

Reativa Tecnologia

Apostila de NodeJS 100% Gratuita

January 12, 2020

Esta apostila é o resultado de alguns meses de pesquisa, revisão e edição, foi baseada na documentação oficial (então as informações são confiáveis).

Este conteúdo eu iria colocar no como adicional no [Guia do Dev Autodidata](#) (que é o meu guia para **iniciantes**). Mas resolvi deixar **gratuito** a toda a comunidade, então aproveite e bons estudos!

Se tiver qualquer dúvida (ou sugestões) me envie no [Instagram](#), faço mentoria coletiva (gratuita) diariamente para programadores iniciantes que se sentem perdidos, você será extremamente bem vindo por lá (é só clicar na imagem).

**Quer uma
direção
na
carreira
como
<dev/>?**

Clique aqui.

Um grande abraço, Paulo Luan.

Dica: Aperte `ctrl+p` e imprima esse artigo, para que você possa fazer anotações!

Introdução ao Node.js

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, removendo o fardo de gerenciar concorrências em threads, que podem ser fontes significativas de bugs.

Node.js tem uma vantagem única porque milhões de desenvolvedores frontend que programam JavaScript para o navegador possam agora desenvolver código server-side (backend) além do client-side (frontend), sem a necessidade de aprender uma linguagem completamente diferente.

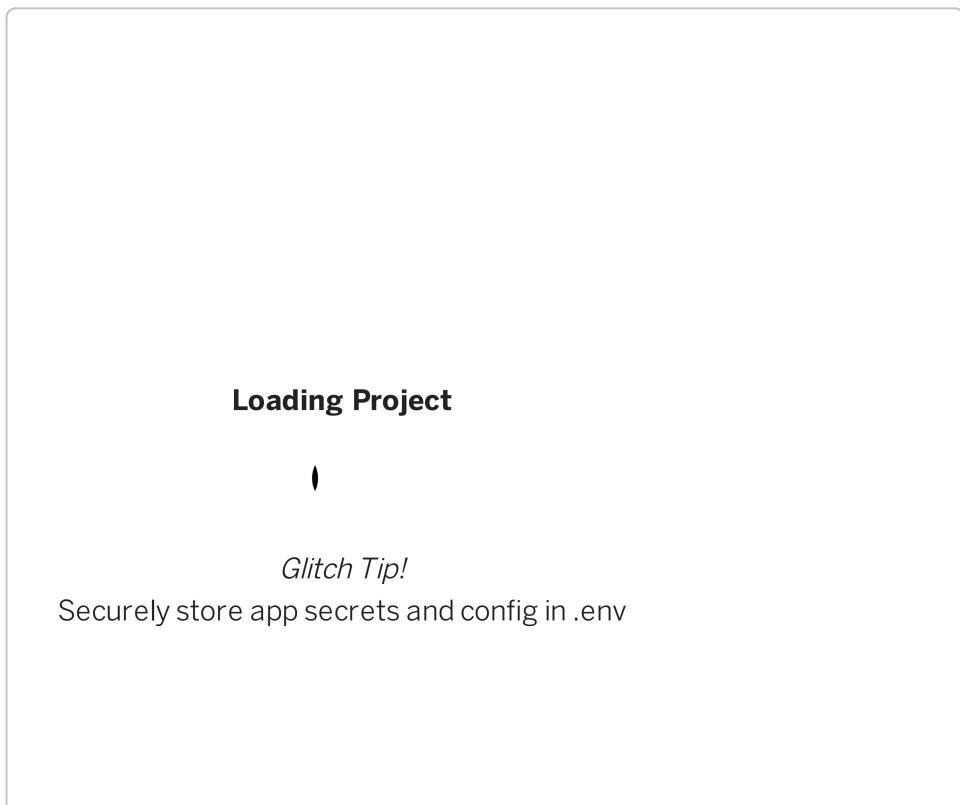
No Node.js os novos padrões do ECMAScript podem ser utilizados sem problemas, como você não precisa esperar todos os seus usuários atualizarem seus navegadores, você tem a liberdade de decidir qual versão do ECMAScript utilizar, bastando trocar a versão do Node.js, além de poder habilitar especificamente features experimentais, bastando executar o Node.js com as flags apropriadas.

Um número imenso de bibliotecas

O npm (node package manager) com sua simples estrutura ajudou na ploriferação do ecossistema Node.js, hospedando atualmente mais de 1,000,000 de pacotes open source que podem ser utilizados gratuitamente.

Um exemplo de aplicação Node.js

O exemplo mais comum de Hello World no Node.js é um servidor web:



```
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.

Primeiramente esse código importa o módulo `http`.

O Node.js tem uma [biblioteca padrão](#) fantástica, incluindo suporte de primeira classe para redes.

O método `createServer()` do `http` cria um novo servidor HTTP e o retorna.

É definido para o servidor escutar em uma porta e host name (nome de anfitrião, ao pé da letra) 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 com:

```
res.statusCode = 200
```

nós definimos a propriedade `statusCode` como 200, para indicar uma resposta bem sucedida.

Nós definimos a header de Content-Type (tipo de conteúdo):

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

e nós fechamos a resposta, adicionando o conteúdo como um argumento do `end()`:

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

Frameworks e ferramentas para Node.js

Node.js é uma plataforma de baixo nível. Para facilitar e inspirar os desenvolvedores, milhares de bibliotecas são construídas sob o Node.js pela comunidade.

Muitas delas se estabeleceram como opções populares ao decorrer do tempo. Aqui está uma pequena lista com algumas que valem a pena aprender:

- [AdonisJs](#): Um framework full-stack altamente focado na ergonomia, estabilidade e confiança do desenvolvedor. Adonis é um dos frameworks web Node.js mais rápidos.
- [Express](#): Provê os meios de se criar servidores web de uma forma muito simples porém poderosa. Têm uma pegada minimalista, não opinada, focada nas funções essenciais de um servidor, que são a chave do sucesso.
- [Fastify](#): Um framework altamente focado em prover a melhor experiência ao desenvolvedor, com o mínimo de gargalos na performance e uma poderosa arquitetura de extensões (plugins). Fastify é um dos mais rápidos frameworks Node.js.
- [Gatsby](#): Um gerador de sites estáticos baseado em [React](#), gerido com [GraphQL](#) e com um ecossistema de plugins e templates iniciais requíssimo.

- [**hapi**](#): Um rico framework para construção de aplicações e serviços que habilitam desenvolvedores a focar na escrita das lógicas de aplicação reutilizáveis em vez de perder tempo montando a infraestrutura.
- [**koa**](#): É mantido pelo time por trás do Express, pretende ser ainda menor e simples, construído sob muitos anos de conhecimento. O novo projeto nasceu da necessidade de criar mudanças incompatíveis sem romper com a comunidade existente.
- [**Loopback.io**](#): Faz com que seja fácil construir aplicações modernas que requerem integrações complexas.
- [**Meteor**](#): Um framework full-stack incrivelmente poderoso, permite que você crie aplicações Javascript com uma abordagem isomórfica, compartilhando código entre cliente e servidor. É uma ferramenta generalista que tenta fornecer tudo, que agora integra com bibliotecas frontend como [React](#), [Vue](#), e [Angular](#). Também pode ser utilizado para criar aplicativos mobile.
- [**Micro**](#): Provê um servidor enxuto para criação de microserviços HTTP assíncronos.
- [**NestJS**](#): Um framework Node.js progressivo baseado em Typescript para construção de aplicação de nível empresarial eficientes, confiáveis e escaláveis.
- [**Next.js**](#): Framework [React](#) que fornece a melhor experiência de desenvolvimento, com todos os recursos que você precisa para produção: renderização híbrida entre servidor (SSR) e estáticos, suporte ao TypeScript, bundle otimizado, rotas pre-fetching, e mais.
- [**Nx**](#): Um conjunto de ferramentas para desenvolvimento full-stack em monorepo, utilizando NestJS, [React](#), [Angular](#), e mais! Nx ajuda a escalar seu desenvolvimento de um time construindo uma aplicação para múltiplos times colaborando em múltiplas aplicações!
- [**Sapper**](#): Sapper é um framework para construção de aplicações web de todos os tamanhos, com uma bela experiência de desenvolvimento e roteamento flexível baseado em filesystems. Oferece SSR e muito mais!

- Socket.io: Um motor de comunicação em tempo real para construir aplicações em rede.
- Strapi: Strapi é um CMS flexível, open-source e independente, que fornece aos desenvolvedores a liberdade de escolher seus frameworks e ferramentas favoritos, enquanto também permite que editores administrem e distribuam seus conteúdos de maneira fácil. Por fazer o painel administrativo uma API extensível através de um sistema de plugins, o Strapi permite que as maiores empresas do mundo acelerem a entrega de conteúdo enquanto constroem belas experiências digitais.

Uma breve história do Node.js

Acredite ou não, Node.js só tem 11 anos de idade.

Em comparação, o JavaScript tem 24 anos e a Web tem 31 anos.

Na tecnologia, 11 anos não é um tempo muito longo, mas o Node.js parece ter existido sempre.

Eu tive o prazer de trabalhar com Node.js desde os primórdios quando ele só tinha 2 anos, e apesar das informações limitadas pela web, você já podia sentir que o crescimento seria enorme.

Nesse post, vamos dissecar a história do Node.js, para colocar as coisas em perspectiva.

Um pouco de história

JavaScript é uma linguagem de programação que foi criada no Netscape como uma ferramente de scripts para manipulação de páginas web dentro do browser Netscape Navigator.

Parte do modelo de negócios da Netscape era vender servidores web, o que incluia um ambiente chamado *Netscape LiveWire*, que podia criar páginas dinâmicas usando JavaScript no server-side. Infelizmente, o *Netscape LiveWire* não obteve muito sucesso e JavaScript server-side não era popular até recentemente, com a introdução do Node.js.

Um fator chave que levou o Node.js ao topo foi o timing. Apenas alguns poucos anos antes, Javascript começou a ser mais considerado como uma linguagem séria, graças às aplicações “Web 2.0” (como Flickr, Gmail, etc.) que mostraram ao mundo como poderia ser uma experiência moderna na web.

Motores JavaScript (engines) também se tornaram consideravelmente melhores com a competição entre vários browsers para fornecer aos usuários a melhor performance. Times de desenvolvimento por trás dos maiores browsers trabalharam duro para oferecer melhor suporte para o JavaScript e encontrar meios que o fizesse rodar mais rápido. A engine que roda por baixo dos panos do Node.js, V8 (também conhecida como Chrome V8 por ser a engine Javascript open-source do Projeto Chromium), melhorou显著mente devido a esta competição.

Aconteceu de o Node.js ser criado no lugar certo e na hora certa, mas sorte não é a única razão do porquê ser tão popular hoje. Ele introduz várias abordagens e estratégias inovadoras para o desenvolvimento sever-side com JavaScript que já ajudaram diversos desenvolvedores.

2009

- Nasce o Node.js
- A primeira forma do [npm](#) é criada

2010

- Nasce o [Express](#)
- Nasce o [Socket.io](#)

2011

- npm alcança a versão 1.0
- Grandes empresas começam a adotar Node.js: LinkedIn, Uber, etc.
- Nasce o [hapi](#)

2012

- Adoção continua muito rápida

2013

- Primeira grande plataforma de blogs usando Node.js: [Ghost](#)
- Nasce o [Koa](#)

2014

- O Grande Fork: [io.js](#) é o maior fork do Node.js, com o objetivo de introduzir suporte ao ES6 e crescer rapidamente.

2015

- Nasce a [Node.js Foundation](#)
- IO.js é mergeado de volta ao Node.js

- npm introduz módulos privados
- Node.js 4 (versões 1, 2 e 3 nunca foram lançadas previamente)

2016

- O incidente de [leftpad](#)
- Nasce o [Yarn](#)
- Node.js 6

2017

- npm aumenta o foco em segurança
- Node.js 8
- HTTP/2
- V8 introduz o Node.js em sua suite de testes, fazendo do Node.js oficialmente um alvo da engine, em adição ao Chrome
- 3 bilhões de downloads no npm toda semana

2018

- Node.js 10
- [ES modules](#): suporte experimental ao .mjs
- Node.js 11

2019

- Node.js 12
- Node.js 13

2020

- Node.js 14
- Node.js 15

Como instalar o Node.js

Node.js pode ser instalado de diferentes formas. Esse post destaca as mais comuns e convenientes.

Pacotes oficiais para todas as principais plataformas estão disponíveis em <https://nodejs.org/en/download/>.

Uma maneira muito conveniente de instalar o Node.js é através de um gerenciador de pacotes. Neste caso, cada sistema operacional tem sua abordagem mais adequada.

No macOS, [Homebrew](#) é a alternativa oficial, e - uma vez instalado - permite que a instalação do Node.js seja feita facilmente, rodando esse comando na CLI:

```
brew install node
```

Outros gerenciadores de pacote para Linux e Windows estão listados em <https://nodejs.org/en/download/package-manager/>.

`nvm` é um maneira muito popular de rodar Node.js. Ele permite que você troque facilmente entre versões, e instale novas versões para testar e troque de volta caso algo pare de funcionar, por exemplo.

Também é muito útil para testar seu código com versões antigas do Node.js

Veja <https://github.com/creationix/nvm> para mais informações sobre essa opção.

Minha sugestão é utilizar o instalador oficial se você só estiver começando e não utiliza o Homebrew, caso contrário, o Homebrew é a minha solução favorita.

Em todo caso, quando o Node.js é instalado você terá acesso ao comando executável `node` na linha de comando.

O que devo saber de Javascript para usar o Node.js?

Como um iniciante, é difícil chegar no ponto onde você é confiante o suficiente nas suas habilidades de programação.

Enquanto aprende a programar, você também pode ficar confuso sobre onde o JavaScript termina, e onde o Node.js começa, e vice versa.

Eu recomendo que você tenha um bom domínio dos principais conceitos do JavaScript antes de mergulhar no Node.js:

- Estrutura Léxica
- Expressões
- Tipos
- Variáveis
- Funções
- `this`
- Arrow Functions

- Loops
- Escopos
- Arrays
- Template Literals
- Semicolons (;)
- Strict Mode
- ECMAScript 6, 2016, 2017

Com esses conceitos em mente, você está no caminho certo para se tornar um desenvolvedor proficiente em JavaScript, tanto Browser como também Node.js.

Os conceitos a seguir também são essenciais para entender programação assíncrona, que é uma parte fundamental do Node.js:

- Programação Assíncrona e callbacks
- Timers
- Promises
- Async e Await
- Closures
- Event Loop

Qual a diferença do Node.JS e do Browser?

Ambos browser e Node.js utilizam JavaScript como sua linguagem de programação.

Construir aplicações que rodem no browser é uma coisa completamente diferente de construir aplicações Node.js

Apesar do fato que é sempre JavaScript, há fatores chave que tornam a experiência radicalmente diferente.

Da perspectiva de um desenvolvedor frontend que usa JavaScript extensivamente, aplicações Node.js trazem consigo uma enorme vantagem: o conforto de programar tudo - o frontend e o backend - em uma única linguagem.

Você tem uma grande oportunidade porque nós sabemos quão difícil é para aprender, completa e profundamente, uma nova linguagem de programação, e por usar a mesma linguagem para fazer todo o trabalho na web - tanto no servidor quanto no cliente, você está em uma posição única de vantagem.

O que muda é o ecossistema.

No browser, na maioria do tempo o que você está fazendo é interagindo com o DOM, ou outras APIs Web como Cookies. Isso não existe no Node.js, é claro. Você não tem o `document`, `window` e todos os outros objetos que são providos pelo browser.

E no browser, nós não temos as APIs legais que o Node.js provê com seus módulos, como a funcionalidade de acesso ao filesystem.

Outra grande diferença é que no Node.js você controla seu ambiente. A não ser que você esteja criando uma aplicação open source que qualquer um pode hospedar em qualquer lugar, você sabe em qual versão do Node.js a aplicação vai rodar. Comparado ao ambiente do browser, onde você não tem o luxo de escolher qual browser seu visitante vai utilizar, isso é muito conveniente.

Isso significa que você pode escrever códigos com os modernos ES6-7-8-9, se atentando ao suporte da sua versão do Node.js.

Visto que o JavaScript se move muito rápido, mas os browsers podem ser um pouco lentos para atualizarem, as vezes na web, você está preso em versões velhas do JavaScript / ECMAScript.

Você pode utilizar o Babel para transformar seu código em um formato compátivel com ES5 antes de enviar para o browser, mas no Node.js, você não precisa disso.

Outra diferença é que o Node.js utiliza o sistema de módulos CommonJS, enquanto que nos browsers ainda estamos vendo o inicio da implementação do padrão ES Modules.

Na prática, isso significa que por enquanto você utiliza `require()` no Node.js e `import` no browser.

O motor V8 do Javascript

V8 é o nome da engine JavaScript que roda no Google Chrome. É a coisa que pega nosso JavaScript e o executa enquanto navegamos com o Chrome.

V8 provê o ambiente de execução em que o JavaScript executa. O DOM, e outras APIs web são fornecidas pelo browser.

O legal é que a engine JavaScript é independente do browser que ela está hospedada. Essa funcionalidade chave possibilitou a ascensão do Node.js. A V8 foi escolhida como engine por trás do Node.js em 2009, e com a explosão de popularidade do Node.js, a V8 se tornou a engine que agora possibilita uma quantidade incrível de código sever-side escrito em JavaScript.

O ecossistema Node.js é enorme e isso graças à V8 que também possibilitou aplicações desktop, com projetos como o Electron.

Outras engines JS

Outros browsers têm suas próprias engines:

- Firefox tem a [SpiderMonkey](#)
- Safari tem a [JavaScriptCore](#) (também chamada de Nitro)
- Edge foi originalmente baseado na [Chakra](#) mas recentemente [foi refeito utilizando Chromium](#) e a V8.

e existem muitas outras também.

Todas essas engines implementam o [padrão ECMA ES-262](#), também chamado de ECMAScript, o padrão utilizado pelo JavaScript.

A busca por performance

V8 foi escrita em C++, e é continuamente melhorada. É portável e roda no Mac, Windows, Linux e diversos outros sistemas.

Nessa introdução à V8, vamos ignorar os detalhes de implementação: eles podem ser encontrados em sites mais apropriados (por exemplo, o [site oficial da V8](#)), e eles mudam com o passar do tempo, frequentemente.

V8 está sempre evoluindo, assim como as outras engines JavaScript ao seu redor, para agilizar a Web e o ecossistema Node.js.

Na web, há uma corrida por performance que vem sendo travada por anos, e nós (como usuários e desenvolvedores) nos beneficiamos muito por essa competição, porque nós possuímos máquinas mais rápidas e otimizadas ano a ano.

Compilação

JavaScript é geralmente considerado como uma linguagem interpretada, mas engines modernas de JavaScript não o interpretam apenas, elas o compilam.

Isso vem acontecendo desde 2009, quando o compilador JavaScript SpiderMonkey foi adicionado no Firefox 3.5, e todo mundo seguiu essa ideia.

JavaScript é internamente compilado pela v8 com **compilação just-in-time** (JIT) para acelerar a execução.

Isso pode parecer contra-intuitivo, mas desde a introdução do Google Maps em 2004, o JavaScript evoluiu de uma linguagem que geralmente executava poucas dúzias de linhas de código, para aplicações completas com centenas de milhares de linhas de código executando no browser.

Nossas aplicações agora podem rodar por horas dentro do browser, em vez de uma simples validação de regras de formulários ou scripts banais.

Nesse *novo mundo*, compilar JavaScript faz total sentido porque, embora possa demorar um pouco mais para termos o código JavaScript *pronto*, uma vez concluída a compilação temos muito mais desempenho do que código puramente interpretado.

Rode scripts do Node a partir da linha de comando do Linux

A maneira mais usual de rodar um programa Node.js é executando o comando `node` disponível globalmente (uma vez instalado o Node.js) e passando o nome do arquivo desejado.

Considerando que o arquivo principal da sua aplicação Node.js se chame `app.js`, você pode executá-lo digitando:

```
node app.js
```

Enquanto executa o comando, tenha certeza de estar no mesmo diretório que contém o arquivo `app.js`.

Como sair de um programa Node.JS

Há várias maneiras de finalizar uma aplicação Node.js.

Com o programa rodando no console, você pode fechá-lo com `ctrl-C`, mas o que queremos discutir aqui é a maneira programática de fazer isso.

Vamos começar com a maneira mais drástica, e note porque é melhor você *não* utilizá-la:

O módulo nativo `process` provê um método prático que permite a você, programaticamente, sair de um programa Node.js: `process.exit()`.

Quando o Node.js executa essa linha, o processo é forçado a terminar imediatamente.

Isso significa que qualquer callback pendente, qualquer request de rede sendo enviada, qualquer acesso ao filesystem, ou processos de escrita no `stdout` ou `stderr` - tudo será finalizado de imediato *desgraciadamente*.

Se isso está bem para você, basta passar um inteiro que sinalize o código de saída para o sistema operacional:

```
process.exit(1)
```

Por padrão, o código de saída é `0`, que significa sucesso. Códigos de saída diferentes tem significados diferentes, que você pode querer utilizar no seu sistema para se comunicar com outros programas.

Você pode ler mais sobre os códigos de saída em
https://nodejs.org/api/process.html#process_exit_codes;

Você também pode definir a propriedade `process.exitCode` como:

```
process.exitCode = 1
```

e quando o programa for encerrado posteriormente, o Node.js retornará esse código de saída.

Um programa irá sair graciosamente quando todos os processos estiverem completos.

Muitas vezes nós iniciamos servidores Node.js, como esse servidor HTTP:

```
const express = require('express')
const app = express()

app.get('/', (req, res) => {
  res.send('Hi!')
})

app.listen(3000, () => console.log('Server ready'))
```

Esse programa nunca terá um fim. Se você chamar `process.exit()`, qualquer requisição corrente/pendente será abortada. Isso *não é legal*.

Nesse caso, você precisa enviar ao comando um sinal de SIGTERM, e lidar com o processo desse sinal:

Nota: process não requer importação, está disponível automaticamente.

