disparar eventos, como `process.exit()`

Pode existir um `process.on ( "exit", funcao() )` que escutará esse exit e fará alguma coisa se ele for disparado.

## Timers: Trabalhar com o tempo e assincronia no NodeJS.

```
setTimeout( funcao callback, milissegundos )
```

A função setTimeout executa uma função depois de um tempo determinado.

O detalhe é que essa função não trava a execução do programa, ela é executada de forma assíncrona. A execução da função é jogada pra fila.

```
clearTimeout( objeto {Timeout} )
```

**Cancela** a execução de um **setTimeout**.

Um `setTimeout` retorna um **objeto Timeout**. Que registra a entrada da execução do **callback na fila**.

Se passarmos esse timeout em uma variavel e depois na função

```
clearTimeout( timeout )
```

, ele será deletado da fila.

```
setInterval( funcao callback, milissegundos )
```

Executa uma função callback a cada x milissegundos

```
clearInterval( obeto {SetInterval} )
```

Cancela a execução de um setInterval

Pode ser jogada como um callback de setTimeout

```
setTimeout( clearInterval ( objeto {SetInterval} ))
```

## Módulo Events.

Permite que códigos sejam rodados quando certo evento for emitido.

Vários módulos herdam as funções do módulo events, como process, https, stream etc.

**Importando:**

```
const { EventEmitter } = require ( "events" )
```

 Retorna uma classe EventEmitter pois o módulo exporta essa classe.

```
let eventos = new EventEmitter
```

 Cria uma instância desse objeto.

**Emissão de evento:**

```
eventos .emit ( "Identificador do evento" , argumentos )
```

Dado alguma condição, vamos precisar emitir um evento que será ouvido posteriormente.

OBS: `process.exit()` emite um evento com o identificador "exit"

### Escuta de evento:

```
eventos .on ( "identificador", funcao callback() )
```

Executa uma escuta assíncrona da emissão dos eventos e executa o callback.

```
eventos .once ( "identificador", funcao callback() )
```

Executa uma escuta assíncrona da emissão dos eventos UMA ÚNICA VEZ e executa o callback

## Express

Express é um módulo Web Framework para criar um servidor na web usando uma porta local.

### Importando

```
const express = require( "express" )
```

```
const app = express()
```

**App.** Essa é uma nomeação muito utilizada por várias pessoas.

App agora é o nosso objeto para manipular o servidor.

### Ligando o servidor

```
app. listen ( "porta desejada" )
```

Utiliza-se a 3000.

## Rotas

O nosso site possui rotas, ou seja, as slugs.

Ao entrar em cada página, naquela página, o client faz um request **GET** pro servidor.

O servidor retorna algum response ou body definido naquela rota.

### Definindo a resposta de uma rota.

```
app .route ( "slug da rota" )
```

### Setando o que receber no GET

```
app .route ( "slug da rota" ) .get ( ( req, res ) => res .send ( "conteudo para enviar ao get" ) )
```

**req e res** são, o request e o response respectivamente.

O req é enviado pelo client/user-agent, e o res é o que o servidor envia ao cliente. Dessa forma, ao acessar a slug, o get retornará esse conteúdo setado.

## Setando o que fazer com um request POST

Antes é necessário fazer com que o express js faça a leitura de jsons.

O chamado *middleware* que é a o middlegroud entre a requisição e a resposta.

```
app.use ( express.json() )
```

```
app.route("posts").post((req, res) => {  
  //Printar os dados enviados pelo request  
  console.log(req.body)  
  //Enviar um conteúdo de volta.  
  res.send("conteudo")  
})
```

Obs, enviamos um request post usando o CURL do git bash.

```
curl -X POST -H "content-type: application/json" -d '{"nome": "arara", "idade": 15}'  
localhost:3000/posts
```

## PUT PATCH E DELETE

Essas três requests geralmente utilizam um indice do array pra apagar ou alterar o seu valor. A diferença é que nós passamos um parâmetro (identificador) na rota que vai ser enviado ao backend. Depois disso, o back end faz a deleção.

Codigo para enviar variáveis na URL e utilizá-las na deleção.

```
let posts = [{"id": 1}, {"id": 2}]  
  
app.route("posts:VARIABLE").delete((req, res) => {  
  posts.splice(req.params.VARIABLE)  
})
```

## Passando parâmetros nas requisições.

É possível passar parâmetro de três formas.

### Body do Request

curl -d e -H

### Route Params :

São parâmetros enviados na URL, rota.

São identificados pelo :variavel

exemplo: `localhost:3000/posts/:variavel`

Elas vão ficar localizadas em `req.params`

### Query params ?


São parâmetros enviados na URL usando o “?”, interrogação.

Identificados por ?variavel=valor

exemplo: `localhost:3000/posts/?variavel=2`

Mais de uma variavel: `localhost:3000/posts/?variavel=2&variavel2="aaa"`

Elas vão ficar localizadas em `req.query`

 Prisma

ex ExpressJS