```
const express = require('express')
const app = express()

app.get('/', (req, res) => {
  res.send('Hi!')
})
```

```
const server = app.listen(3000, () => console.log('Server ready'))  
  
process.on('SIGTERM', () => {  
  server.close(() => {  
    console.log('Process terminated')  
  })  
})
```

O que são sinais? Sinais são um sistema de intercomunicação POSIX: uma notificação enviada para um processo com o objetivo de notificar que um evento ocorreu.

- `SIGKILL` é o sinal que diz ao processo para que finalize imediatamente, e agirá idealmente como o `process.exit()`.
- `SIGTERM` é o sinal que diz ao processo para que termine graciosamente. É o sinal que é enviado por gerenciadores de processo como `upstart` ou `supervisord` e muitos outros.

Você pode enviar esse sinal por dentro da aplicação, em outra função:

```
process.kill(process.pid, 'SIGTERM')
```

ou de outro programa Node.js em execução, ou qualquer outra aplicação rodando em seu sistema que saiba o PID do processo que você deseja finalizar.

Como ler Variáveis de ambiente a partir do Node.js?

O módulo nativo `process` do Node.js fornece a propriedade `env` que armazena todas as variáveis de ambiente que foram definidas no momento de início da aplicação.

Aqui vai um exemplo que acessa a variável de ambiente `NODE_ENV`, que é definida com o valor `development` por padrão.

Nota: `process` não requer importação, está disponível automaticamente.

```
process.env.NODE_ENV // "development"
```

Defini-lá para “production” antes do script executar contará ao Node.js que é um ambiente de produção.

Da mesma forma, você pode acessar/definir qualquer variável de ambiente customizada.

Como usar o REPL no Node

Nós usamos o comando `node` para executar scripts Node.js:

```
node script.js
```

Se nós omitirmos o nome do arquivo, iniciaremos o modo REPL:

```
node
```

Nota: REPL (Read Evaluate Print Loop) é um ambiente para linguagem de programação (basicamente uma aba de console) que lê instruções individuais do input do usuário e após a execução, retorna o resultado no console.

Se você tentar isso agora no seu terminal, isto é o que vai acontecer:

```
> node
```

```
>
```

o comando permanece em modo de espera (`idle`) e aguarda alguma entrada.

Dica: se você não tem certeza de como abrir seu terminal, pesquise por “How to open terminal on <seu SO>“.

Para ser mais exato, o REPL está aguardando a entrada de código JavaScript.

Vamos começar de forma simples:

```
> console.log('test')
test
undefined
>
```

O primeiro valor, `test`, é a saída que esperavamos do log do console, e depois tivemos um `undefined`, que é o retorno de rodar `console.log()`.

Agora podemos inserir uma nova entrada de JavaScript.

Use o tab para autocomplete

O legal do REPL é que ele é interativo.

Conforme você escreve seu código, se você pressionar o tecla `tab` o REPL vai tentar completar o que foi escrito para combinar com variáveis previamente definidas.

Explorando objetos do JavaScript

Tente digitar o nome de uma classe do JavaScript, como `Number`, e adicione um ponto e pressione `tab`.

O REPL vai listar todas as propriedades e métodos que você pode acessar naquela classe:

```
node-handbook: node
> Number.
Number.__defineGetter__ Number.__defineSetter__ Number.__lookupGetter__
Number.__lookupSetter__ Number.__proto__ Number.constructor
Number.hasOwnProperty Number.isPrototypeOf Number.propertyIsEnumerable
Number.toLocaleString Number.toString Number.valueOf

Number.apply Number.arguments Number.bind
Number.call Number.caller Number.length
Number.name

Number.EPSILON Number.MAX_SAFE_INTEGER Number.MAX_VALUE
Number.MIN_SAFE_INTEGER Number.MIN_VALUE Number.NEGATIVE_INFINITY
Number.NaN Number.POSITIVE_INFINITY Number.isFinite
Number.isInteger Number.isNaN Number.isSafeInteger
Number.parseFloat Number.parseInt Number.prototype

> |
```

Explore objetos globais

Você pode inspecionar os objetos globais que você tem acesso digitando `global.` e pressionando `tab`:

```
node-handbook: node
> global.
global.__defineGetter__ global.__defineSetter__ global.__lookupSetter__
global.__lookupSetter__ global.__proto__ global.constructor
global.hasOwnProperty global.isPrototypeOf global.toLocaleString
global.propertyIsEnumerable global.toString global.valueOf

global.Array global.ArrayBuffer
global.Boolean global.Buffer
global.DTRACE_HTTP_CLIENT_REQUEST global.DTRACE_HTTP_CLIENT_RESPONSE
global.DTRACE_HTTP_SERVER_REQUEST global.DTRACE_HTTP_SERVER_RESPONSE
global.DTRACE_NET_SERVER_CONNECTION global.DTRACE_NET_STREAM_END
global.DataView global.Date
global.Error global.EvalError
global.Float32Array global.Float64Array
global.Function global.GLOBAL
```

```
global.Infinity          global.Int16Array
global.Int32Array        global.Int8Array
global.Intl               global.JSON
global.Map                global.Math
global.NaN                 global.Number
global.Object              global.Promise
global.Proxy                global.RangeError
global.ReferenceError      global.Reflect
global.RegExp               global.Set
global.String              global.Symbol
global.SyntaxError         global.TypeError
global.URIError             global.Uint16Array
global.Uint32Array          global.Uint8Array
global.Uint8ClampedArray    global.WeakMap
global.WeakSet               global.WebAssembly
```

A variável especial `_`

Se depois de um código você digitar `_`, isso fará com que seja exibido o resultado da última operação.

Comandos com ponto (dot commands)

O REPL tem alguns comandos especiais, todos começando com um `.`. Eles são:

- `.help`: exibe a guia de ajuda dos dot commands.
- `.editor`: habilita o modo de editor, para escrever múltiplas linhas de código JavaScript com facilidade. Uma vez nesse modo, pressione `ctrl-D` para executar o código que foi escrito.
- `.break`: quando estiver inserindo um código com múltiplas linhas, utilizar o comando `.break` fará com que a entrada seja abortada. Mesma funcionalidade de pressionar `ctrl-C`.
- `.clear`: reinicia o contexto do REPL para um objeto vazio e cancela qualquer entrada corrente de múltiplas linhas.
- `.load`: carrega um arquivo JavaScript, relativo ao diretório atual.

- `.save`: salva todas suas entradas na sessão REPL em um arquivo (especifique um nome pro arquivo)
- `.exit`: sai do repl (mesmo funcionamento de pressionar ctrl-C duas vezes)

O REPL sabe quando você está inserindo uma entrada com múltiplas linhas sem a necessidade de utilizar o `.editor`.

Por exemplo, se você começar a digitar uma iteração como essa:

```
[1, 2, 3].forEach(num => {
```

e pressionar `enter`, o REPL irá para uma nova linha que começa com 3 pontos, indicando que você pode continuar a trabalhar naquele bloco.

```
... console.log(num)  
... })
```

Se você digitar `.break` no fim de uma linha, o modo múltiplas linhas irá parar e o código não será executado.

O Node.JS aceita argumentos a partir da linha de comando.

Você pode passar qualquer quantidade de argumentos quando está invocando uma aplicação Node.js, usando:

```
node app.js
```

Argumentos podem ser com chave e valor ou diretos (sem chave).

Por exemplo:

```
node app.js joe
```

ou

```
node app.js name=joe
```

Isso muda em como você obterá os valores no código Node.js.

A maneira de obtê-los é usando o objeto nativo `process`.

Ele expõe uma propriedade chamada `argv`, que é um array que contém todos os argumentos passados na invocação.

O primeiro elemento é o caminho absoluto do comando `node`.

O segundo é o caminho absoluto do arquivo em execução.

Todos os argumentos adicionais estão presentes da terceira posição em diante.

Você pode iterar sobre todos os argumentos (incluindo o caminho do node e o do arquivo) usando um loop:

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

ou obter apenas os argumentos adicionais, criando um novo array que exclui os 2 primeiros parâmetros:

```
const args = process.argv.slice(2)
```

Se você tem um argumento direto (sem chave), como esse:

```
node app.js joe
```

você pode acessá-lo usando

```
const args = process.argv.slice(2)  
args[0]
```

Já nesse caso:

```
node app.js name=joe
```

`args[0]` é `name=joe`, e você precisa tratá-lo. A melhor maneira de fazer isso é usando a biblioteca [minimist](#), que ajuda a lidar com argumentos:

```
const args = require('minimist')(process.argv.slice(2))  
args['name'] //joe
```

Desta vez você precisa usar dois traços antes do nome do argumento:

```
node app.js --name=joe
```

Output para a linha de comando usando Node.JS

Saída (output) básico usando o módulo console

O Node.js provê o [módulo console](#) que possui uma infinidade de maneiras muito úteis para interagir com a linha de comando.

É basicamente o mesmo objeto `console` encontrado no browser.

O método mais básico e mais usado é o `console.log()`, que imprime a string que você passar como parâmetro.

Se você passar um objeto, ele irá renderizá-lo como uma string.

Você pode passar múltiplas variáveis para o `console.log`, por exemplo:

```
const x = 'x'  
const y = 'y'  
console.log(x, y)
```

o Node.js vai printar ambas.

Nós também podemos formatar mensagens mais sofisticadas passando variáveis e um formato específico.

Por exemplo:

```
console.log('My %s has %d years', 'cat', 2)
```

- `%s` formata a variável como uma string
- `%d` formata a variável como um número
- `%i` formata a variável como um número, porém só a parte inteira
- `%o` formata a variável como um objeto

Exemplo:

```
console.log('%o', Number)
```

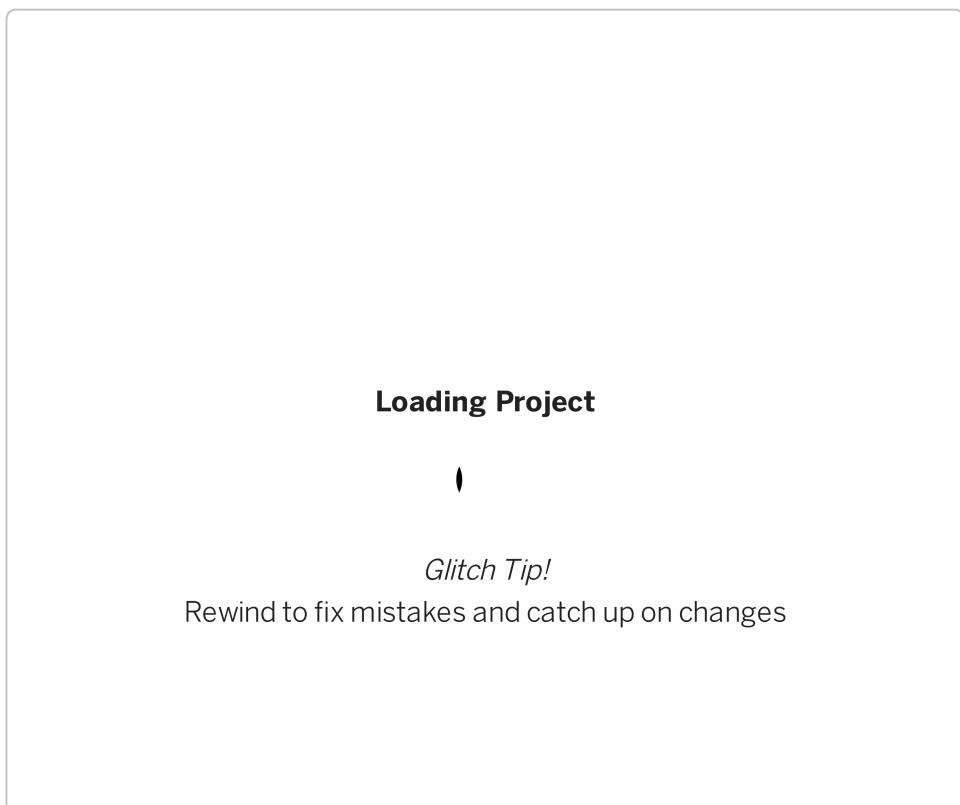
Limpando o console

`console.clear()` limpa o console (o comportamento pode mudar dependendo do console utilizado).

Contando elementos

`console.count()` é um método muito conveniente.

Dado o código:



```
const x = 1
const y = 2
const z = 3
console.count(
  'O valor de x é ' + x + ' e foi validado .. quantas vezes?'
)
console.count(
```

```
'O valor de x é ' + x + ' e foi validado .. quantas vezes?'
)
console.count(
  'O valor de y é ' + y + ' e foi validado .. quantas vezes?'
)
```

O que acontece é que ele vai contar o número de vezes que uma string foi impressa, e imprime o total:

Você pode contar maçãs e laranjas, por exemplo:

```
const oranges = ['orange', 'orange']
const apples = ['just one apple']
oranges.forEach(fruit => {
  console.count(fruit)
})
apples.forEach(fruit => {
  console.count(fruit)
})
```

Imprimindo a pilha de execução

Existem casos onde é útil imprimir a pilha de execução da função, talvez para responder a clássica pergunta *como você chegou nessa parte do código?*

Você pode isso fazer usando `console.trace()`:

```
const function2 = () => console.trace()
const function1 = () => function2()
function1()
```

Isso irá imprimir a pilha de execução. Isso é o que é impresso se tentarmos usá-lo no REPL do Node.js:

Trace

```
    at function2 (repl:1:33)
    at function1 (repl:1:25)
    at repl:1:1
    at ContextifyScript.Script.runInThisContext (vm.js:44:33)
    at REPLServer.defaultEval (repl.js:239:29)
    at bound (domain.js:301:14)
    at REPLServer.runBound [as eval] (domain.js:314:12)
    at REPLServer.onLine (repl.js:440:10)
    at emitOne (events.js:120:20)
    at REPLServer.emit (events.js:210:7)
```

Calculando o tempo gasto

Você pode calcular facilmente o tempo que uma função gasta para rodar, usando `time()` e `timeEnd()`

```
const doSomething = () => console.log('test')
const measureDoingSomething = () => {
  console.time('doSomething()')
  //faça algo, e calcule o tempo que isso levou
  doSomething()
  console.timeEnd('doSomething()')
}
measureDoingSomething()
```

stdout e stderr

Como nós vimos o `console.log` é ótimo para imprimir mensagens no console. Isso é o que chamamos de saída padrão, ou `stdout`.

`console.error` imprime na stream `stderr`.

Isso não vai aparecer no console, mas vai aparecer no log de erro.

Colorindo o saída

Você pode colorir a saída do seu texto no console utilizando [escape sequences](#), que basicamente são um conjunto de caracteres que identificam uma cor.

Exemplo:

```
console.log('\x1b[33m%s\x1b[0m', 'hi!')
```

Você pode testar isso no REPL no Node.js, onde será impresso `hi!` em amarelo.

Entretanto, essa é uma abordagem mais baixo nível. O jeito mais simples de colorir saídas no console é utilizando uma biblioteca. [Chalk](#) é uma biblioteca que além de colorir, também ajuda com outras facilidades de estilização, como deixar textos em negrito, ítalo ou sublinhados.

Instale com `npm install chalk`, e então use-o assim:

```
const chalk = require('chalk')
console.log(chalk.yellow('hi!'))
```

Usando `chalk.yellow` é muito mais conveniente do que tentar lembrar o código de cor correto, e o código fica muito mais legível.

Confira o link do projeto postado acima para mais exemplos de uso.

Crie uma barra de progresso

[Progress](#) é um pacote incrível para criar uma barra de progresso no console.

Instale utilizando `npm install progress`.

Esse trecho cria uma barra de progresso com 10 passos, e a cada 100ms um passo é completado. Quando a barra é completada nós finalizamos o contador.

```
const ProgressBar = require('progress')

const bar = new ProgressBar(':bar', { total: 10 })
const timer = setInterval(() => {
  bar.tick()
  if (bar.complete) {
    clearInterval(timer)
  }
}, 100)
```

Aceite inputs do terminal no Node.js

Como criar um programa Node.js de CLI interativa?

Desde a versão 7 o Node.js possui o [módulo readline](#) para fazer exatamente isso: obter entradas de uma stream de leitura, como a `process.stdin`, via terminal, durante a execução de um programa Node.js, uma linha por vez.

```
const readline = require('readline').createInterface({
  input: process.stdin,
  output: process.stdout
})

readline.question(`What's your name?`, name => {
  console.log(`Hi ${name}!`)
  readline.close()
})
```

Esse trecho de código pergunta o nome do usuário, e uma vez que o texto é inserido e o usuário pressiona enter, nós enviamos uma saudação.

O método `question()` exibe o primeiro parâmetro (a pergunta) e aguarda pela entrada do usuário. A função de callback é invocada uma vez que o enter é pressionado.

Nessa função de callback, nós fechamos a interface do readline.

`readline` oferece diversos outros métodos, e você pode conferir todos na documentação linkada acima.

Se você precisa solicitar uma senha, o ideal é que os caracteres digitados sejam trocados pelo símbolo de `*`.

A maneira mais simples de fazer isso é utilizar o [pacote readline-sync](#), que é muito similar em termos de API.

Outra solução mais completa e abstrata é fornecida pelo [pacote Inquirer.js](#).

Instale-o utilizando `npm install inquirer`, e então você poderá replicar o código acima dessa forma:

```
const inquirer = require('inquirer')

var questions = [
  {
    type: 'input',
    name: 'name',
    message: "What's your name?"
  }
]

inquirer.prompt(questions).then(answers => {
  console.log(`Hi ${answers['name']}!`)
})
```

Inquirer.js permite com que você faça diversas coisas como perguntas de múltipla escolha, ter radio buttons, confirmações e muito mais.

Vale a pena conhecer todas as alternativas, especialmente as nativas do Node.js, mas se você planeja levar a CLI para outro nível, Inquirer.js é ótima escolha.

Expondo uma funcionalidade em um arquivo do Node.js usando exports

O Node.js tem um sistema de módulos nativo.

Um arquivo Node.js pode importar funcionalidades expostas por outros arquivos Node.js.

Quando você quer importar algo você deve utilizar

```
const library = require('./library')
```

para importar a funcionalidade exposta no arquivo `library.js` que reside na pasta atual.

Nesse arquivo, a funcionalidade deve ser exposta antes de poder ser importada por outros arquivos.

Qualquer outra variável ou objeto definido no arquivo é privado por padrão e não exposto ao mundo exterior.

Isso é permitido por meio da API `module.exports`, oferecida pelo [module system](#).

Quando você atribui um objeto ou uma função como uma nova propriedade do `exports`, isso se torna o que está sendo exposto, e como tal, pode ser importado em outras partes da sua aplicação, ou até em outras aplicações.

Você pode fazer isso de 2 formas.

A primeira é atribuir um objeto ao `module.exports`, que é um objeto provido nativamente pelo sistema de módulos, e ele fará com que o arquivo exporte *apenas aquele objeto*:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta'
}

module.exports = car

//..em outro arquivo

const car = require('./car')
```

A segunda é definir o objeto exportado como propriedade do `exports`. Essa abordagem permite que você exporte múltiplos objetos, funções ou dados:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta'
}

exports.car = car
```

ou diretamente

```
exports.car = {
  brand: 'Ford',
  model: 'Fiesta'
}
```

Em outro arquivo, você irá utilizá-lo referenciando a propriedade da sua importação:

```
const items = require('./items')
items.car
```

ou

```
const car = require('./items').car
```

Qual a diferença entre `module.exports` e `exports`?

O primeiro expõe o objeto para qual ele aponta. O último expõe *as propriedades* do objeto que ele aponta.

Uma introdução ao gerenciador de pacotes NPM

Introdução ao npm

`npm` é o gerenciador de pacotes padrão do Node.js.

Em janeiro de 2017 mais de 350.000 pacotes foram listados no registro do npm, fazendo dele o maior repositório de código de uma única linguagem na Terra, e você pode ter certeza que existe um pacote para (quase!) tudo.

Ele iniciou como um meio de fazer download e gerenciar dependências de pacotes Node.js, mas desde então ele se tornou uma ferramenta utilizada também no frontend.

Há muitas coisas que o `npm` faz.

Yarn é uma alternativa ao `npm`. Não deixe de conferir.

Downloads

`npm` gerencia downloads de dependências do seu projeto.

Instalando todas dependências

Se o projeto tem um arquivo `package.json`, ao rodar

```
npm install
```

ele vai instalar tudo que o projeto precisa, na pasta `node_modules`, criando-a se não existir.

Instalando um único pacote

Você também pode instalar um pacote específico ao rodar

```
npm install <package-name>
```

Geralmente você verá mais flags adicionadas a esse comando:

- `--save` instala e adiciona uma entrada no campo `dependencies` do arquivo `package.json`
- `--save-dev` instala e adiciona uma entrada no campo `devDependencies` do arquivo `package.json`

A principal diferença é que no `devDependencies` ficam as ferramentas de desenvolvimento, como uma biblioteca de testes, enquanto no `dependencies` ficam os pacotes necessários à aplicação em ambiente de produção.

Atualizando pacotes

Atualizar também é fácil, ao rodar

```
npm update
```

O `npm` vai buscar em todos os pacotes por uma versão atualizada que satisfaça suas restrições de versionamento.

Você também pode especificar um único pacote para atualizar:

```
npm update <package-name>
```

Versionamento

Em adição aos downloads, o `npm` também gerencia o **versionamento**, assim você pode especificar qualquer versão do pacote, ou uma versão maior ou menor do que você precisa.

Muitas vezes você vai encontrar uma biblioteca que só é compatível com a versão atual de outra biblioteca.

Ou um bug na versão mais atual da biblioteca, ainda não solucionado, causando um problema.

Especificar explicitamente a versão da biblioteca também ajuda a manter todos na exata mesma versão do pacote, assim todos do time rodam a mesma versão até que o arquivo `package.json` seja atualizado.

Em todos os casos, versionar ajuda muito, e o `npm` segue o padrão semântico de versionamento chamado semver.

Executando tarefas

O arquivo package.json possui um campo chamado “scripts”, que é usado para especificar tarefas de linha de comando que podem ser rodadas usando

```
npm run <task-name>
```

Por exemplo:

```
{
  "scripts": {
    "start-dev": "node lib/server-development",
    "start": "node lib/server-production"
  },
}
```

É muito comum usar esse recurso para rodar o Webpack:

```
{
  "scripts": {
    "watch": "webpack --watch --progress --colors --config webpack.config.js",
    "dev": "webpack --progress --colors --config webpack.conf.js",
    "prod": "NODE_ENV=production webpack -p --config webpack.conf.js"
  },
}
```

Então em vez de digitar esses comandos longos, que são muito fáceis de errar ou esquecer, você pode rodar

```
npm run watch
npm run dev
npm run prod
```

Onde o Node instala os pacotes?

Quando você instala um pacote utilizando `npm` você pode executar 2 tipos de instalação:

- local
- global

Por padrão, quando você digita o comando `npm install`, como por exemplo:

```
npm install lodash
```

o pacote será instalado na árvore de arquivos atual, em uma subpasta dentro da `node_modules`.

Quando isso ocorre, o `npm` também adiciona uma entrada do `lodash` na propriedade `dependencies` do arquivo `package.json` da pasta atual.

Para executar uma instalação global, basta utilizar a flag `-g`:

```
npm install -g lodash
```

Quando isso ocorre, o `npm` não instalará na pasta local, em vez disso, ele utilizará uma localização global.

Onde, exatamente?

O comando `npm root -g` te dirá a localização exata na sua máquina.

No macOS ou Linux, essa localização costuma ser
`/usr/local/lib/node_modules`. No Windows costuma ser
`C:\Users\YOU\AppData\Roaming\npm\node_modules`

Todavia, se você utiliza o `nvm` para gerenciar versões do Node.js, a localização pode ser diferente.

Eu por exemplo utilizo `nvm` e a localização dos meus pacotes foi exibida como
`/Users/joe/.nvm/versions/node/v8.9.0/lib/node_modules`.

Como usar ou executar um pacote instalado usando NPM?

Quando você instala um pacote usando `npm` dentro da sua pasta `node_modules`, ou também globamente, como você o utiliza em seu código Node.js?

Digamos que você instale a famosa biblioteca JavaScript de utilidades chamada `lodash`, usando

```
npm install lodash
```

Isso irá instalar o pacote na sua pasta `node_modules` local.

Para utilizá-lo em seu código, você só precisa importá-lo no seu programa utilizando `require`:

```
const _ = require('lodash')
```

Mas e se o seu pacote for um executável?

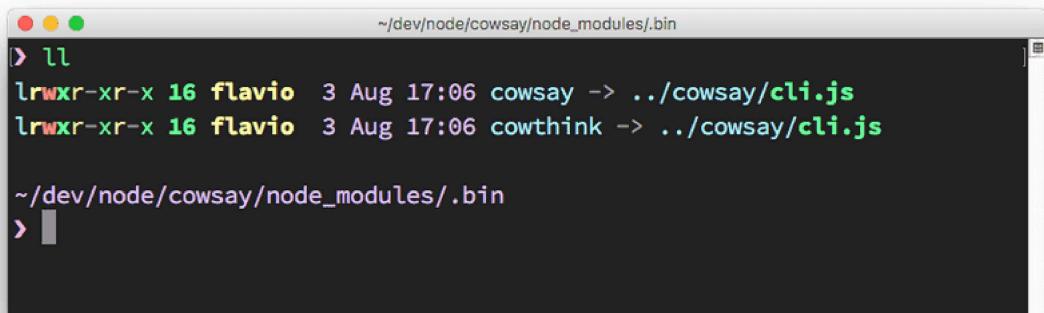
Nesse caso, ele para a pasta `node_modules/.bin/`.

Um jeito fácil de demonstrar isso é utilizando o [cowsay](#).

O pacote cowsay fornece um programa de linha de comando que faz com que uma vaca diga algo (e outros animais também 🐄).

Quando você instala o pacote utilizando `npm install cowsay`, ele irá instalar a si mesmo e algumas poucas dependências na sua pasta `node_modules` local:

Há uma pasta oculta chamada .bin, que contém links simbólicos para os binários do cowsay:



The screenshot shows a terminal window with the following output:

```
~/dev/node/cowsay/node_modules/.bin
> ll
lrwxr-xr-x 16 fladio 3 Aug 17:06 cowsay -> ../cowsay/cli.js
lrwxr-xr-x 16 fladio 3 Aug 17:06 cowthink -> ../cowsay/cli.js

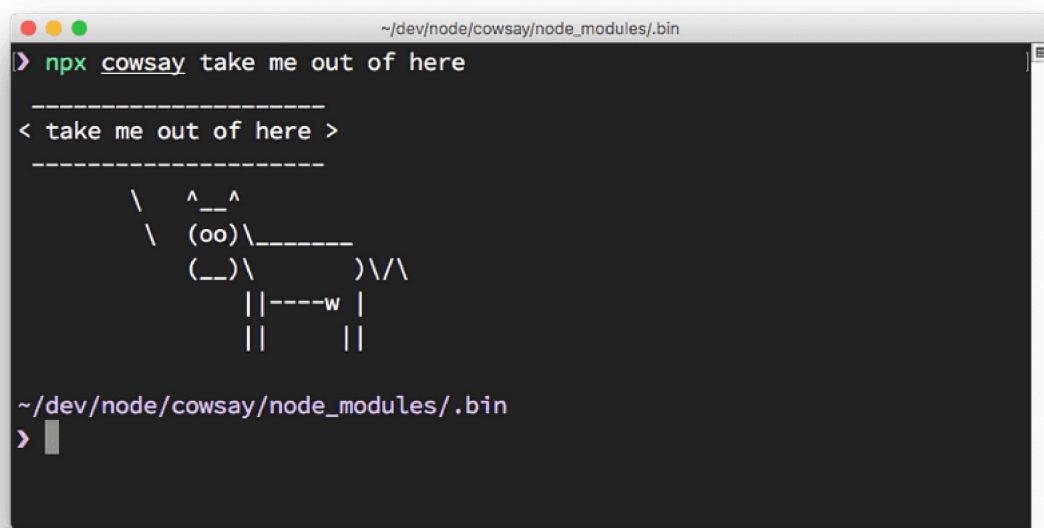
~/dev/node/cowsay/node_modules/.bin
>
```

Como você executa isso?

É claro que você pode digitar `./node_modules/.bin/cowsay` para executar, e vai funcionar, mas o npx, incluso nas versões recentes do npm (desde a 5.2), é uma opção muito melhor. Você só roda:

```
npx cowsay
```

e o npx vai encontrar a localização do pacote.



A screenshot of a terminal window titled 'npx cowsay take me out of here'. The terminal displays a cow ASCII art and the command prompt again.

O Guia do package.json

Se você trabalha com JavaScript, ou já interagiu com um projeto JavaScript, Node.js ou um projeto frontend, você certamente conheceu o arquivo `package.json`.

Pra que ele serve? O que você deveria saber sobre ele, e o quê de legal pode ser feito com ele?

O `package.json` é uma espécie de manifesto do seu projeto. Ele pode fazer uma variedade de coisas, completamente não relacionadas. É um repositório central de configurações para ferramentas, por exemplo. Também é onde o `npm` e o `yarn` armazenam os nomes e versões de todos os pacotes instalados.

The file structure

Aqui temos um exemplo de arquivo `package.json`:

{}

Está vazio! Não há campos fixos obrigatórios do que pode ser colocado no arquivo `package.json`, de uma aplicação. A única exigência é que respeite o formato JSON, caso contrário programas que tentem acessar as propriedades programaticamente não terão sucesso.

Se você está criando um pacote Node.js que precisa ser distribuído no `npm` as coisas mudam radicalmente, e você necessita preencher algumas propriedades que vão ajudar outras pessoas a utilizar seu pacote. Vamos ver mais sobre isso em breve.

Esse é outro `package.json`:

```
{
  "name": "test-project"
}
```

Ele define uma propriedade `name`, que diz o nome da aplicação, ou pacote, que está contido na mesma pasta em que o arquivo reside.

Aqui temos outro exemplo muito mais complexo, que foi extraído de uma aplicação Vue.js:

```
{
  "name": "test-project",
  "version": "1.0.0",
  "description": "A Vue.js project",
  "main": "src/main.js",
  "private": true,
  "scripts": {
    "dev": "webpack-dev-server --inline --progress --config build/webpack.config.js",
    "start": "npm run dev",
    "unit": "jest --config test/unit/jest.conf.js --coverage",
    "test": "npm run unit",
    "lint": "eslint --ext .js,.vue src test/unit",
    "build": "node build/build.js"
  },
}
```

```
"dependencies": {
    "vue": "^2.5.2"
},
"devDependencies": {
    "autoprefixer": "^7.1.2",
    "babel-core": "^6.22.1",
    "babel-eslint": "^8.2.1",
    "babel-helper-vue-jsx-merge-props": "^2.0.3",
    "babel-jest": "^21.0.2",
    "babel-loader": "^7.1.1",
    "babel-plugin-dynamic-import-node": "^1.2.0",
    "babel-plugin-syntax-jsx": "^6.18.0",
    "babel-plugin-transform-es2015-modules-commonjs": "^6.26.0",
    "babel-plugin-transform-runtime": "^6.22.0",
    "babel-plugin-transform-vue-jsx": "^3.5.0",
    "babel-preset-env": "^1.3.2",
    "babel-preset-stage-2": "^6.22.0",
    "chalk": "^2.0.1",
    "copy-webpack-plugin": "^4.0.1",
    "css-loader": "^0.28.0",
    "eslint": "^4.15.0",
    "eslint-config-airbnb-base": "^11.3.0",
    "eslint-friendly-formatter": "^3.0.0",
    "eslint-import-resolver-webpack": "^0.8.3",
    "eslint-loader": "^1.7.1",
    "eslint-plugin-import": "^2.7.0",
    "eslint-plugin-vue": "^4.0.0",
    "extract-text-webpack-plugin": "^3.0.0",
    "file-loader": "^1.1.4",
    "friendly-errors-webpack-plugin": "^1.6.1",
    "html-webpack-plugin": "^2.30.1",
    "jest": "^22.0.4",
    "jest-serializer-vue": "^0.3.0",
    "node-notifier": "^5.1.2",
    "optimize-css-assets-webpack-plugin": "^3.2.0",
    "ora": "^1.2.0",
    "portfinder": "^1.0.13",
    "postcss-import": "^11.0.0",
    "postcss-loader": "^2.0.8",
    "postcss-url": "^7.2.1",
}
```

```
    "rimraf": "^2.6.0",
    "semver": "^5.3.0",
    "shelljs": "^0.7.6",
    "uglifyjs-webpack-plugin": "^1.1.1",
    "url-loader": "^0.5.8",
    "vue-jest": "^1.0.2",
    "vue-loader": "^13.3.0",
    "vue-style-loader": "^3.0.1",
    "vue-template-compiler": "^2.5.2",
    "webpack": "^3.6.0",
    "webpack-bundle-analyzer": "^2.9.0",
    "webpack-dev-server": "^2.9.1",
    "webpack-merge": "^4.1.0"
  },
  "engines": {
    "node": ">= 6.0.0",
    "npm": ">= 3.0.0"
  },
  "browserslist": ["> 1%", "last 2 versions", "not ie <= 8"]
}
```

há muitas coisas rolando aqui:

- `version` indica a versão atual
- `name` define o nome da aplicação
- `description` é uma breve descrição da aplicação
- `main` define o ponto de entrada da aplicação
- `private` se definido como `true`, previne da aplicação ser publicada acidentalmente no `npm`
- `scripts` define alguns scripts node que você pode rodar na linha de comando
- `dependencies` define uma lista de pacotes `npm` instalados como dependências
- `devDependencies` define uma lista de pacotes `npm` instalados como dependências de desenvolvimento

- `engines` define em quais versões do Node.js essa aplicação funciona
- `browserslist` é usado para definir quais browsers (e suas versões) você quer dar suporte

Todas essas propriedades são usadas tanto pelo `npm` quanto outras ferramentas.

Separação de propriedades

Essa seção descreve as propriedades que você pode usar em detalhes. Nós nos referimos como “pacote” mas o mesmo se aplica à aplicações locais que você não utiliza como pacotes.

A maioria dessas propriedades são usadas apenas no <https://www.npmjs.com/>, outras por scripts que interagem com seu código, como o `npm` ou outros.

name

Define o nome do pacote.

Exemplo:

```
"name": "test-project"
```

O nome deve ter menos de 214 caracteres, não pode ter espaços, apenas letras minúsculas, hífens (-) ou underlines (_).

Isso ocorre porque quando um pacote é publicado no `npm`, ele ganha sua própria URL baseada nessa propriedade.

Se você publicou esse pacote no GitHub, é uma boa definir essa propriedade com o nome do repositório no GitHub.

author

Lista o nome do autor do pacote

Exemplo:

```
{  
  "author": "Joe <joe@whatever.com> (https://whatever.com)"  
}
```

Também pode ser usado com esse formato:

```
{  
  "author": {  
    "name": "Joe",  
    "email": "joe@whatever.com",  
    "url": "https://whatever.com"  
  }  
}
```

contributors

Assim como o autor, o projeto pode ter muitos contribuidores. Essa propriedade é um array que os lista.

Exemplo:

```
{  
  "contributors": ["Joe <joe@whatever.com> (https://whatever.com)"]  
}
```

Também pode ser usado com esse formato:

```
{  
  "contributors": [  
    {  
      "name": "Joe",  
      "email": "joe@whatever.com",  
      "url": "https://whatever.com"  
    }  
  ]  
}
```

bugs

Link para o rastreador de issues do pacote, geralmente uma página de issues do GitHub

Exemplo:

```
{  
  "bugs": "https://github.com/whatever/package/issues"  
}
```

homepage

Define a página inicial do site do pacote

Exemplo:

```
{  
  "homepage": "https://whatever.com/package"  
}
```

version

Indica a versão atual do pacote.

Exemplo:

```
"version": "1.0.0"
```

Essa propriedade segue o padrão de notação de versionamento semântico chamado *semver*, o que significa que é sempre expresso com 3 números: `x.x.x`.

O primeiro número é sempre a versão major, o segundo a versão minor e o terceiro a versão patch.

Há um significado nesses números: uma versão que só corrige bugs é uma versão patch, uma versão que introduz mudanças compatíveis com versões anteriores é uma versão minor, uma versão major pode conter quebra de comportamentos com versões antigas.

license

Indica a licença do pacote.

Exemplo:

```
"license": "MIT"
```

keywords

Esse propriedade contém um array de palavras chave que são associadas a o quê esse projeto faz.

Exemplo:

```
"keywords": [  
    "email",  
    "machine learning",  
    "ai"  
]
```

Isso ajuda pessoas a encontrarem seu pacote quando estiverem navegando por pacotes semelhantes, ou quando estiverem navegando no site <https://www.npmjs.com/>.

description

Essa propriedade contém uma breve descrição do pacote

Exemplo:

```
"description": "Um pacote para trabalhar com strings"
```

Isso é útil especialmente se você decide publicar seu pacote no npm, assim as pessoas podem encontrar sobre o quê se trata aquele pacote.

repository

Essa propriedade especifica onde o repositório desse pacote está localizado.

Exemplo:

```
"repository": "github:whatever/testing",
```

Note o prefixo github. Também há outros serviços populares:

```
"repository": "gitlab:whatever/testing",
```

```
"repository": "bitbucket:whatever/testing",
```

Você pode explicitar o sistema de controle de versionamento:

```
"repository": {  
  "type": "git",  
  "url": "https://github.com/whatever/testing.git"  
}
```

Você pode usar diferentes tipos de controle de versionamento:

```
"repository": {  
  "type": "svn",  
  "url": "..."  
}
```

main

Define o ponto de entrada do pacote.

Quando você importa esse pacote em uma aplicação, é onde a aplicação vai procurar pelo *module exports*.

Exemplo:

```
"main": "src/main.js"
```

private

Se definido como `true`, previne do pacote ser publicado acidentalmente no `npm`

Exemplo:

```
"private": true
```

scripts

Define um conjunto de scripts node que você pode executar

Exemplo:

```
"scripts": {  
  "dev": "webpack-dev-server --inline --progress --config build/webpack.config.js",  
  "start": "npm run dev",  
  "unit": "jest --config test/unit/jest.conf.js --coverage",  
  "test": "npm run unit",  
  "lint": "eslint --ext .js,.vue src test/unit",  
  "build": "node build/build.js"  
}
```

Esses scripts são aplicações de linha de comando. Você pode rodá-los usando `npm run XXXX` ou `yarn XXXX`, sendo `XXXX` o nome do comando. Exemplo: `npm run dev`.

Você pode usar o nome que quiser para o comando, e os scripts podem fazer qualquer coisa.

dependencies

Define uma lista de pacotes `npm` instalados como dependências.

Quando você instala um pacote usando npm ou yarn:

```
npm install <PACKAGENAME>
yarn add <PACKAGENAME>
```

esse pacote é inserido automaticamente nessa lista.

Exemplo:

```
"dependencies": {
  "vue": "^2.5.2"
}
```

devDependencies

Define uma lista de pacotes npm instalados como dependências de desenvolvimento.

Eles se diferem dos pacotes no dependencies porque eles são instalados apenas em ambiente de desenvolvimento, não são necessários para rodar o código em produção.

Quando você instala um pacote usando npm ou yarn:

```
npm install --save-dev <PACKAGENAME>
yarn add --dev <PACKAGENAME>
```

esse pacote é inserido automaticamente nessa lista.

Exemplo:

```
"devDependencies": {  
  "autoprefixer": "^7.1.2",  
  "babel-core": "^6.22.1"  
}
```

engines

Define em quais versões do Node.js e outros comandos esse pacote opera

Exemplo:

```
"engines": {  
  "node": ">= 6.0.0",  
  "npm": ">= 3.0.0",  
  "yarn": "^0.13.0"  
}
```

browserslist

É usado para definir quais browsers (e suas versões) você deseja dar suporte. É refenciado pelo Babel, Autoprefixer, e outras ferramentas, apenas para adicionar polyfills e fallbacks necessárias para o browser em questão.

Exemplo:

```
"browserslist": [  
  "> 1%",  
  "last 2 versions",  
  "not ie <= 8"  
]
```

Essa configuração implica que você quer dar suporte para as 2 últimas versões major de todos os browsers com pelo menos 1% de uso (das estatísticas do [CanIUse.com](#)), exceto IE8 ou anterior.

([veja mais sobre](#))

Propriedades específicas de comandos

O arquivo `package.json` também pode conter configurações específicas de comandos, como por exemplo Babel, ESLint, e mais.

Cada um tem propriedades específicas, como `eslintConfig`, `babel` e outras. Você pode encontrar como utilizá-las em suas respectivas documentações.

Versões dos pacotes

Você viu nos exemplos acima números como esse: `~3.0.0` ou `^0.13.0`. O que eles significam, e quais outros especificadores de versão você pode usar?

O símbolo especifica quais atualizações seu pacote aceita, para aquela dependência.

Dado que ao usar semver (versionamento semântico) todas versões têm 3 dígitos, o primeiro sendo a versão major, o segundo a versão minor e o terceiro a versão patch, você tem essas [regras](#).

Você pode combinar a maioria das versões em intervalos, como esse: `1.0.0 || >=1.1.0 <1.2.0`, para usar ou 1.0.0 ou uma versão maior ou igual que 1.1.0, mas menor que 1.2.0.

O arquivo `package-lock.json`

Na versão 5, o npm introduziu o arquivo `package-lock.json`.

O que é isso? Você provavelmente conhece sobre o arquivo `package.json`, que é muito mais comum e está por aí a mais tempo.

O objeto do arquivo é manter rastreada a versão exata de cada pacote que está instalado, assim o produto é 100% reproduzível mesmo que pacotes sejam atualizados pelos seus mantenedores.

Isso resolve um problema muito específico que o `package.json` deixou para trás. No `package.json` você pode definir para quais versões você quer atualizar (patch ou minor), usando a notação `semver`, por exemplo:

- se você especifica `~0.13.0`, você quer atualizar apenas para versões patch: `0.13.1` está ok, mas `0.14.0` não está.
- se você especifica `^0.13.0`, você quer atualizar tanto para versões patch quanto minor: `0.13.1`, `0.14.0` e assim por diante.
- se você especifica `0.13.0`, essa é a versão exata que será utilizada, sempre

Você não commita a pasta `node_modules` no seu Git, que geralmente é gigantesca, e quando você tenta replicar o projeto em outra máquina utilizando o comando `npm install`, se você especificou com a sintaxe `~` e uma versão patch do pacote foi lançada, ela que será instalada. O mesmo para `^` e versões inferiores.

Se você especifica versões exatas, como a `0.13.0` do exemplo, você não será afetado por esse problema.

Pode ser com você, ou com outra pessoa do outro lado do mundo tentando inicializar o projeto rodando `npm install`.

Então seu projeto original e o novo projeto recém inicializado são na verdade diferentes. Mesmo que uma versão patch ou minor não introduza mudanças

que quebrem códigos antigos, nós todos sabemos que bugs podem (e vão) surgir.

O `package-lock.json` grava na pedra a exata versão instalada de cada pacote, e o `npm` vai utilizar essas exatas versões quando rodar `npm install`.

Esse conceito não é novo, e gerenciadores de pacotes de outras linguagens (como o Composer no PHP) utilizam desse sistema por anos.

O arquivo `package-lock.json` precisa ser commitado no seu repositório Git, assim ele pode ser encontrado por outras pessoas, se o projeto é público ou você tem colaboradores, ou se você utiliza o GIT como fonte para deploys.

As versões das dependências vão ser atualizadas no arquivo `package-lock.json` quando você rodar `npm update`.

Um exemplo

Essa é uma estrutura de exemplo de um arquivo `package-lock.json` que é obtida quando rodamos `npm install cowsay` em uma pasta vazia:

```
{
  "requires": true,
  "lockfileVersion": 1,
  "dependencies": {
    "ansi-regex": {
      "version": "3.0.0",
      "resolved": "https://registry.npmjs.org/ansi-regex/-/ansi-regex-3.0.0.tgz",
      "integrity": "sha1-7QMXwyIGT31GbAKWa922Bas32Zg="
    },
    "cowsay": {
      "version": "1.3.1",
      "resolved": "https://registry.npmjs.org/cowsay/-/cowsay-1.3.1.tgz"
    }
  }
}
```

```
"integrity": "sha512-3PVFe6FePVtPj1HTeLin9v8WyLl+VmM1l1H/5P+BT' Ajufp+0F9eLjzRn0HzVAYeIYFF5po5NjRrgefnRMQ==",
  "requires": {
    "get-stdin": "^5.0.1",
    "optimist": "~0.6.1",
    "string-width": "~2.1.1",
    "strip-eof": "^1.0.0"
  }
},
"get-stdin": {
  "version": "5.0.1",
  "resolved": "https://registry.npmjs.org/get-stdin/-/get-stdin-1.1.1.tgz",
  "integrity": "sha1-Ei4WFZHiH/TFJTAwVpPyDmOTo5g="
},
"is-fullwidth-code-point": {
  "version": "2.0.0",
  "resolved": "https://registry.npmjs.org/is-fullwidth-code-point/-/is-fullwidth-code-point-2.0.0.tgz",
  "integrity": "sha1-o7MKXE8ZkYMWeqq50+764937ZU8="
},
"minimist": {
  "version": "0.0.10",
  "resolved": "https://registry.npmjs.org/minimist/-/minimist-0.0.10.tgz",
  "integrity": "sha1-3j+YVD2/lggr5IrRoMfNqDYwHc8="
},
"optimist": {
  "version": "0.6.1",
  "resolved": "https://registry.npmjs.org/optimist/-/optimist-0.6.1.tgz",
  "integrity": "sha1-2j6nRob6IaGaERwybpDrFaAZzoY=",

  "requires": {
    "minimist": "~0.0.1",
    "wordwrap": "~0.0.2"
  }
},
"string-width": {
  "version": "2.1.1",
  "resolved": "https://registry.npmjs.org/string-width/-/string-width-2.1.1.tgz",
  "integrity": "sha1-3j+YVD2/lggr5IrRoMfNqDYwHc8="
}
```

```
"integrity": "sha512-n0qH59deCq9SRHlxq1Aw85Jnt4w6KvLKqWVik6oA9",
  "requires": {
    "is-fullwidth-code-point": "^2.0.0",
    "strip-ansi": "^4.0.0"
  },
  "strip-ansi": {
    "version": "4.0.0",
    "resolved": "https://registry.npmjs.org/strip-ansi/-/strip-ansi-4.0.0.tgz",
    "integrity": "sha1-qEeQIusaw2iocTibY1JixQXuNo8=",
    "requires": {
      "ansi-regex": "^3.0.0"
    }
  },
  "strip-eof": {
    "version": "1.0.0",
    "resolved": "https://registry.npmjs.org/strip-eof/-/strip-eof-1.0.0.tgz",
    "integrity": "sha1-u0P/VZim6wXYm1n80SnJgzE2Br8="
  },
  "wordwrap": {
    "version": "0.0.3",
    "resolved": "https://registry.npmjs.org/wordwrap/-/wordwrap-0.0.3.tgz",
    "integrity": "sha1-o9XabNXAvAAI03I0u68b7WMFkQc="
  }
}
```

Nós instalamos o `cowsay`, que depende de:

- `get-stdin`
- `optimist`
- `string-width`
- `strip-eof`

Por sua vez, esses pacotes dependem de outros pacotes, como podemos ver nos `requires` de alguns deles:

- `ansi-regex`
- `is-fullwidth-code-point`
- `minimist`
- `wordwrap`
- `strip-eof`

Eles são adicionados ao arquivo em ordem alfabética, e cada um têm um campo `version`, um `resolver`, que aponta para a localização do pacote, e um `integrity`, que é uma string para validação do pacote.

Veja as versões instaladas de um pacote NPM

Para ver a última versão de todos os pacotes npm instalados, incluindo suas dependências:

```
npm list
```

Por exemplo:

```
> npm list
/Users/joe/dev/node/cowsay
└── cowsay@1.3.1
    ├── get-stdin@5.0.1
    ├── optimist@0.6.1
    │   └── minimist@0.0.10
    ├── wordwrap@0.0.3
    ├── string-width@2.1.1
    │   ├── is-fullwidth-code-point@2.0.0
    │   └── strip-ansi@4.0.0
    └── ansi-regex@3.0.0
        └── strip-eof@1.0.0
```

Você também pode abrir o arquivo `package-lock.json`, mas você terá que ler muita informação desnecessária.

`npm list -g` tem o mesmo funcionamento, porém para pacotes instalados globalmente.

Para obter apenas os pacotes top-level (basicamente aqueles que você pediu para o npm instalar e estão listados no `package.json`), execute `npm list --depth=0`:

```
> npm list --depth=0
/Users/joe/dev/node/cowsay
└── cowsay@1.3.1
```

Você pode obter a versão de um pacote em específico fornecendo o nome dele:

```
> npm list cowsay
/Users/joe/dev/node/cowsay
└── cowsay@1.3.1
```

Isso também funciona para dependências dos pacotes instalados:

```
> npm list minimist
/Users/joe/dev/node/cowsay
└── cowsay@1.3.1
    └── optimist@0.6.1
        └── minimist@0.0.10
```

Se você quer ver qual é a última versão disponível de um pacote no repositório npm, execute `npm view [package_name] version`:

```
> npm view cowsay version
1.3.1
```

Instale uma versão mais antiga de um pacote NPM

Você pode instalar uma versão antiga de um pacote npm usando a sintaxe @:

```
npm install <package>@<version>
```

Exemplo:

```
npm install cowsay
```

instala a versão 1.3.1 (no momento de escrita).

Instale a versão 1.2.0 assim:

```
npm install cowsay@1.2.0
```

O mesmo pode ser feito com pacotes globais:

```
npm install -g webpack@4.16.4
```

Você também pode estar interessado em listar todas as versões anteriores de um pacote. Você pode fazer isso executando `npm view <package> versions`:

```
> npm view cowsay versions
```

```
[ '1.0.0',
  '1.0.1',
  '1.0.2',
  '1.0.3',
  '1.1.0',
  '1.1.1',
  '1.1.2',
  '1.1.3',
```

```
'1.1.4',
'1.1.5',
'1.1.6',
'1.1.7',
'1.1.8',
'1.1.9',
'1.2.0',
'1.2.1',
'1.3.0',
'1.3.1' ]
```

Atualize todas as versões do Node.js para a última versão.

Quando você instala um pacote utilizando `npm install <packagename>`, a última versão disponível do pacote é baixada e colocada na pasta `node_modules`, e uma entrada correspondente é adicionada nos arquivos `package.json` e `package-lock.json` presentes na pasta atual.

o npm calcula as dependências e também instala a última versão delas.

Digamos que você instale o `cowsay`, uma ferramenta de linha de comando bem legal que te permite fazer uma vaca falar *coisas*.

Quando você roda `npm install cowsay`, essa entrada é adicionada no arquivo `package.json`:

```
{
  "dependencies": {
    "cowsay": "^1.3.1"
  }
}
```

e esse é um trecho do `package-lock.json`, onde removemos as subdependências para facilitar a leitura:

```
{  
  "requires": true,  
  "lockfileVersion": 1,  
  "dependencies": {  
    "cowsay": {  
      "version": "1.3.1",  
      "resolved": "https://registry.npmjs.org/cowsay/-/cowsay-1.3.1..tgz  
      "integrity": "sha512-3PVFe6FePVtPj1HTeLin9v8WyLl+VmM1l1H/5P+BT  
      "requires": {  
        "get-stdin": "^5.0.1",  
        "optimist": "~0.6.1",  
        "string-width": "~2.1.1",  
        "strip-eof": "^1.0.0"  
      }  
    }  
  }  
}
```

Agora esses 2 arquivos nos dizem que instalamos a versão **1.3.1** do cowsay, e nossa regra para atualizações é **^1.3.1**, o que o npm pode atualizar para versões patch e minor: **1.3.2**, **1.4.0** e assim por diante.

Se há uma nova versão minor ou patch e nós digitamos **npm update**, a versão instalada é atualizada, e o arquivo **package-lock.json** rapidamente se atualiza com a nova versão.

o **package.json** permanece sem alterações.

Para descobrir novas atualizações dos pacotes, rode **npm outdated**.

Aqui está uma lista de alguns pacotes desatualizados em um repositório que não recebe atualizações há um bom tempo:

Algumas dessas atualizações são para versões major. Executando `npm update` elas não serão atualizadas. Versões major nunca são atualizadas desse jeito porque elas (por definição) introduzem mudanças sujeitas a quebras em versões antigas, e o `npm` quer nos poupar desse problema.

Para atualizar para uma versão major desses pacotes, instale o pacote `npm-check-updates` globalmente:

```
npm install -g npm-check-updates
```

e então rode:

```
ncu -u
```

isso irá atualizar as versões no `package.json`, tanto `dependencies` quanto `devDependencies`, assim o `npm` poderá instalar as versões major.

Agora você está pronto para executar as atualizações:

```
npm update
```

Se você só baixou o projeto sem as dependências do `node_modules` e quer instalar as novas versões, basta rodar

`npm install`

Usando Semantic Versioning no NPM

Se há uma coisa incrível nos pacotes Node.js, é que todos eles permitem utilizar Versionamento Semântico em suas numerações de versão.

O Versionamento Semântico é um conceito simples: todas versões têm 3 dígitos: `x.y.z`.

- o primeiro dígito é a versão major
- o segundo dígito é a versão minor
- o terceiro dígito é a versão patch

Quando você cria uma nova versão, você não aumenta um número como bem entender, existem algumas regras:

- você aumenta a versão major quando você cria mudanças de API incompatíveis
- você aumenta a versão minor quando você adiciona uma funcionalidade compatível com versões anteriores
- você aumenta a versão patch quando você arruma bugs sem quebrar versões anteriores

Essa convenção é adotada em todas linguagens de programação, e é muito importante que todo pacote `npm` a siga, pois o sistema como um todo depende disso.

Por que isso é tão importante?

Porque o `npm` define algumas regras que podemos usar no `package.json` para escolher quais versões podem ser atualizadas ao rodar `npm update`.

As regras utilizam esses símbolos:

- `^`
- `~`
- `>`
- `>=`
- `<`
- `<=`
- `=`
- `-`
- `||`

Vamos ver essas regras em detalhes:

- `^`: Só vai fazer atualizações que não alterem o número mais a esquerda diferente de zero. Se você definir `^0.13.0`, ao rodar `npm update`, ele pode atualizar para `0.13.1`, `0.13.2`, e assim por diante, mas não para `0.14.0` ou posteriores. Se você definir `^1.13.0`, ao rodar `npm update`, ele pode atualizar para `1.13.1`, `1.14.0` e assim por diante, mas não para `2.0.0` ou posterior.
- `~`: se você definir `~0.13.0`, ao rodar `npm update` ele pode atualizar para versões patch: `0.13.1` está ok, mas `0.14.0` não está.
- `>`: aceita qualquer versão maior que a especificada
- `>=`: aceita qualquer versão maior ou igual a especificada
- `<=`: aceita qualquer versão menor ou igual a especificada

- <: aceita qualquer versão menor que a especificada
- ==: aceita aquela exata versão
- -: aceita qualquer versão dentro de um intervalo. Exemplo: 2.1.0 - 2.6.2
- ||: combina restrições. Exemplo: < 2.1 || > 2.6

Você pode combinar algumas dessas notações, por exemplo, utilize 1.0.0 || >=1.1.0 <1.2.0 para usar 1.0.0 ou uma versão maior ou igual a 1.1.0, porém menor que 1.2.0.

Também há outras regras:

- sem símbolo: aceita apenas a versão especificada (1.2.1)
- latest: usa a versão mais recente disponível

Desinstalando pacotes NPM

Para desinstalar um pacote que você tenha instalado previamente de forma local (usando `npm install <package-name>` na pasta `node_modules`), execute

```
npm uninstall <package-name>
```

a partir da pasta raiz do projeto (a pasta que contém a pasta `node_modules`).

Usando a flag `-S`, ou `--save`, essa operação também irá remover a referência no arquivo `package.json`.

Se o pacote era uma dependência de desenvolvimento, listada no `devDependencies` do arquivo `package.json`, você deve utilizar a flag `-D` / `--save-dev` para removê-la do arquivo.

```
npm uninstall -S <package-name>
npm uninstall -D <package-name>
```

Se o pacote está instalado **globalmente**, você precisa adicionar a flag `-g / --global`:

```
npm uninstall -g <package-name>
```

por exemplo:

```
npm uninstall -g webpack
```

e você pode rodar esse comando em qualquer lugar do seu sistema pois a pasta que você está não importa. and you can run this command from anywhere you want on your system because the folder where you currently are does not matter.

NPM: pacotes globais ou locais

A principal diferença entre pacotes locais e globais é:

- **pacotes locais** são instalados no diretório em que você executa `npm install <package-name>`, e eles são inseridos na pasta `node_modules` desse diretório.
- **pacotes globais** são todos colocados em único lugar do seu sistema (dependendo da sua configuração), independentemente de onde você execute `npm install -g <package-name>`

No seu código você só pode importar pacotes locais:

```
require('package-name')
```

então quando instalar um ou outro?

No geral, todos os pacotes devem ser instalados localmente.

Isso garante que você possa ter dúzias de aplicações no seu computador, todas rodando com uma versão diferente de cada pacote se necessário.

Atualizar um pacote global fará com que todos seus projetos usem a nova versão, e como você pode imaginar isso pode causar pesadelos em termos de manutenção, pois alguns pacotes podem quebrar.

Todos projetos tem sua própria versão local de um pacote, mesmo que isso possa parecer um desperdício de recursos, é algo pequeno comparado as possíveis consequências.

Um pacote deve ser instalado globalmente quando ele fornece um comando executável que possa ser rodado pelo terminal (CLI), e é reutilizado entre projetos.

Você também pode instalar comandos executáveis localmente e rodá-los usando o npx, porém alguns pacotes funcionam melhor instalados de forma global.

Alguns otimos exemplos de pacotes globais populares que você pode conhecer:

- npm
- create-react-app
- vue-cli
- grunt-cli
- mocha
- react-native-cli
- gatsby-cli
- forever

- nodemon

Você provavelmente já tem alguns deles instalados globalmente no seu sistema. Você pode conferir executando

```
npm list -g --depth 0
```

no seu terminal.

npm: diferença entre dependencies e devDependencies

Quando você instala um pacote usando `npm install <package-name>`, ele está sendo instalado como uma **dependência**.

O pacote é listado automaticamente no arquivo `package.json`, na lista de **dependencies** (a partir do npm 5: antes você tinha que especificar manualmente com `--save`).

Quando você adiciona a flag `-D`, ou `--save-dev`, você está instalado como uma dependência de desenvolvimento, que é adicionada na lista de **devDependencies**.

Dependências de desenvolvimento são entendidas como pacotes apenas para desenvolver, não são necessárias em produção. Por exemplo pacotes de teste, webpack ou Babel.

Quando você vai para produção, se digitar `npm install` e a pasta conter um arquivo `package.json`, elas são instaladas, pois o npm assume que é um deploy de desenvolvimento.

Você precisa passar a flag `--production` (`npm install --production`) para evitar instalar essas dependências de desenvolvimento.

Conhecendo o NPX

`npx` é um comando poderoso que está disponível no `npm` a partir da versão 5.2, lançada em Julho de 2017.

Se você não quer instalar o npm, você pode [instalar o npx como um pacote standalone](#)

O `npx` permite rodar seu código feito com Node.js e publicado nos registros do `npm`.

Rode comandos locais facilmente

Desenvolvedores Node.js costumam publicar a maioria dos comandos executáveis como pacotes globais, pois assim é garantido que eles estejam salvos no `path` e executados imediatamente.

Isso é bem chato porque você não pode instalar versões diferentes do mesmo comando.

Ao rodar `npx nomedocomando` a referência correta do comando dentro da pasta `node_modules` do projeto é encontrada automaticamente, e sem a necessidade de saber o caminho exato, e sem a necessidade de ter o pacote instalado globalmente e no `path` do usuário.

Execução de comandos sem instalação

Outro excelente recurso do `npx` é permitir rodar comandos sem tê-los instalados.

Isso é extremamente útil, porque:

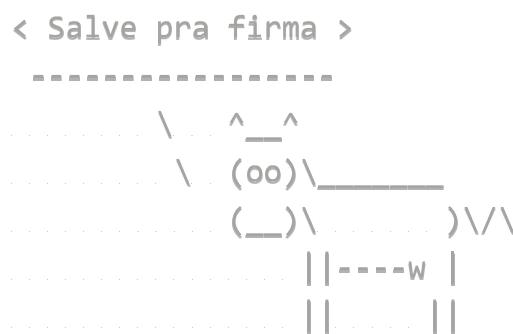
1. você não precisa instalar nada

2. você pode rodar diferentes versões do mesmo comando, usando a sintaxe

`@version`

Uma demonstração típica do uso do `npx` é através do comando `cowsay`. O `cowsay` vai imprimir uma vaca na tela dizendo o que você digitou no comando. Por exemplo:

`cowsay "Salve pra firma"` vai resultar em



Isso só funciona se você tiver o comando `cowsay` previamente instalado pelo npm. Caso contrário você receberá um erro ao tentar rodar o comando.

O `npx` permite que você rode o comando sem tê-lo instalado localmente:

`npx cowsay "Salve pra firma"`

Outros cenários de exemplo:

- rodar a CLI do `vue` para criar novas aplicações: `npx @vue/cli create my-vue-app`
- criar uma aplicação React usando `create-react-app`: `npx create-react-app my-react-app`

Uma vez executado, o código baixado será apagado.

Rodando um código em uma versão diferente do Node.js

Utilize o `@` para especificar a versão, e combine-o com uma [versão empacotada do node](#):

```
npx node@10 -v #v10.18.1  
npx node@12 -v #v12.14.1
```

Isso ajuda a evitar ferramentas como o `nvm` ou outros gerenciadores de versão para Node.js.

Rodando códigos diretamente de uma URL

O `npx` não te limita aos pacotes publicados nos registros do npm.

Você pode rodar códigos de um gist no GitHub, por exemplo:

```
npx https://gist.github.com/zkat/4bc19503fe9e9309e2bfaa2c58074d32
```

Mas é claro que você precisa ter cuidado ao rodar códigos que estão fora do seu controle, com grandes poderes vem grandes responsabilidades.

O Event Loop do Node.js

Introdução

O Event Loop é um dos aspectos do Node.js mais importantes de se entender.

Por que ele é tão importante? Porque ele explica como o Node.js pode ser assíncrono e ter I/O não bloqueante, e também explica o diferencial do Node.js,

o quê fez dele bem-sucedido.

O código JavaScript Node.js roda em uma única thread. Há apenas uma coisa acontecendo por vez.

Essa é uma limitação que na verdade é bem útil, pois ela simplifica muito como você programa sem se preocupar com problemas de concorrência.

Você só precisa prestar atenção em como escrever seu código e evitar qualquer coisa que possa bloquear a thread, como chamadas de rede síncronas ou loops infinitos.

No geral, na maioria dos browsers existe um event loop para cada aba aberta, para tratar cada processo de forma isolada e evitar que uma página web com loops infinitos ou processamento pesado bloqueie o browser por completo.

O ambiente de execução gerencia múltiplos event loops concorrentes, para fazer chamadas à APIs por exemplo. Web Workers rodam em seus próprios event loops também.

Você precisa se preocupar principalmente com que seu código rode um único event loop, e escrever o código com isso em mente a fim de evitar bloquear a thread.

Bloqueando o event loop

Qualquer código JavaScript que demore muito para retornar ao controle do event loop irá bloquear a execução de algum código JavaScript na página, até mesmo bloquear a thread de UI, impossibilitando o usuário de clicar, rolar, e etc.

Quase todas primitivas de I/O no JavaScript são não bloqueantes. Requisições de rede, operações no filesystem, e etc. Ser bloqueante é a exceção, e esse é o

porquê do JavaScript ser extremamente baseado em callbacks, e mais recentemente em promises e async/await.

A call stack

A call stack (pilha de chamadas) é uma fila do tipo LIFO (Last In, First Out, que em português significa último a entrar, primeiro a sair).

O event loop checa continuamente a **call stack** para ver se há qualquer função que precise rodar.

Enquanto faz isso, ele adiciona na call stack qualquer chamada de função que encontrar e executa cada uma em ordem.

Sabe quando ocorre um erro e no debugger ou no console do browser aparece uma sequência hierárquica de nomes de funções? Basicamente o browser procura os nomes das funções na call stack para te informar qual função originou a chamada corrente:

```
> const bar = () => {
    throw new DOMException()
}

const baz = () => console.log('baz')

const foo = () => {
  console.log('foo')
  bar()
  baz()
}

foo()

foo
✖ ▼ Uncaught DOMException
```

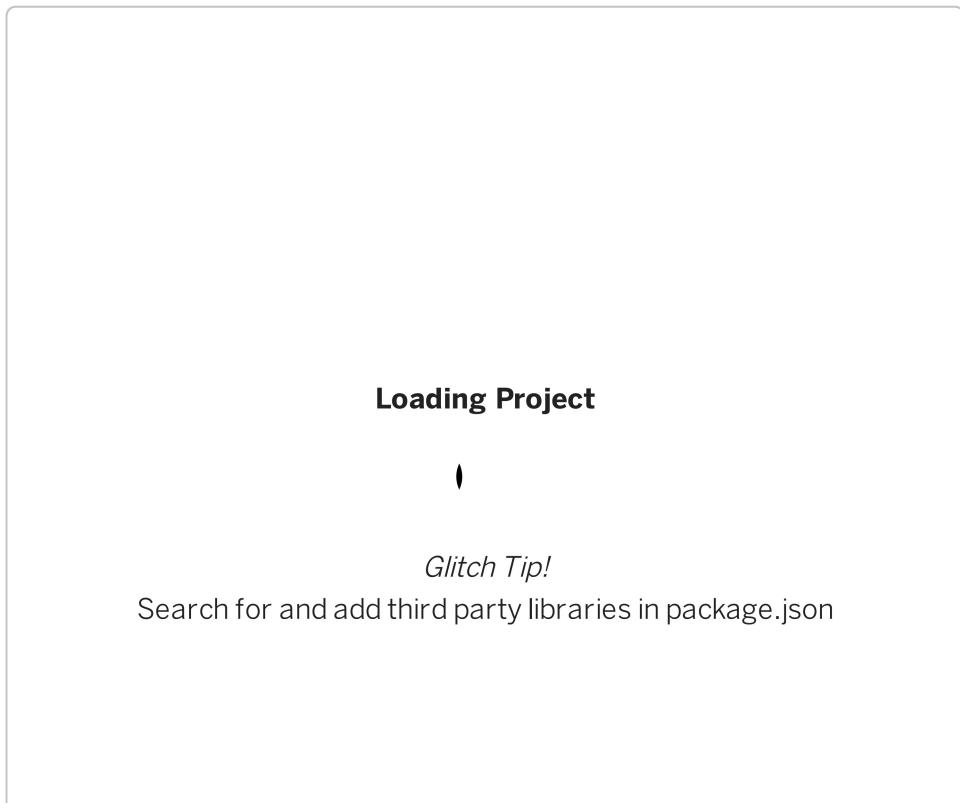
```
bar      @ VM570:2
foo      @ VM570:9
(anonymous) @ VM570:13
```



|

Uma explicação simples sobre o event loop

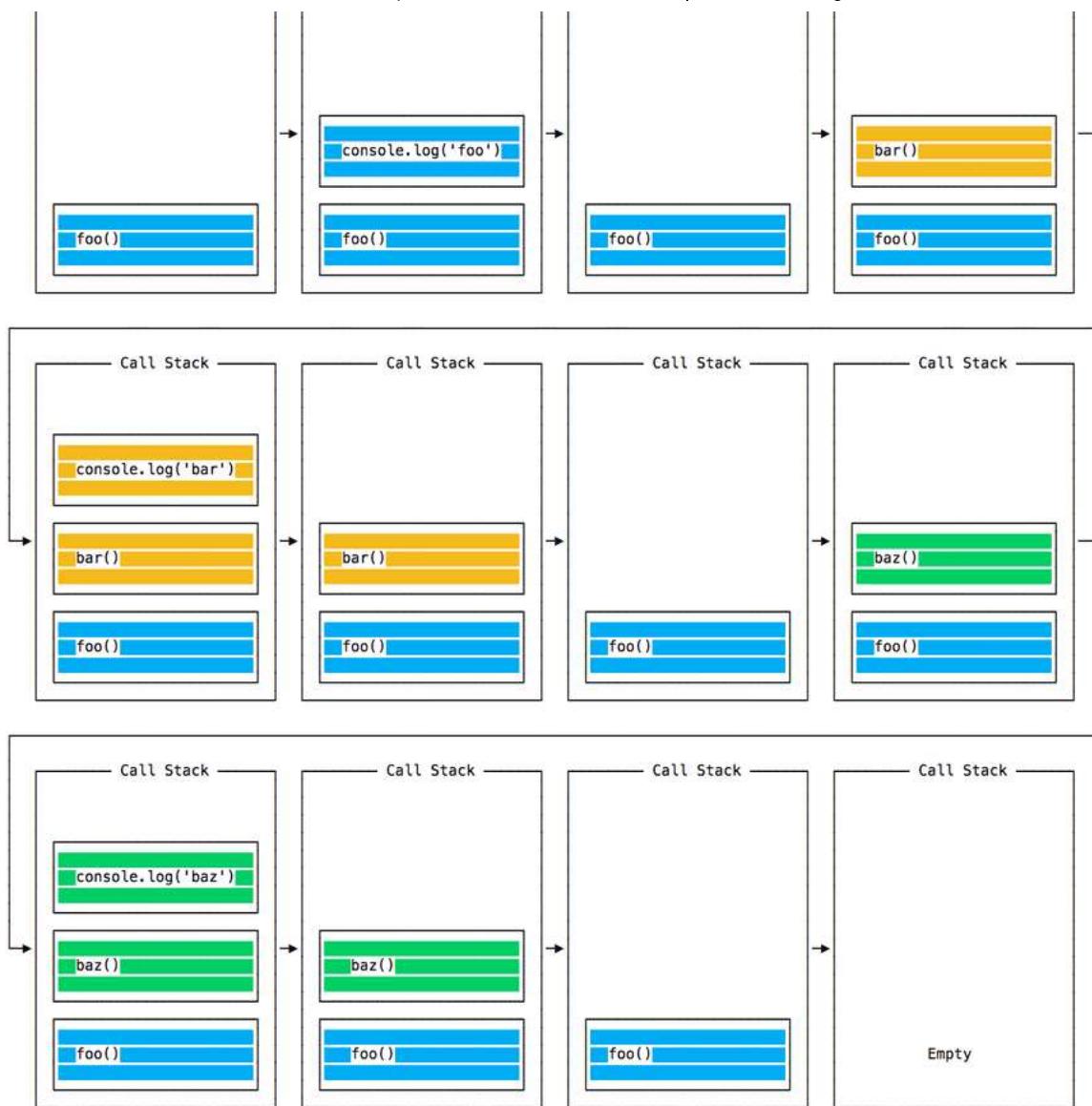
Dado esse exemplo:



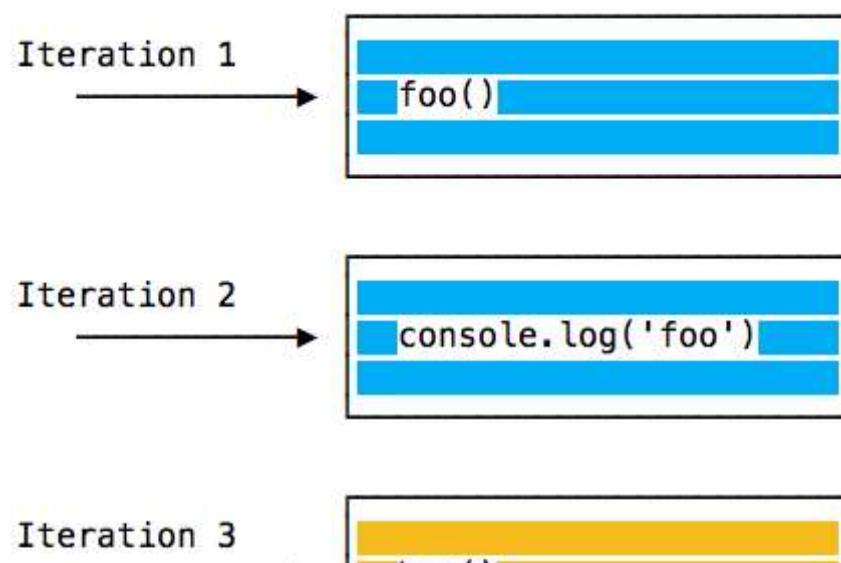
Quando esse código roda, primeiro é chamada `foo()`. Dentro de `foo()` nós chamamos `bar()`, e então nós chamamos `baz()`.

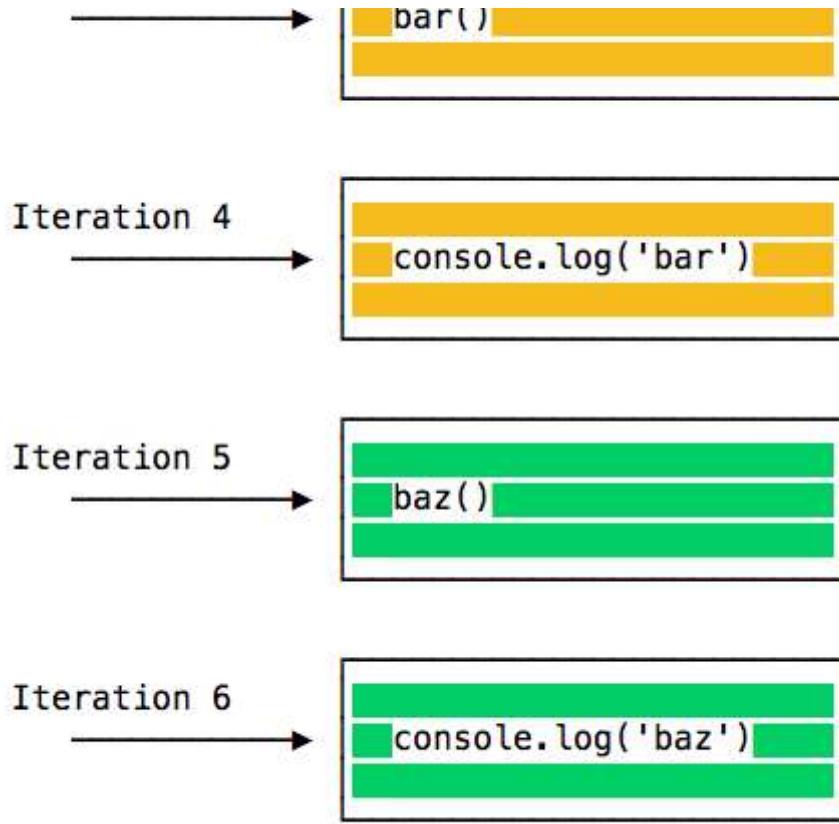
Nesse ponto a call stack se parece com isso:





O event loop olha em toda iteração se há algo na call stack, e o executa:





até que a call stack esteja vazia.

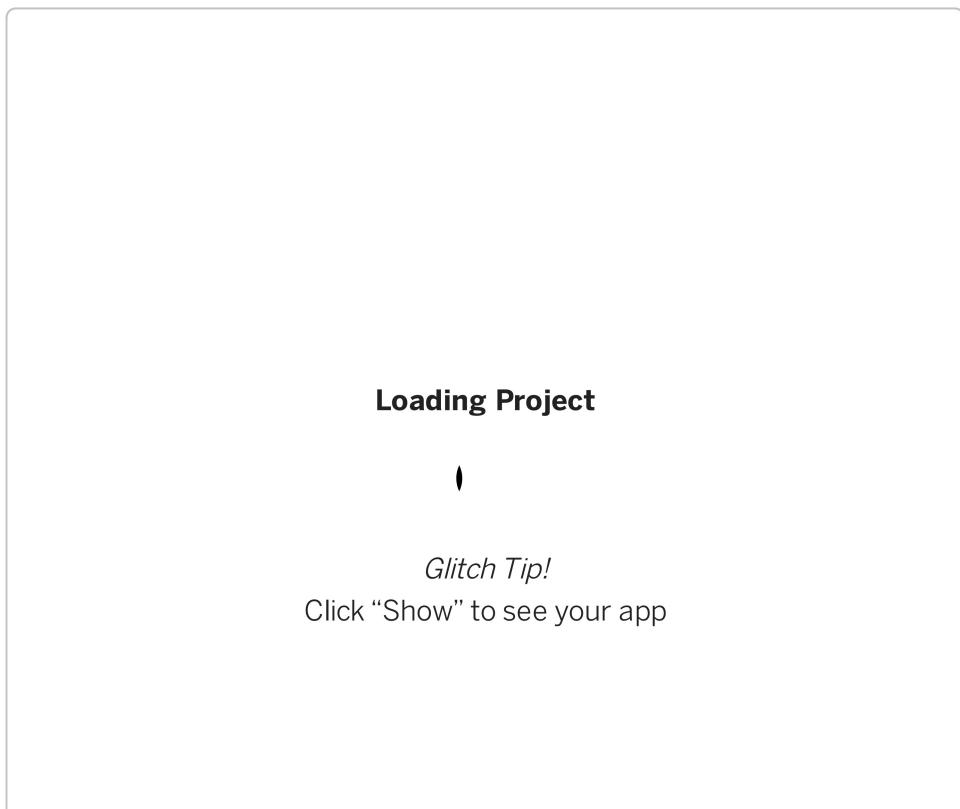
Enfileirando a execução de funções

O exemplo acima parece normal, não há nada de especial sobre ele: o JavaScript encontra coisas para executar, e as executa em ordem.

Vamos ver como adiar uma função até que a stack esteja vazia.

O caso de uso do `setTimeout(() => {}, 0)` é chamar uma função, mas só executá-la assim que todas outras funções no código tenham executado.

Veja esse exemplo:



Para nossa surpresa, esse código imprime:

```
foo  
baz  
bar
```

Quando esse código roda, primeiro `foo()` é chamada. Dentro de `foo()` nós chamamos `setTimeout`, passando `bar` como um argumento, e nós instruimos a rodá-la imediatamente, o mais rápido possível, passando `0` como o tempo de contagem. E então nós chamamos `baz()`.

Nesse ponto a call stack se parece com isso:

Aqui temos a ordem de execução de todas as funções no nosso programa:

Por que isso está acontecendo?

A Message Queue

Quando `setTimeout()` é chamado, o Browser ou o Node.js começa a contagem. Uma vez que ela encerre, nesse caso imediatamente pois definimos o como parâmetro, a função callback é colocada na **Message Queue** (fila de mensagens).

A Message Queue também é onde estão eventos iniciados pelo usuário, como clicks ou pressionamento de teclas, ou respostas de requisições enfileiradas. antes do seu código ter a oportunidade de reagir à elas. Ou também eventos do DOM, como `onLoad`.

O loop dá prioridade à call stack, primeiro ele processa tudo que encontrar na call stack, e só depois que não há nada lá, ele começa a pegar coisas na message queue.

Nós não temos que esperar por funções como `setTimeout`, `fetch` ou outras finalizarem, porque elas são providas pelo browser, elas vivem em suas

próprias threads. Por exemplo, se você definir o tempo do `setTimeout` como 2 segundos, você não precisa esperar por 2 segundos - a espera ocorre em outro lugar.

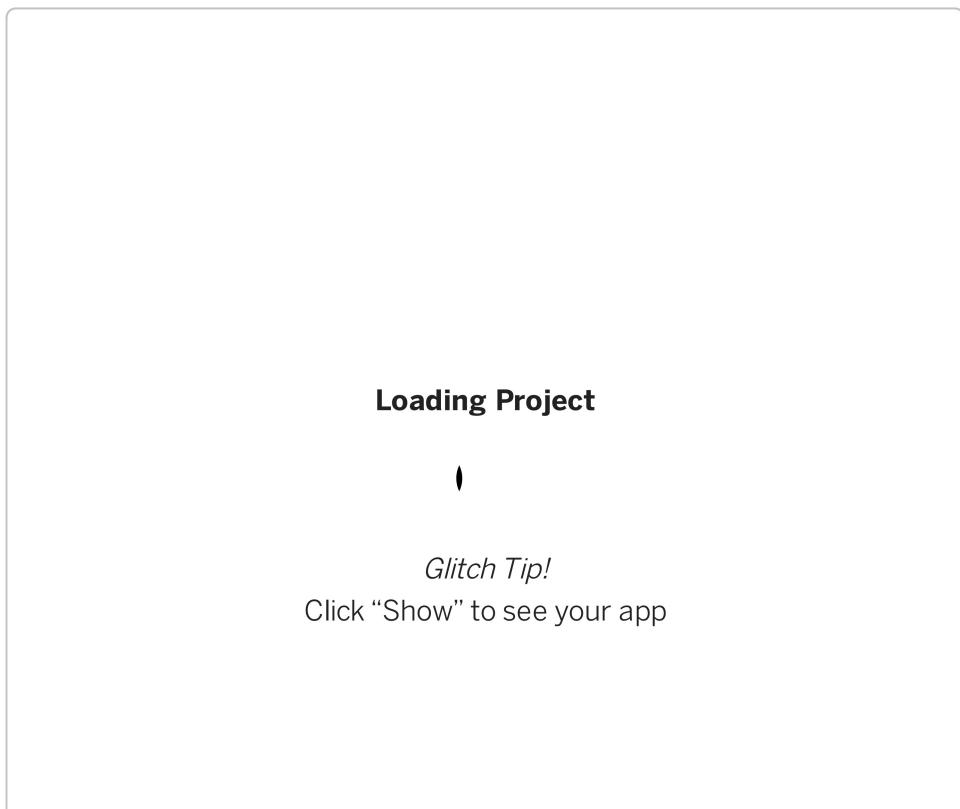
ES6 Job Queue

O ECMAScript 2015 introduziu o conceito de Job Queue (fila de trabalhos), que é usada por Promises (também introduzidas no ES6/ES2015). É uma maneira de executar o resultado de uma função assíncrona o mais rápido possível, invés de colocar no fim da call stack.

Promises que são resolvidas antes da função corrente encerrar serão executadas logo após a função corrente.

Eu acho legal a analogia de uma montanha russa em um parque de diversões: a message queue te coloca no final da fila, antes de todas outras pessoas, onde você terá que esperar pela sua vez, enquanto que a job queue é um ticket corta filas que deixa você ir de novo após a primeira ida.

Exemplo:



Há uma grande diferença entre Promises (e Async/await, que é feito com promises) e funções assíncronas antigas como `setTimeout()` ou outras APIs de plataforma.

Entendendo o `process.nextTick()`

Uma parte importante do event loop é o `process.nextTick()`.

Toda vez que o event loop termina um ciclo, nós chamamos isso de *tick*.

Quando nós passamos uma função pro `process.nextTick()`, nós instruimos a engine a invocar essa função no fim da operação atual, antes que o próximo tick do event loop tenha início.

```
process.nextTick(() => {
  //faça algo
})
```

O event loop está ocupando processando o código da função atual.

Quando a operação finaliza, a engine JS executa todas as funções passadas por chamadas do `nextTick` durante aquela operação.

Esse é o jeito que nós podemos dizer à engine JS para processar uma função assíncronamente (depois da função atual), mas o mais breve possível, sem enfileirar.

Chamando `setTimeout(() => {}, 0)` vai executar a função no fim do próximo tick, muito mais tarde do que quando usando `nextTick()`, que prioriza a chamada e a executa antes do próximo tick.

Use `nextTick()` quando você quer garantir que na próxima iteração do event loop o código já tenha sido executado.

Entendendo `setImmediate()`

Quando você quer executar um trecho de código assíncronamente, mas o mais rápido possível, uma opção é utilizar a função `setImmediate()` provida pelo Node.js:

```
setImmediate(() => {  
  //faça algo  
})
```

Qualquer função passada como argumento ao `setImmediate()` será executada na próxima iteração do event loop.

Qual a diferença entre `setImmediate()`, `setTimeout(() => {}, 0)` (passando 0ms como delay), e `process.nextTick()`?

A função passada para o `process.nextTick()` será executada na iteração atual do event loop, depois que a operação corrente finalizar. Isso significa que ela sempre executa antes do `setTimeout` e do `setImmediate`.

Uma callback `setTimeout()` com delay de 0ms é muito similar ao `setImmediate()`. A ordem de execução vai depender de vários fatores, mas ambas irão rodar na próxima iteração do event loop.

Timers do JavaScript

`setTimeout()`

Quando estiver escrevendo código JavaScript, você pode querer aplicar um delay na execução de uma função.

Esse é o trabalho da função `setTimeout`. Você especifica uma função de callback para ser executada mais tarde, e um valor em milisegundos especificando o quanto mais tarde ela deve rodar:

```
setTimeout(() => {
  // roda após 2 segundos
}, 2000)

setTimeout(() => {
  // roda após 50 milisegundos
}, 50)
```

Essa sintaxe define uma nova função. Você pode chamar qualquer outra função que quiser, ou passar o nome de uma função existente, e definir os parâmetros:

```
const myFunction = (firstParam, secondParam) => {
  // faz algo
```

```
}
```

```
// roda após 2 segundos
setTimeout(myFunction, 2000, firstParam, secondParam)
```

`setTimeout` retorna o código de identificação do timer. Geralmente esse código não é usado, mas você pode guardá-lo, e “limpar” o timer se você quiser desagendar a execução daquela função:

```
const id = setTimeout(() => {
  // deve rodar daqui 2 segundos
}, 2000)

// mudei de ideia :v
clearTimeout(id)
```

Delay de zero

Se você especificar o delay do timeout como `0`, a função de callback será executada assim que possível, mas só depois da execução das funções correntes:

```
setTimeout(() => {
  console.log('depois ')
}, 0)

console.log(' antes ')
```

imprime `antes` `depois`.

Isso é especialmente útil para evitar bloqueio da CPU em tarefas intensas e deixar outras funções serem executadas enquanto são feitos cálculos pesados, enfileirando funções no fluxo.

Alguns browsers (IE e Edge) implementam um método chamado `setImmediate()` que tem a exata mesma funcionalidade, mas não é padronizado e indisponível em outros navegadores. Porém é um método padrão no Node.js.

setInterval()

`setInterval` é uma função similar ao `setTimeout`, mas com uma diferença: em vez de rodar a função de callback uma vez, ela será rodada pra sempre, a cada intervalo de tempo especificado (em milisegundos):

```
setInterval(() => {
  // roda a cada 2 segundos
}, 2000)
```

A função abaixo roda a cada 2 segundos até que você a diga pra parar, usando `clearInterval`, passando o identificador retornado por aquele `setInterval`:

```
const id = setInterval(() => {
  // roda a cada 2 segundos
}, 2000)

 clearInterval(id)
```

É comum chamar o `clearInterval` dentro da função de callback do `setInterval`, para auto-determinar se ele deve rodar de novo ou parar. Por exemplo, esse código roda algo a não ser que o valor de `App.somethingIWait` seja `arrived`:

```
const interval = setInterval(() => {
  if (App.somethingIWait === 'arrived') {
    clearInterval(interval)
    return
  }
})
```

```
// caso contrário, executa o conteúdo restante
}, 100)
```

setTimeout recursivo

O `setInterval` inicia uma função a cada n milisegundos, sem levar em consideração quando uma função finalizou sua execução.

Se uma função gasta sempre o mesmo espaço de tempo, show de bola:

Talvez a função gaste tempos diferentes de execução, dependendo das condições da internet por exemplo:

E talvez uma longa execução sobreponha a próxima:

Para evitar isso, você pode agendar um `setTimeout` recursivo para ser chamado quando a função de callback finaliza:

```
const myFunction = () => {
  // faz algo
```

```
    .setTimeout(myFunction, 1000)
}

setTimeout(myFunction, 1000)
```

Pra chegar nesse cenário:

`setTimeout` e `setInterval` estão disponíveis no Node.js, através do [módulo Timers](#).

O Node.js também disponibiliza o `setImmediate()`, que é o equivalente de usar `setTimeout(() => {}, 0)`, normalmente utilizado para trabalhar com o Event Loop do Node.js.

Programação Assíncrona e Callbacks

Assincronicidade em Linguagens de Programação

Computadores são assíncronos por definição.

Assíncrono significa que as coisas podem acontecer independentemente do fluxo principal do programa.

Nos computadores de uso doméstico atuais, cada programa roda por um intervalo de tempo específico e, em seguida, interrompe sua execução para permitir que outro programa continue sua execução. Isso roda em um ciclo tão rápido que é impossível de notar. Nós pensamos que nossos computadores

rodam diversos programass simultaneamente, mas isso é uma ilusão (exceto em máquinas multiprocessadas).

Programas usam internamente *sinais de interrupção (interrupts)*, que são emitidos ao processador para ganharem a atenção do sistema.

Não irei muito a fundo no conceito, mas só tenha em mente que é normal para programas serem assíncronos e pararem a execução até que precisem de atenção, permitindo ao computador executar outras coisas nesse meio tempo. Quando um programa está esperando por uma resposta da rede, ele não pode parar o processador até que a requisição finalize.

Normalmente, linguagens de programação são síncronas e algumas fornecem maneiras de gerenciar assincronicidade na própria linguagem ou por meio de bibliotecas. C, Java, C#, PHP, Go, Ruby, Swift, e Python são todas síncronas por padrão. Algumas delas gerenciam assíncronismo usando threads, gerando um novo processo.

JavaScript

JavaScript é **síncrono** por padrão e roda em uma única thread. Isso significa que o código não pode criar novas threads e rodar em paralelo.

Linhas de código são executadas em séries, uma após a outra, por exemplo:

```
const a = 1
const b = 2
const c = a * b
console.log(c)
doSomething()
```

Mas o JavaScript nasceu dentro dos browsers, sua função principal, desde o começo, era responder às interações do usuário, como `onClick`, `onMouseOver`,

`onChange`, `onSubmit` e etc. Como isso pode ser feito com um modelo programático síncrono?

A resposta era o ambiente. O **browser** dá um jeito nisso fornecendo um conjunto de APIs que podem lidar com esses tipos de funcionalidades.

Mais recentemente, o Node.js introduziu um ambiente de I/O não bloqueante para estender esse conceito a acesso de arquivos, chamadas de rede e etc.

Callbacks

Não tem como você saber quando um usuário vai clicar em um botão. Então, você **define uma função para lidar com o evento de click**. Essa função será chamada quando o evento for acionado:

```
document.getElementById('button').addEventListener('click', () => {
  // item clicado
})
```

Isso também é chamado de **callback**.

Uma callback é uma simples função que é passada como valor para outra função, e só será executada quando o evento ocorrer. Nós podemos fazer isso por que o JavaScript têm funções de “primeira classe”, que podem ser atribuídas à variáveis e passadas para outras funções chamadas de **higher-order functions (funções de ordem superior)**.

É comum envolver todo código do cliente em um event listener do tipo `load` no objeto `window`, que roda só roda a função callback quando a página está pronta:

```
window.addEventListener('load', () => {
  // window carregou
```

```
// faça o que quiser
})
```

Callbacks são usadas em todo lugar, não só em eventos do DOM.

Um exemplo comum é utilizando temporizadores:

```
setTimeout(() => {
  // roda após 2 segundos
}, 2000)
```

Requisições XHR também aceitam uma callback, nesse exemplo ao passar uma função para uma propriedade que será chamada quando um evento em particular ocorrer (nesse caso, o estado da requisição mudar):

```
const xhr = new XMLHttpRequest()
xhr.onreadystatechange = () => {
  if (xhr.readyState === 4) {
    xhr.status === 200 ? console.log(xhr.responseText) : console.error(xhr.statusText)
  }
}
xhr.open('GET', 'https://yoursite.com')
xhr.send()
```

Tratando erros em callbacks

Como você trata erros em callbacks? Uma estratégia muito comum é usar o que o Node.js adotou: o primeiro parâmetro em qualquer função callback é o objeto de erro: **error-first callbacks**

Se não há erro, o objeto é nulo (`null`). Se há um erro, ele contém alguma descrição do erro e outras informações.

```
fs.readFile('/file.json', (err, data) => {
  if (err !== null) {
    // tratando erro
    console.log(err)
    return
  }

  // sem erros, processe o data
  console.log(data)
})
```

O problema com callbacks

Callbacks são ótimas para casos simples!

Entretanto, toda callback adiciona um nível de aninhamento, e quando você têm muitas callbacks, o código começa a se complicar rapidamente:

```
window.addEventListener('load', () => {
  document.getElementById('button').addEventListener('click', () => {
    setTimeout(() => {
      items.forEach(item => {
        // seu código aqui
      })
    }, 2000)
  })
})
```

Esse é só um exemplo simples com 4 níveis de aninhamento, mas eu já vi códigos com muito mais níveis e não é nem um pouco divertido.

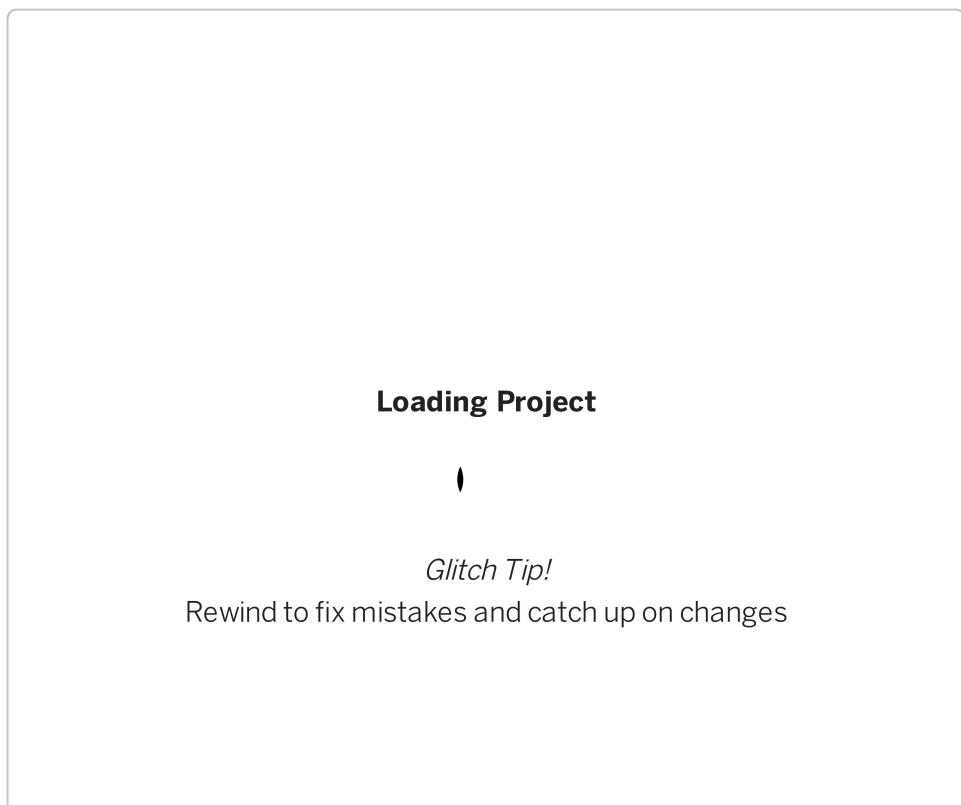
Como resolver isso?

Alternativas à callbacks

A partir do ES6, o JavaScript introduziu várias funcionalidades que nos ajudam a lidar com código assíncrono sem o uso de callbacks: Promises (ES6) e Async/Await (ES2017).

Promises no JavaScript

Introdução a Promises



Uma promise (promessa) é comumente definida como **um proxy para um valor que eventualmente ficará disponível**.

Promises são um jeito de lidar com código assíncrono, sem ficar preso em [“inferno de callbacks” \(callback hell\)](#).

Promises são parte da linguagem por anos (introduzidas e padronizadas no ES2015), e se tornaram mais integradas recentemente, com **async** e **await** no ES2017.

Funções **async** usam promises por baixo dos panos, então entender como promises funcionam é fundamental para entender como **async** e **await** funcionam.

Como promises funcionam, resumidamente

Uma vez que a promise tenha sido chamada, ela inicia em um **estado pendente (pending)**. Isso significa que a função que a chamou continua executando, enquanto a promise estiver pendente até que se resolva, retornando à função que a invocou os dados que foram requisitados.

A promise criada vai eventualmente se encerrar em um **estado resolvido (resolved)**, ou em um **estado rejeitado (rejected)**, chamando as respectivas funções callback (**then** e **catch**) ao terminar.

Quais APIs JS usam promises?

Em adição ao seu código e o de bibliotecas, promises são usadas por padrão em APIs web modernas, como:

- Battery API
- Fetch API
- Service Workers

É improvável que você mesmo *não* esteja utilizando promises no JavaScript moderno, então vamos começar a mergulhar nelas.

Criando uma promise

A Promise API expõe um construtor `Promise`, por onde você inicializa usando `new Promise()`:

```
let done = true

const isItDoneYet = new Promise((resolve, reject) => {
  if (done) {
    const workDone = 'Here is the thing I built'
    resolve(workDone)
  } else {
    const why = 'Still working on something else'
    reject(why)
  }
})
```

Como você pode ver, a promise checa a constante global `done`, e se for `true`, a promise vai para o estado `resolved` (uma vez que a callback `resolve` tenha sido chamada); caso contrário, a callback `reject` é executada, colocando a promise em um estado `rejected`. (Se nenhuma dessas funções é chamada, a promise permanece no estado `pending`).

Usando `resolve` e `reject`, podemos nos comunicar com a função invocadora dizendo qual estado a promise estava, e o que fazer com isso. No caso acima nós só retornamos uma string, mas poderia ser um objeto, ou mesmo `null`. Só de termos criado a promise no trecho acima, ela já começou a executar. É importante entender o que está acontecendo na seção [Consumindo uma promise](#) logo abaixo.

Um exemplo mais comum que você pode encontrar por aí é uma técnica chamada **Promisifying (promissificar)**. Essa técnica é um jeito de poder usar uma função JavaScript clássica que recebe uma callback, e retornar uma promise:

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

const getFile = (fileName) => {
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, (err, data) => {
      if (err) {
        reject(err) // chamar `reject` vai fazer com que a promise
                    // e não queremos ir mais longe
      }
      resolve(data)
    })
  })
}

getFile('/etc/passwd')
  .then(data => console.log(data))
  .catch(err => console.error(err))
```

Em versões recentes do Node.js, você não precisa fazer essa conversão manual para grande parte da API. Há uma função *promisifying* disponível no [módulo util](#) que fará isso por você, dado que a função que você esteja “promissificando” siga a assinatura correta.

Consumindo uma promise

Na última seção, nós introduzimos como uma promise é criada.

Agora vamos ver como uma promise pode ser *consumida* ou usada.

```
const isItDoneYet = new Promise(/* ... como acima ... */)
//...

const checkIfItsDone = () => {
```

```
    .isItDoneYet()
      .then(ok => {
        console.log(ok)
      })
      .catch(err => {
        console.error(err)
      })
  }
}
```

Rodar `checkIfItsDone()` vai especificar funções para executar quando a promise `isItDoneYet` resolver (na chamada `then`) ou rejeitar (na chamada `catch`).

Encadeando promises

Uma promise pode ser retornada para outra promise, criando uma cadeia de promises.

Um ótimo exemplo de encadeamento de promises é a Fetch API, que podemos usar para obter um recurso e enfileirar uma cadeia de promises para executar quando o recurso for obtido.

A Fetch API tem um mecanismo baseado em promise, e chamar `fetch()` é equivalente a definir suas próprias promises usando `new Promise()`.

Exemplo de encadeamento de promises

```
const status = response => {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response)
  }
  return Promise.reject(new Error(response.statusText))
}
```

```
const json = response => response.json()

fetch('/todos.json')
  .then(status) // note que a função `status` é na verdade **chamada**
  .then(json) // da mesma forma, a única diferença aqui é que a
  .then(data => { // ... por isso que `data` aparece aqui como prim
    console.log('Request succeeded with JSON response', data)
  })
  .catch(error => {
    console.log('Request failed', error)
  })
```

Nesse exemplo, nós chamamos `fetch()` para obter uma lista de items TODO do arquivo `todos.json` localizado na raiz do domínio, e nós criamos uma cadeia de promises.

Rodar `fetch()` retorna uma [response \(resposta\)](#), que contém várias propriedades, e dentre elas as que usamos no exemplo:

- `status`, um valor numérico representando o código de status HTTP
- `statusText`, uma mensagem de status, que será `OK` se a requisição for bem-sucedida.

`response` também tem um método `json()`, que retorna uma promise que vai resolver com o conteúdo do body processado e transformado em JSON.

Então dadas essas promises, isso é o que acontece: a primeira promise na cadeia é a função que nós definimos, chamada `status()`, que checa o status da `response` e se não for uma resposta bem-sucedida (entre 200 e 299), a promise é rejeitada.

Essa operação fará com que a cadeia de promises pule todas promises encadeadas e pule diretamente para o `catch()` no final, logando o texto `Request failed` e a mensagem de erro.

Se em vez disso obter sucedido, é chamada a função `json()` que definimos. Desde que a promise anterior, quando bem-sucedida, tenha retornado o objeto `response`, nós o obtemos como um input para a segunda promise.

Nesse caso, nós retornamos os dados processados em JSON, assim a terceira promise recebe o JSON diretamente:

```
.then((data) => {
  .  console.log('Request succeeded with JSON response', data)
})
```

e nós simplesmente o logamos no console.

Tratando erros

No exemplo da seção anterior, nós tínhamos um `catch` que foi adicionado na cadeia de promises.

Quando qualquer coisa na cadeia de promises falha e dispara um erro ou rejeita uma promise, o controle vai para o `catch()` mais próximo na cadeia.

```
new Promise((resolve, reject) => {
  .  throw new Error('Error')
}).catch(err => {
  .  console.error(err)
})

// or

new Promise((resolve, reject) => {
  .  reject('Error')
}).catch(err => {
  .  console.error(err)
})
```

Cascadeando erros

Se dentro do `catch()` você lançar um erro, você pode adicionar um segundo `catch()` para tratá-lo, e assim por diante.

```
new Promise((resolve, reject) => {
  .throw new Error('Error')
})
  .catch(err => {
    .throw new Error('Error')
  })
  .catch(err => {
    console.error(err)
  })
```

Orquestrando promises

`Promise.all()`

Se você precisa sincronizar promises diferentes, `Promise.all()` te ajuda a definir uma lista de promises, e executa algo quando elas todas são resolvidas.

Exemplo:

```
const f1 = fetch('/something.json')
const f2 = fetch('/something2.json')

Promise.all([f1, f2])
  .then(res => {
    .console.log('Array of results', res)
```

```
    })
    .catch(err => {
      console.error(err)
    })
  )
```

A sintaxe de atribuição por desestruturação do ES2015 permite que você também faça isso:

```
Promise.all([f1, f2]).then(([res1, res2]) => {
  console.log('Results', res1, res2)
})
```

É claro que você não está limitado a só usar o `fetch`, **qualquer promise pode ser usada dessa forma.**

Promise.race()

`Promise.race()` roda quando a primeira das promises que você passar for resolvida, e roda a callback anexada apenas uma vez, com o resultado da primeira promise resolvida.

Exemplo:

```
const first = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'first')
})
const second = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'second')
})

Promise.race([first, second]).then(result => {
  console.log(result) // second
})
```

Erros comuns

Uncaught TypeError: undefined is not a promise

Se você obter um erro `Uncaught TypeError: undefined is not a promise` no console, certifique-se de usar `new Promise()` em vez de apenas `Promise()`.

UnhandledPromiseRejectionWarning

Isso significa que a promise que você chamou foi rejeitada, mas não tinha nenhum `catch` preparado para tratar o erro. Adicione um `catch` depois do `then` causador do erro para tratá-lo propriamente.

Programação assíncrona moderna com Async Await no Javascript

Introdução

O JavaScript evoluiu de callbacks para promises (ES2015) em um curto período de tempo, e desde o ES2017, JavaScript assíncrono está ainda mais simples com a sintaxe `async/await`.

Funções `async` são uma combinação de promises e generators, e basicamente, elas são uma abstração de alto nível das promises. Permita-me repetir: `async/await` é feito com promises.

Por que a sintaxe `async/await` foi criada?

Ela reduz o amontoado de código em torno das promises e a limitação de “não quebre a corrente” das promises encadeadas.

Quando as Promises foram introduzidas no ES2016, elas foram feitas para resolver um problema com código assíncrono, e elas fizeram isso, mas ao longos dos 2 anos que separaram o ES2015 do ES2017, ficou claro que *promises poderiam não ser a solução definitiva*.

Promises foram introduzidas para resolver o famoso problema de *callback hell*, mas elas introduziram complexidade por si só, e complexidade sintática.

Elas foram boas primitivas enquanto uma sintaxe melhor não surgia, então, quando chegou a hora certa, obtivemos as **funções async**.

Elas fazem o código parecer síncrono, mas por baixo dos panos ele é assíncrono e não bloqueante.

Como isso funciona

Uma função `async` retorna uma promise, como nesse exemplo:

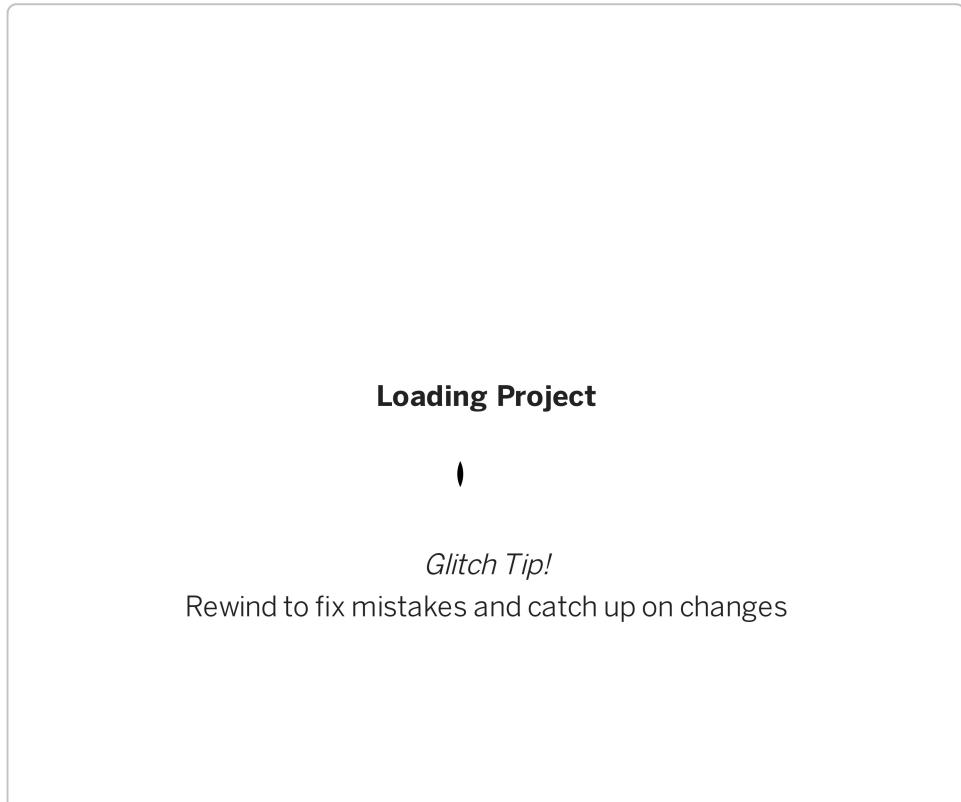
```
const doSomethingAsync = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('I did something'), 3000)
  })
}
```

Quando você quer chamar essa função você prefixa um `await`, e o código corrente vai parar até que a promise seja resolvida ou rejeitada. Uma ressalva: a função deve ser definida como `async`. Segue um exemplo:

```
const doSomething = async () => {
  console.log(await doSomethingAsync())
}
```

Um exemplo rápido

Esse é um exemplo bem simples de `async/await` sendo usado para rodar uma função assíncronamente:



“Promissifique” tudo

Prefixar o termo `async` em qualquer função implica que a função vai retornar uma promise.

Mesmo que isso não seja feito tão explicitamente, internamente fará com que retorne uma promise.

Por isso que esse código é válido:

```
const aFunction = async () => {
  return 'test'
}

aFunction().then(alert) // Isso vai invocar um alert com o texto 'te
```

e o mesmo aqui:

```
const aFunction = async () => {
  return Promise.resolve('test')
}

aFunction().then(alert) // Isso vai invocar um alert com o texto 'te
```

O código é muito mais simples de ler

Como você pode ver no exemplo acima, nosso código parece muito simples. Compare-o com os códigos usando promises simples, encadeamento e funções callback.

E esse é um exemplo bem simples, os maiores benefícios vão aparecer quando o código for mais complexo.

Por exemplo, aqui temos um código de como obter um recurso JSON e “parsear”, usando promises:

```
const getFirstUserData = () => {
  return fetch('/users.json') // pega a lista de users
    .then(response => response.json()) // parse JSON
    .then(users => users[0]) // pega o primeiro usuário
    .then(user => fetch(`/users/${user.name}`)) // pega os dados des
    .then(userResponse => userResponse.json()) // parse JSON
```

}

`getFirstUserData()`

E aqui temos a mesma funcionalidade usando `async/await`:

```
const getFirstUserData = async () => {
  const response = await fetch('/users.json') // pega a lista de usuários
  const users = await response.json() // parse JSON
  const user = users[0] // pega o primeiro usuário
  const userResponse = await fetch(`/users/${user.name}`) // pega os detalhes do usuário
  const userData = await userResponse.json() // parse JSON
  return userData
}

getFirstUserData()
```

Múltiplas funções `async` em série

Funções `async` podem ser encadeadas facilmente, e a sintaxe é muito mais legível do que usando promises simples:

Loading Project

Glitch Tip!

Securely store app secrets and config in .env

Mais fácil de debugar

É difícil debugar promises porque o debugger não passa por cima de código assíncrono.

Já Async/await facilita muito isso porque para o compilador é como se fosse apenas código síncrono.

O Event emitter

Se você já trabalhou com JavaScript no browser, já sabe o quanto grande é a quantidade de ações do usuário que são tratadas por eventos: clicks do mouse, teclas sendo pressionadas, movimentos do mouse, e muito mais.

No lado do backend, o Node.js nos oferece a opção de trabalhar com um sistema similar usando o [módulo events](#).

Esse módulo, em particular, fornece a classe `EventEmitter`, que nós utilizaremos para lidar com nossos eventos.

Você a inicializa usando

```
const EventEmitter = require('events')
const eventEmitter = new EventEmitter()
```

Esse objeto expõe, entre muitos outros, os métodos `on` e `emit`.

- `emit` é usado para acionar um evento
- `on` é usado para adicionar uma função callback que será executada quando o evento for acionado

Por exemplo, vamos criar um evento `start`, e para dar uma amostra grátis, vamos reagir a ele logando no console:

```
eventEmitter.on('start', () => {
  console.log('started')
})
```

Quando rodamos

```
eventEmitter.emit('start')
```

a função que lida com o evento é acionada, e obtemos o log.

Você pode passar argumentos à função passando-os como argumentos adicionais do `emit()`:

```
eventEmitter.on('start', number => {
  console.log(`started ${number}`)
})
```

```
eventEmitter.emit('start', 23)
```

Múltipos argumentos:

```
eventEmitter.on('start', (start, end) => {
  console.log(`started from ${start} to ${end}`)
})
```

```
eventEmitter.emit('start', 1, 100)
```

O objeto EventEmitter também expõe diversos outros métodos que interagem com eventos, como

- `once()`: escuta o evento apenas uma vez
- `removeListener() / off()`: remove um escutador (listener) de um evento
- `removeAllListeners()`: remove todos escutadores (listeners) de um evento

Você pode ler o detalhe de todos eles na página do módulo events em
<https://nodejs.org/api/events.html>

Construa um HTTP Server

Aqui temos um simples web server HTTP de Hello World:

Loading Project

Glitch Tip!

Rewind to fix mistakes and catch up on changes

Vamos analisar esse código brevemente. Nós incluimos o [módulo http](#).

Nós usamos o módulo para criar um servidor HTTP.

É definido para o servidor escutar na porta especificada, `3000`. Quando o servidor está pronto, a função callback `listen` é chamada.

A função callback que passamos é a que será executada em toda requisição. Sempre que uma nova requisição é recebida, o [evento request](#) é chamado, fornecendo dois objetos: `request` (instância de `http.IncomingMessage`) e `response` (instância de `http.ServerResponse`).

`request` contém os detalhes da requisição. Através dele, nós acessamos os cabeçalhos (headers) e os dados da requisição.

`response` é usado para popular os dados que vamos retornar ao cliente.

No seguinte caso

```
res.statusCode = 200
```

nós definimos a propriedade statusCode para 200, para indicar uma resposta bem sucedida.

Nós também definimos a header Content-Type:

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

e fechamos a resposta, adicionando o conteúdo como um argumento do `end()`:

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

Fazendo requisições HTTP com o Node.js

Realizando uma requisição do tipo GET

```
const https = require('https')
const options = {
  hostname: 'whatever.com',
  port: 443,
  path: '/todos',
  method: 'GET'
}

const req = https.request(options, res => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', d => {
    process.stdout.write(d)
  })
})
```

```
req.on('error', error => {
  console.error(error)
})

req.end()
```

Realizando uma requisição do tipo POST

```
const https = require('https')

const data = JSON.stringify({
  todo: 'Buy the milk'
})

const options = {
  hostname: 'whatever.com',
  port: 443,
  path: '/todos',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': data.length
  }
}

const req = https.request(options, res => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', d => {
    process.stdout.write(d)
  })
})

req.on('error', error => {
  console.error(error)
})
```

```
req.write(data)
req.end()
```

Requisições do tipos PUT e DELETE

Requisições PUT e DELETE usam o mesmo formato da requisição POST, mudando apenas o valor do `options.method`.

Faça um HTTP POST request com Node.js

Existem diversas maneiras de realizar uma requisição do tipo POST no Node.js, dependendo do nível de abstração que você quer utilizar.

O jeito mais simples de fazer uma requisição HTTP usando Node.js é usar a [biblioteca Axios](#):

```
const axios = require('axios')

axios
  .post('https://whatever.com/todos', {
    todo: 'Buy the milk'
  })
  .then(res => {
    console.log(`statusCode: ${res.statusCode}`)
    console.log(res)
  })
  .catch(error => {
    console.error(error)
  })
```

Utilizar o Axios requer o uso de uma biblioteca de terceiros.

É possível fazer uma requisição POST usando apenas os módulos nativos do Node.js, porém é mais verboso que as duas opções anteriores:

```
const https = require('https')

const data = JSON.stringify({
  todo: 'Buy the milk'
})

const options = {
  hostname: 'whatever.com',
  port: 443,
  path: '/todos',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': data.length
  }
}

const req = https.request(options, res => {
  console.log(`statusCode: ${res.statusCode}`)

  res.on('data', d => {
    process.stdout.write(d)
  })
})

req.on('error', error => {
  console.error(error)
})

req.write(data)
req.end()
```

Faça uma requisição GET de HTTP com body usando Node.js

Aqui temos como você pode extrair dados que foram enviados como JSON no corpo da requisição:

Se você está usando Express, é bem simples: utilize o módulo `body-parser`.

Por exemplo, para obter o corpo dessa requisição:

```
const axios = require('axios')

axios.post('https://whatever.com/todos', {
  todo: 'Buy the milk'
})
```

Esse é o código do servidor que vai receber a requisição:

```
const express = require('express')
const app = express()

app.use(
  express.urlencoded({
    extended: true
  })
)

app.use(express.json())

app.post('/todos', (req, res) => {
  console.log(req.body.todo)
})
```

Se você não está usando Express, e você quer fazer isso em Node.js raiz, terá um pouco mais de trabalho, é claro, pois o Express abstrai muitas coisas pra você.

O ponto chave pra se entender é que quando você inicializa um servidor HTTP usando `http.createServer()`, a callback é chamada quando o servidor obtém todos os cabeçalhos HTTP, mas não o body (corpo da requisição).

O objeto `request` passado na conexão de callback é uma stream.

Então, nós devemos escutar pelo conteúdo do body a ser processado, e ele é processado em chunks(pedaços).

Primeiro nós obtemos os dados escutando os eventos de `data` da stream, e quando eles finalizam, a stream emite o evento `end`:

```
const server = http.createServer((req, res) => {
  // nós podemos acessar os cabeçalhos HTTP
  req.on('data', chunk => {
    console.log(`Data chunk available: ${chunk}`)
  })
  req.on('end', () => {
    // fim dos eventos data
  })
})
```

Então para acessar os dados, assumindo que esperamos receber uma string, nós devemos colocá-los em um array:

```
const server = http.createServer((req, res) => {
  let data = '';
  req.on('data', chunk => {
    data += chunk;
  })
  req.on('end', () => {
    JSON.parse(data).todo // 'Buy the milk'
  })
})
```

Trabalhando com file descriptors node Node.js

Antes que você seja capaz de interagir com um arquivo presente no seu filesystem, você deve obter um file descriptor (descriptor de arquivo).

Um file descriptor é o que é retornado ao abrir um arquivo utilizando o método `open()` do módulo `fs`:

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

fs.open('/Users/joe/test.txt', 'r', (err, fd) => {
  // fd é o nosso file descriptor
})
```

Note o `r` que usamos como segundo parâmetro da chamada `fs.open()`.

Essa flag significa que nós abrimos o arquivo para leitura.

Outras flags que você normalmente utilizará são:

- `r+` abre o arquivo para leitura e escrita
- `w+` abre o arquivo para leitura e escrita, posicionando a stream no início do arquivo. O arquivo é criado se não existir.
- `a` abre o arquivo para escrita, posicionando a stream no fim do arquivo. O arquivo é criado se não existir.
- `a+` abre o arquivo para leitura e escrita, posicionando a stream no fim do arquivo. O arquivo é criado se não existir.

Você também pode abrir o arquivo utilizando o método `fs.openSync`, que retorna o file descriptor, em vez de fornecê-lo dentro de uma callback:

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

try {
  const fd = fs.openSync('/Users/joe/test.txt', 'r')
} catch (err) {
  console.error(err)
}
```

Uma vez que você obtenha o file descriptor, independentemente do método escolhido, você pode fazer todas operações que o requerem, como chamar `fs.open()` e muitas outras operações que interagem com o filesystem.

Node.js file stats

Todo arquivo vem com um conjunto de detalhes que nós podemos inspecionar usando Node.js.

Em particular, usando o método `stat()` do módulo `fs`.

Você o chama passando o caminho do arquivo, e uma vez que o Node.js obtenha os detalhes do arquivo ele chamará a função callback que você passar, com 2 parâmetros: uma mensagem de erro, e os detalhes do arquivo.

```
const fs = require('fs')
fs.stat('/Users/joe/test.txt', (err, stats) => {
  if (err) {
    console.error(err)
    return
  }
  // nós temos acesso aos detalhes do arquivo no `stats`
})
```

O Node.js também fornece um método síncrono, que bloqueia a thread até que os detalhes do arquivo estejam prontos:

```
const fs = require('fs')
try {
  const stats = fs.statSync('/Users/joe/test.txt')
} catch (err) {
  console.error(err)
}
```

As informações do arquivo estão inclusas na variável stats. Que tipos de informações nós podemos extrair usando o stats?

Muitas, incluindo:

- se o arquivo é um diretório ou um arquivo, usando `stats.isFile()` e `stats.isDirectory()`
- se o arquivo é um link simbólico, usando `stats.isSymbolicLink()`
- o tamanho do arquivo, usando `stats.size`.

Há outros métodos avançados, mas a maior parte do que você vai usar na programação do dia a dia são esses.

```
const fs = require('fs')
fs.stat('/Users/joe/test.txt', (err, stats) => {
  if (err) {
    console.error(err)
    return
  }

  stats.isFile() // true
  stats.isDirectory() // false
  stats.isSymbolicLink() // false
  stats.size // 1024000 //≈ 1MB
})
```

Node.js File Paths

Todo arquivo no sistema tem um caminho.

No Linux e no macOS, um caminho tem essa cara:

/users/joe/file.txt

enquanto que em computadores Windows é bem diferente, a estrutura é algo assim:

```
C:\users\joe\file.txt
```

Você precisa prestar atenção quando está utilizando caminhos na sua aplicação, as diferenças devem ser levadas em consideração.

Para trabalhar com caminhos, inclua esse módulo em seus arquivos:

```
const path = require('path')
```

Obtendo informações de um caminho

Dado um caminho, você pode extrair informações dele usando esses métodos:

- `dirname`: obtêm a pasta pai do arquivo
- `basename`: obtêm a parte do nome do arquivo
- `extname`: obtêm a parte da extensão do arquivo

Exemplo:

```
const notes = '/users/joe/notes.txt'

path.dirname(notes) // /users/joe
path.basename(notes) // notes.txt
path.extname(notes) // .txt
```

Você pode obter o nome do arquivo sem a extensão ao especificar um segundo argumento no `basename`:

```
path.basename(notes, path.extname(notes)) // notes
```

Trabalhando com caminhos

Você pode unir duas ou mais partes de um caminho ao utilizar `path.join()`:

```
const name = 'joe'  
path.join('/', 'users', name, 'notes.txt') // '/users/joe/notes.txt'
```

Você pode obter o cálculo do caminho absoluto de um caminho relativo utilizando `path.resolve()`:

```
path.resolve('joe.txt') // '/Users/joe/joe.txt' se rodar a partir da
```

Nesse caso o Nodejs vai simplesmente adicionar `/joe.txt` na pasta atual. Se você especificar uma pasta como segundo parâmetro, o `resolve` vai usar o primeiro como base pro segundo:

```
path.resolve('tmp', 'joe.txt') // '/Users/joe/tmp/joe.txt' se rodar a
```

Se o primeiro parâmetro começa com uma barra, isso implica que é um caminho absoluto:

```
path.resolve('/etc', 'joe.txt') // '/etc/joe.txt'
```

`path.normalize()` é outra função bem útil, que tentará calcular o caminho real, quando ele contêm especificadores relativos como `.` ou `..`, ou barras duplas:

```
path.normalize('/users/joe/.../test.txt') // /users/test.txt
```

Ambos resolve e normalize não irão validar se o caminho existe. Eles só calculam um caminho baseado nas informações fornecidas.

Lendo arquivos com Node.js

O jeito mais fácil de ler um arquivo no Node.js é utilizar o método `fs.readFile()`, passando o caminho do arquivo, a codificação e uma função callback que será invocada com os dados do arquivo (e o erro):

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

fs.readFile('/Users/joe/test.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(data)
})
```

Alternativamente, você pode usar a versão síncrona, `fs.readFileSync()`:

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

try {
  const data = fs.readFileSync('/Users/joe/test.txt', 'utf8')
  console.log(data)
} catch (err) {
  console.error(err)
}
```

Ambos `fs.readFile()` e `fs.readFileSync()` lêem todo conteúdo do arquivo em memória antes de retornar os dados.

Isso significa que arquivos grandes vão causar grandes impactos no consumo de memória e na velocidade de execução do programa.

Nesse caso, uma opção melhor é ler o conteúdo do arquivo usando streams.

Escrevendo arquivos com Node.js

O jeito mais fácil de escrever em arquivos com Node.js é utilizando a API `fs.writeFile()`.

Exemplo:

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

const content = 'Some content!'

fs.writeFile('/Users/joe/test.txt', content, err => {
  if (err) {
    console.error(err)
    return
  }
  // arquivo escrito com sucesso
})
```

Alternativamente, você pode usar a versão síncrona, `fs.writeFileSync()`:

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

const content = 'Some content!'

try {
  const data = fs.writeFileSync('/Users/joe/test.txt', content)
  // arquivo escrito com sucesso
} catch (err) {
```

```
    console.error(err)
}
```

Por padrão, essa API vai **apagar o conteúdo do arquivo** se ele existir. By default, this API will **replace the contents of the file** if it does already exist.

Você pode modificar o comportamento padrão especificando uma flag:

```
fs.writeFile('/Users/joe/test.txt', content, { flag: 'a+' }, err => {
```

As flags que você normalmente utilizará são

- `r+` abre o arquivo para leitura e escrita
- `w+` abre o arquivo para leitura e escrita, posicionando a stream no início do arquivo. O arquivo é criado se não existir
- `a` abre o arquivo para escrita, posicionando a stream no fim do arquivo. O arquivo é criado se não existir
- `a+` abre o arquivo para leitura e escrita, posicionando a stream no fim do arquivo. O arquivo é criado se não existir

(você pode encontrar mais flags em
https://nodejs.org/api/fs.html#fs_file_system_flags)

Acrescentar conteúdo em um arquivo

Um método muito útil acrescentar conteúdo no fim de um arquivo é o `fs.appendFile()` (e sua contraparte `fs.appendFileSync()`):

```
const content = 'Some content!'

fs.appendFile('file.log', content, err => {
```

```
if (err) {  
    console.error(err)  
    return  
}  
// feito!  
})
```

Usando streams

Todos esses métodos escrevem o conteúdo completo no arquivo antes de retornarem o controle de volta ao programa (na versão assíncrona, isso significa executar a callback)

Nesse caso, uma opção melhor é escrever o conteúdo do arquivo usando streams.

Trabalhando com folders no Node.js

O módulo nativo `fs` do Node.js fornece diversos métodos versáteis para trabalhar com pastas.

Valide se uma pasta existe

Utilize `fs.access()` para validar se a pasta existe e o Node.js pode acessá-la com suas permissões.

Crie uma nova pasta

Utilize `fs.mkdir()` ou `fs.mkdirSync()` para criar uma nova pasta.

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

const folderName = '/Users/joe/test'

try {
  if (!fs.existsSync(folderName)) {
    fs.mkdirSync(folderName)
  }
} catch (err) {
  console.error(err)
}
```

Leia o conteúdo de um diretório

Utilize `fs.readdir()` ou `fs.readdirSync()` para ler os conteúdos de um diretório.

Esse trecho de código lê o conteúdo de uma pasta, ambos arquivos e subpastas, e retorna o caminho relativo deles:

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

const folderPath = '/Users/joe'

fs.readdirSync(folderPath)
```

Você pode obter o caminho inteiro:

```
fs.readdirSync(folderPath).map(fileName => {
  return path.join(folderPath, fileName)
})
```

Você também pode filtrar os resultados para retornar apenas os arquivos, excluindo as pastas:

```
const isFile = fileName => {
  return fs.lstatSync(fileName).isFile()
}

fs.readdirSync(folderPath).map(fileName => {
  return path.join(folderPath, fileName)
})
.filter(isFile)
```

Renomeie uma pasta

Utilize `fs.rename()` ou `fs.renameSync()` para renomear uma pasta. O primeiro parâmetro é o caminho atual, o segundo o novo caminho:

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

fs.rename('/Users/joe', '/Users/roger', err => {
  if (err) {
    console.error(err)
    return
  }
  // feito
})
```

`fs.renameSync()` é a versão síncrona:

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

try {
  fs.renameSync('/Users/joe', '/Users/roger')
} catch (err) {
```

```
    .console.error(err)
}
```

Delete uma pasta

Utilize `fs.rmdir()` ou `fs.rmdirSync()` para deletar uma pasta.

Deletar uma pasta que tem conteúdo pode ser mais complicado do que o esperado.

Nesse caso é melhor instalar o módulo `fs-extra`, que é bem popular e atualizado. É uma substituição do módulo `fs`, que possui mais funcionalidades.

Nesse caso o método que você quer é o `remove()`.

Instale usando

```
npm install fs-extra
```

e use-o assim:

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

const folder = '/Users/joe'

fs.remove(folder, err => {
  .console.error(err)
})
```

Também pode ser utilizado com promises:

```
fs.remove(folder)
  .then(() => {
```

```
//done
})
.catch(err => {
  console.error(err)
})
```

ou com `async/await`:

```
async function removeFolder(folder) {
  try {
    await fs.remove(folder)
    //done
  } catch (err) {
    console.error(err)
  }
}

const folder = '/Users/joe'
removeFolder(folder)
```

O módulo `fs`

O módulo `fs` disponibiliza diversas funcionalidades úteis para acessar e interagir com o file system.

Não há necessidade de intalá-lo. Sendo parte do núcleo do Node.js, basta importar para poder usá-lo:

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

Feito isso, você tem acesso a todos os métodos dele, o que inclui:

- `fs.access()`: checa se o arquivo existe e se o Node.js pode acessá-lo com suas permissões atuais

- `fs.appendFile()`: acrescenta dados em um arquivo. Se o arquivo não existir, ele é criado.
- `fs.chmod()`: muda as permissões de um arquivo especificado pelo nome que foi passado. Relacionado: `fs.lchmod()`, `fs.fchmod()`
- `fs.chown()`: muda o proprietário e o grupo do arquivo especificado pelo nome que foi passado. Relacionado: `fs.fchown()`, `fs.lchown()`
- `fs.close()`: fecha um descriptor de arquivo
- `fs.copyFile()`: copia um arquivo
- `fs.createReadStream()`: cria uma stream de arquivo para leitura
- `fs.createWriteStream()`: cria uma stream de arquivo para escrita
- `fs.link()`: cria um novo hard link para um arquivo
- `fs.mkdir()`: cria um novo diretório
- `fs.mkdtemp()`: cria um novo diretório temporário
- `fs.open()`: define o file mode
- `fs.readdir()`: lê os conteúdos de um diretório
- `fs.readFile()`: lê os conteúdos de um arquivo. Relacionado: `fs.read()`
- `fs.readlink()`: lê o valor de um link simbólico
- `fs.realpath()`: resolve caminhos relativos de arquivo com ponteiros (`..`, `...`) para o caminho completo
- `fs.rename()`: renomeia um arquivo ou pasta
- `fs.rmdir()`: deleta um diretório
- `fs.stat()`: retorna o status do arquivo identificado pelo nome que foi passado. Relacionado: `fs.fstat()`, `fs.lstat()`
- `fs.symlink()`: cria um novo link simbólico para um arquivo

- `fs.truncate()`: trunca o arquivo identificado pelo nome que foi passado para o tamanho especificado. Relacionado: `fs.ftruncate()`
- `fs.unlink()`: remove um arquivo ou um link simbólico
- `fs.unwatchFile()`: para de observar mudanças em um arquivo
- `fs.utimes()`: muda a timestamp do arquivo identificado pelo nome que foi passado. Relacionado: `fs.futimes()`
- `fs.watchFile()`: começa a observar mudanças em um arquivo. Relacionado: `fs.watch()`
- `fs.writeFile()`: escreve dados em um arquivo. Relacionado: `fs.write()`

Uma coisa peculiar sobre o módulo `fs` é que todos os métodos são assíncronos por padrão, mas eles também podem rodar de forma síncrona adicionando `Sync` no nome.

Por exemplo:

- `fs.rename()`
- `fs.renameSync()`
- `fs.write()`
- `fs.writeFileSync()`

Isso faz uma enorme diferença no fluxo da sua aplicação.

O Node.js 10 inclui [suporte experimental](#) para uma API baseada em promises

Por exemplo, vamos analisar o método `fs.rename()`. A API assíncrona é usada com uma callback:

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

fs.rename('before.json', 'after.json', err => {
  if (err) {
    return console.error(err)
  }

  // feito
})
```

Uma API síncrona pode ser usada com um bloco de try/catch para tratar erros:

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

try {
  fs.renameSync('before.json', 'after.json')
  // feito
} catch (err) {
  console.error(err)
}
```

A principal diferença aqui é que a execução do seu script no segundo exemplo vai bloquear a thread, até que a operação no arquivo seja bem-sucedida.

O módulo path

O módulo `path` disponibiliza diversas funcionalidades úteis para acessar e interagir com o file system.

Não há necessidade de intalá-lo. Sendo parte do núcleo do Node.js, basta importar para poder usá-lo:

```
const path = require('path')
```

Esse módulo possui o `path.sep` que provê o caracter separador de segmento de caminho (\ no Windows, e / no Linux / macOS), e o `path.delimiter` que provê o caracter delimitador de caminho (; no Windows, e : no Linux / macOS).

Esse são os métodos do `path`:

`path.basename()`

Retorna a última parte de um caminho. Um segundo parâmetro pode filtrar a extensão do arquivo:

```
require('path').basename('/test/something') // something
require('path').basename('/test/something.txt') // something.txt
require('path').basename('/test/something.txt', '.txt') // something
```

`path.dirname()`

Retorna a parte do diretório de um caminho:

```
require('path').dirname('/test/something') // /test
require('path').dirname('/test/something/file.txt') // /test/somethi
```

`path.extname()`

Retorna a parte da extensão de um caminho

```
require('path').extname('/test/something') // ''
require('path').extname('/test/something/file.txt') // '.txt'
```

path.isAbsolute()

Retorna true se o caminho for absoluto

```
require('path').isAbsolute('/test/something') // true  
require('path').isAbsolute('./test/something') // false
```

path.join()

Junta duas ou mais partes de um caminho:

```
const name = 'joe'  
require('path').join('/', 'users', name, 'notes.txt') // '/users/joe,
```

path.normalize()

Tenta calcular o caminho correto quando ele contém especificadores relativos como . ou .., ou barras duplas:

```
require('path').normalize('/users/joe/../test.txt') // '/users/test
```

path.parse()

Cria um objeto contendo os segmentos que compõe o caminho fornecido:

- `root`: a raiz
- `dir`: o caminho da pasta a partir da raiz
- `base`: o nome do arquivo + extensão
- `name`: o nome do arquivo
- `ext`: a extensão do arquivo

Exemplo:

```
require('path').parse('/users/test.txt')
```

resulta em

```
{
  root: '/',
  dir: '/users',
  base: 'test.txt',
  ext: '.txt',
  name: 'test'
}
```

path.relative()

Aceita 2 caminhos como argumentos. Retorna o caminho relativo do primeiro para o segundo, baseando-se no diretório atual.

Exemplo:

```
require('path').relative('/Users/joe', '/Users/joe/test.txt') // 'test.txt'
require('path').relative('/Users/joe', '/Users/joe/something/test.txt')
```

path.resolve()

Você pode obter o cálculo do caminho absoluto de um caminho relativo usando `path.resolve()`:

```
path.resolve('joe.txt') // '/Users/joe/joe.txt' se rodar da minha hoi
```

Ao especificar um segundo parâmetro, o `resolve` vai usar o primeiro como base para o segundo:

```
path.resolve('tmp', 'joe.txt') // '/Users/joe/tmp/joe.txt' se rodar da minha hoi
```

Se o primeiro parâmetro começar com uma barra, isso significa que é um caminho absoluto:

```
path.resolve('/etc', 'joe.txt') // '/etc/joe.txt'
```

O Módulo OS

Esse módulo disponibiliza diversas funções que você pode usar para obter informações do sistema operacional e do computador em que o programa está executando, e interagir com ele.

```
const os = require('os')
```

Há algumas propriedades úteis que nos dizem algumas coisas importantes relacionadas a manipulação de arquivos:

`os.EOL` retorna o a sequência delimitadora de linha (caracter de pular linha). `\n` no Linux e macOS, e `\r\n` no Windows.

`os.constants.signals` retorna todas contantes relacionadas a sinais de processos, como SIGHUP, SIGKILL e outros.

`os.constants.errno` define as constantes para reportar erros, como EADDRINUSE, EOVERRLOW e mais outras.

Você pode ver todas constantes em
https://nodejs.org/api/os.html#os_signal_constants.

Agora vamos ver os principais métodos do `os`:

`os.arch()`

Retorna uma string que identifica a arquitetura, como `arm`, `x64`, `arm64`.

`os.cpus()`

Retorna informações sobre as CPUs disponíveis no seu sistema.

Exemplo:

```
[  
... {  
...   model: 'Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz',  
...   speed: 2400,  
...   times: {  
...     user: 281685380,  
...     nice: 0,  
...     sys: 187986530,  
...     idle: 685833750,  
...     irq: 0  
...   }  
... },  
... ]
```

```
  {
    model: 'Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz',
    speed: 2400,
    times: {
      user: 282348700,
      nice: 0,
      sys: 161800480,
      idle: 703509470,
      irq: 0
    }
  }
]
```

os.endianness()

Retorna `BE` ou `LE` dependendo se o Node.js foi compilado com [Big Endian ou Little Endian](#).

os.freemem()

Retorna o número de bytes que representam a memória livre no seu sistema.

os.homedir()

Retorna o caminho do diretório home do usuário corrente.

Exemplo:

```
'/Users/joe'
```

os.hostname()

Retorna o host name.

os.loadavg()

Retorna o cálculo feito pelo sistema operacional na média de carregamento.

Só retorna um valor significativo no Linux e no macOS.

Exemplo:

```
[3.68798828125, 4.00244140625, 11.1181640625]
```

os.networkInterfaces()

Retorna os detalhes das interfaces de rede disponíveis no seu sistema.

Exemplo:

```
{ lo0:
  [ { address: '127.0.0.1',
      netmask: '255.0.0.0',
      family: 'IPv4',
      mac: 'fe:82:00:00:00:00',
      internal: true },
    { address: '::1',
      netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
      family: 'IPv6',
      mac: 'fe:82:00:00:00:00',
      scopeid: 0,
```

```
        internal: true },
        { address: 'fe80::1',
          netmask: 'ffff:ffff:ffff:ffff::',
          family: 'IPv6',
          mac: 'fe:82:00:00:00:00',
          scopeid: 1,
          internal: true } ],
  en1:
  [ { address: 'fe82::9b:8282:d7e6:496e',
      netmask: 'ffff:ffff:ffff:ffff::',
      family: 'IPv6',
      mac: '06:00:00:02:0e:00',
      scopeid: 5,
      internal: false },
    { address: '192.168.1.38',
      netmask: '255.255.255.0',
      family: 'IPv4',
      mac: '06:00:00:02:0e:00',
      internal: false } ],
  utun0:
  [ { address: 'fe80::2513:72bc:f405:61d0',
      netmask: 'ffff:ffff:ffff:ffff::',
      family: 'IPv6',
      mac: 'fe:80:00:20:00:00',
      scopeid: 8,
      internal: false } ] }
```

os.platform()

Retorna a plataforma em que o Node.js foi compilado:

- darwin
- freebsd
- linux
- openbsd

- `win32`
- ...outros

`os.release()`

Retorna uma string que identifica o número da versão do sistema operacional

`os.tmpdir()`

Retorna o caminho da pasta temp atribuida.

`os.totalmem()`

Retorna o número de bytes que representam o total de memória disponível no sistema.

`os.type()`

Identifica o sistema operacional:

- `Linux`
- `Darwin` no macOS
- `Windows_NT` no Windows

`os.uptime()`

Retorna o número de segundos em que o computador está rodando desde o último reinício.

os.userInfo()

Retorna um objeto contendo o `username` corrente, `uid`, `gid`, `shell`, e `homedir`

O módulo de eventos

O módulo `events` nos disponibiliza a classe `EventEmitter`, que é essencial para trabalhar com eventos no Node.js.

```
const EventEmitter = require('events')
const door = new EventEmitter()
```

O `event listener` usa esses eventos:

- `newListener` quando um `listener` é adicionado
- `removeListener` quando um `listener` é removido

Aqui detamos uma descrição detalhada dos métodos mais usados:

emitter.addListener()

Alias para `emitter.on()`.

emitter.emit()

Emite um evento. Chama sincronamente todo event listener na ordem em que foram registrados.

```
door.emit("slam") // emitindo o evento "slam"
```

emitter.eventNames()

Retorna um array de string que representa os eventos registrados no objeto `EventEmitter` atual:

```
door.eventNames()
```

emitter.getMaxListeners()

Obtêm a quantidade máxima de listeners que podem ser adicionados em um objeto `EventEmitter`, que por padrão é 10 mas pode ser aumentado ou diminuido utilizando `setMaxListeners()`

```
door.getMaxListeners()
```

emitter.listenerCount()

Obtêm a contagem total de listeners do evento passado como parâmetro:

```
door.listenerCount('open')
```

emitter.listeners()

Obtêm um array de listeners do evento passado como parâmetro:

```
door.listeners('open')
```

emitter.off()

Alias para `emitter.removeListener()` adicionado no Node.js 10

emitter.on()

Adiciona uma função callback que é chamada quando um evento é emitido.

Uso:

```
door.on('open', () => {
  console.log('Door was opened')
})
```

emitter.once()

Adiciona uma função callback que será chamada quando um evento for emitido pela primeira vez depois de registrado. Essa callback só será chamada uma vez.

```
const EventEmitter = require('events')
const ee = new EventEmitter()
```

```
ee.once('my-event', () => {
  // chama a função callback uma vez
})
```

emitter.prependListener()

Quando você adiciona um listener usando `on` ou `addListener`, ele é adicionado no fim da fila de listener, e chamado por último. Usando `prependListener` ele será adicionado, e chamado, antes dos outros listeners.

emitter.prependOnceListener()

Quando você adiciona um listener usando `once`, ele é adicionado na fila de listeners, e chamado por último. Usando `prependOnceListener` ele é adicionado, e chamado, antes dos outros listeners.

emitter.removeAllListeners()

Remove todos os listeners de um objeto `EventEmitter` escutando um evento em específico:

```
door.removeAllListeners('open')
```

emitter.removeListener()

Remove um listener em específico. Você pode fazer isso salvando a função callback em uma variável, quando adicionada, ela pode ser referenciada mais tarde:

```
const doSomething = () => {}
door.on('open', doSomething)
door.removeListener('open', doSomething)
```

emitter.setMaxListeners()

Define a quantidade máxima de listeners que podem ser adicionados em um objeto `EventEmitter`, que por padrão é 10 mas pode ser aumentado ou diminuido.

```
door.setMaxListeners(50)
```

Módulo HTTP

O módulo HTTP é um módulo essencial para aplicações de rede em Node.js.

Ele pode ser importado assim:

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

O módulo disponibiliza algumas propriedades e métodos, e também algumas classes.

Properties

http.METHODS

Essa propriedade lista todos os métodos HTTP suportados:

```
> require('http').METHODS
[ 'ACL',
  'BIND',
  'CHECKOUT',
  'CONNECT',
  'COPY',
  'DELETE',
  'GET',
  'HEAD',
  'LINK',
  'LOCK',
  'M-SEARCH',
  'MERGE',
  'MKACTIVITY',
  'MKCALENDAR',
  'MKCOL',
  'MOVE',
  'NOTIFY',
  'OPTIONS',
  'PATCH',
  'POST',
  'PROPFIND',
  'PROPPATCH',
  'PURGE',
  'PUT',
  'REBIND',
  'REPORT',
  'SEARCH',
  'SUBSCRIBE',
  'TRACE',
  'UNBIND',
  'UNLINK',
  'UNLOCK',
  'UNSUBSCRIBE' ]
```

http.STATUS_CODES

Essa propriedade lista todos os códigos de status HTTP e suas descrições:

```
> require('http').STATUS_CODES
{
  '100': 'Continue',
  '101': 'Switching Protocols',
  '102': 'Processing',
  '200': 'OK',
  '201': 'Created',
  '202': 'Accepted',
  '203': 'Non-Authoritative Information',
  '204': 'No Content',
  '205': 'Reset Content',
  '206': 'Partial Content',
  '207': 'Multi-Status',
  '208': 'Already Reported',
  '226': 'IM Used',
  '300': 'Multiple Choices',
  '301': 'Moved Permanently',
  '302': 'Found',
  '303': 'See Other',
  '304': 'Not Modified',
  '305': 'Use Proxy',
  '307': 'Temporary Redirect',
  '308': 'Permanent Redirect',
  '400': 'Bad Request',
  '401': 'Unauthorized',
  '402': 'Payment Required',
  '403': 'Forbidden',
  '404': 'Not Found',
  '405': 'Method Not Allowed',
  '406': 'Not Acceptable',
  '407': 'Proxy Authentication Required',
  '408': 'Request Timeout',
  '409': 'Conflict',
  '410': 'Gone',
  '411': 'Length Required',
  '412': 'Precondition Failed',
  '413': 'Payload Too Large',
  '414': 'URI Too Long',
```

```
'415': 'Unsupported Media Type',
'416': 'Range Not Satisfiable',
'417': 'Expectation Failed',
'418': 'I\'m a teapot',
'421': 'Misdirected Request',
'422': 'Unprocessable Entity',
'423': 'Locked',
'424': 'Failed Dependency',
'425': 'Unordered Collection',
'426': 'Upgrade Required',
'428': 'Precondition Required',
'429': 'Too Many Requests',
'431': 'Request Header Fields Too Large',
'451': 'Unavailable For Legal Reasons',
'500': 'Internal Server Error',
'501': 'Not Implemented',
'502': 'Bad Gateway',
'503': 'Service Unavailable',
'504': 'Gateway Timeout',
'505': 'HTTP Version Not Supported',
'506': 'Variant Also Negotiates',
'507': 'Insufficient Storage',
'508': 'Loop Detected',
'509': 'Bandwidth Limit Exceeded',
'510': 'Not Extended',
'511': 'Network Authentication Required' }
```

http.globalAgent

Aponta para a instância global do objeto Agente, que é uma instância da classe `http.Agent`.

É usado para gerenciar persistência de conexões e reuso para clientes HTTP.

Veremos mais da descrição da classe `http.Agent`.

Methods

http.createServer()

Retorna uma nova instância da classe `http.Server`.

Uso:

```
const server = http.createServer((req, res) => {
  // gerencie cada requisição com essa callback
})
```

http.request()

Faz uma requisição HTTP para um servidor, criando uma instância da classe `http.ClientRequest`.

http.get()

Parecido com o `http.request()`, mas define automaticamente o verbo HTTP para GET, e chama `req.end()` automaticamente.

Classes

O módulo HTTP provê 5 classes:

- `http.Agent`
- `http.ClientRequest`

- `http.Server`
- `http.ServerResponse`
- `http.IncomingMessage`

http.Agent

O Node.js cria uma instância global da classe `http.Agent` para gerenciar persistência de conexões e reuso para clientes HTTP.

Esse objeto garante que toda requisição feita ao servidor seja enfileirada e um único socket seja reusado.

Também mantém um pool de sockets. Isso é necessário por razões de performance.

http.ClientRequest

Um objeto `http.ClientRequest` é criado quando `http.request()` ou `http.get()` são chamados.

Quando uma resposta é recebida, o evento `response` é chamado com a resposta, contendo uma instância de `http.IncomingMessage` como argumento.

Os dados retornados de uma resposta podem ser lidos de 2 formas:

- você pode chamar o método `response.read()`
- na callback do evento `response` você pode preparar um event listener para o evento `data`, assim você pode ouvir os dados transmitidos.

http.Server

Essa classe é comumente instanciada e retornada quando se cria um novo servidor usando `http.createServer()`.

Uma vez que você tenha um objeto server, você pode acessar os métodos dele:

- `close()` impede o servidor de aceitar novas conexões
- `listen()` inicia o servidor HTTP e escuta as conexões

http.ServerResponse

Criado por um `http.Server` e passado como segundo parâmetro para o evento `request` acionado.

Normalmente é usado com o `res`:

```
const server = http.createServer((req, res) => {
  // res é um objeto http.ServerResponse
})
```

Você sempre deve chamar o método `end()` na callback, que fecha a response, a mensagem está completa e o servidor pode enviá-la ao client. Deve ser chamado em cada response.

Esse métodos são usados para interagir com headers HTTP:

- `getHeaderNames()` obtém a lista de nomes das headers HTTP já definidas
- `getHeaders()` obtém uma cópia das headers HTTP já definidas
- `setHeader('headername', value)` define uma header HTTP e seu valor
- `getHeader('headername')` obtém uma header HTTP já definida

- `removeHeader('headername')` remove uma header HTTP já definida
- `hasHeader('headername')` retorna true se a response já contém aquela header
- `headersSent()` retorna true se as headers já foram enviadas ao client

Depois de processar as headers você pode enviá-las ao client chamando `response.writeHead()`, que aceita o `statusCode` como primeiro parâmetro, e mensagem do status de forma opcional, e o objeto de headers.

Para enviar os dados ao client pelo body da response, você usa `write()`. Ele enviará os dados armazenados em buffer para a stream de resposta HTTP.

Se os headers ainda não foram enviados usando `response.writeHead()`, as headers serão enviadas primeiro, com o código de status e a mensagem que foram definidos na request, que você pode editar definindo os valores das propriedades `statusCode` e `statusMessage`:

```
response.statusCode = 500  
response.statusMessage = 'Internal Server Error'
```

http.IncomingMessage

Um objeto `http.IncomingMessage` é criado quando:

- `http.Server` está escutando o evento `request`
- `http.ClientRequest` está escutando o evento `response`

Pode ser usado para acessar da response:

- status usando os métodos `statusCode` e `statusMessage`
- headers usando o método `headers` ou `rawHeaders`

- método HTTP usando o método `method`
- versão HTTP usando o método `httpVersion`
- URL usando o método `url`
- socket subjacente usando o método `socket`

Os dados são acessados usando streams, uma vez que `http.IncomingMessage` implementa a interface Readable Stream.

Buffers no Node.js

O que é um buffer?

Um buffer é uma área de memória. Desenvolvedores JavaScript não são familiarizados com esse conceito, muito menos que desenvolvedores C, C++ ou Go (ou qualquer programador que use uma linguagem de programação de sistema), que interagem com memória diariamente.

Ele representa um pedaço de memória de tamanho fixo (não pode ser redimensionado) alocado fora da engine JavaScript V8.

Você pode pensar em um buffer com um array de inteiros, em que cada um representa um byte de dados.

Ele é implementado pela [classe Buffer](#) no Node.js.

Por que precisamos de um buffer?

Buffers foram introduzidos para ajudar desenvolvedores a lidar com dados binários, em um ecossistema que tradicionalmente só lida com strings em vez de binários.

Buffers são profundamente linkados com streams. Quando um processador de stream recebe mais rápido do que pode aguentar, ele coloca os dados em um buffer.

Uma visualização simples de um buffer é quando você está assistindo um vídeo no Youtube e a linha vermelha vai além do seu ponto de visualização: você está baixando dados mais rápido do que está os vendo, e o seu browser os bufferiza.

Como criar um buffer

Um buffer é criado usando os métodos `Buffer.from()`, `Buffer.alloc()`, e `Buffer.allocUnsafe()`.

```
const buf = Buffer.from('Hey!')
```

- `Buffer.from(array)`
- `Buffer.from(arrayBuffer[, byteOffset[, length]])`
- `Buffer.from(buffer)`
- `Buffer.from(string[, encoding])`

Você também pode inicializar passando apenas o tamanho. Isso cria um buffer de 1KB:

```
const buf = Buffer.alloc(1024)
//or
const buf = Buffer.allocUnsafe(1024)
```

Enquanto ambos `alloc` e `allocUnsafe` alocam um `Buffer` do tamanho especificado em bytes, o `Buffer` criado via `alloc` vai ser *initializado* com zeros e o criado via `allocUnsafe` vai ser *não inicializado*. Isso significa que enquanto `allocUnsafe` pode ser bem rápido em comparação ao `alloc`, o segmento de

memória alocado pode conter dados antigos que podem ser potencialmente sensíveis.

Dados antigos, se presentes em memória, podem ser acessados ou vazados quando a memória do `Buffer` é lida. Isso é o que realmente faz do `allocUnsafe` inseguro e tenha cuidado extra ao usá-lo.

Usando um buffer

Acessando o conteúdo de um buffer

Um buffer, sendo um array de bytes, pode ser acessado como um array:

```
const buf = Buffer.from('Hey!')
console.log(buf[0]) // 72
console.log(buf[1]) // 101
console.log(buf[2]) // 121
```

Esses números são o Código Unicode que identifica o caracter naquela posição do buffer (H => 72, e => 101, y => 121)

Você pode imprimir o conteúdo completo de um buffer usando o método `toString()`:

```
console.log(buf.toString())
```

Observe que se você inicializar um buffer com um número que defina o tamanho dele, você vai ter acesso à memória pre-inicializada, que vai conter dados aleatórios, não um buffer vazio!

Obtendo o comprimento de um buffer

Use a propriedade `length`:

```
const buf = Buffer.from('Hey!')  
console.log(buf.length)
```

Iterando sob o conteúdo de um buffer

```
const buf = Buffer.from('Hey!')  
for (const item of buf) {  
  console.log(item) // 72 101 121 33  
}
```

Mudando o conteúdo de um buffer

Você pode gravar uma string inteira de dados em um buffer usando o método `write()`:

```
const buf = Buffer.alloc(4)  
buf.write('Hey!')
```

Assim como você pode acessar um buffer com uma sintaxe de array, você também pode definir os conteúdos do buffer dessa mesma forma:

```
const buf = Buffer.from('Hey!')  
buf[1] = 111 // o  
console.log(buf.toString()) // Hoy!
```

Copiando um buffer

É possível copiar um buffer usando o método `copy()`:

```
const buf = Buffer.from('Hey! ')
let bufcopy = Buffer.alloc(4) // allocate 4 bytes
buf.copy(bufcopy)
```

Por padrão você copia o buffer inteiro. Mais 3 parâmetros deixam você definir a posição inicial, a posição final, e o comprimento do novo buffer:

```
const buf = Buffer.from('Hey! ')
let bufcopy = Buffer.alloc(2) // allocate 2 bytes
buf.copy(bufcopy, 0, 0, 2)
bufcopy.toString() // 'He'
```

Fatiando um buffer

Se você quer criar uma visualização parcial de um buffer, você pode criar uma fatia (slice). Uma fatia não é uma cópia: o buffer original ainda é a fonte de verdade. Se ele mudar, sua fatia também muda.

Use o método `slice()` para criar a fatia. O primeiro parâmetro é a posição inicial, e você pode especificar um segundo parâmetro opcional com a posição final:

```
const buf = Buffer.from('Hey! ')
buf.slice(0).toString() // Hey!
const slice = buf.slice(0, 2)
console.log(slice.toString()) // He
buf[1] = 111 // o
console.log(slice.toString()) // Ho
```

Node.js Streams

O que são streams

Streams são um dos conceitos fundamentais que empoderam aplicações Node.js.

Elas são um jeito de lidar com leitura/escrita de arquivos, comunicações em rede, ou qualquer tipo de troca de informação end-to-end de uma forma eficiente.

Streams não são um conceito único do Node.js. Elas foram introduzidas no sistema operacional Unix décadas atrás, permitindo que programas possam interagir uns com os outros passando streams, através do operador pipe (|).

Por exemplo, em uma abordagem tradicional, quando você diz ao programa para ler um arquivo, o arquivo é lido na memória, do começo ao fim, e então é processado.

Utilizando streams você lê pedaço por pedaço, processando o conteúdo sem mantê-lo completo na memória.

O [módulo stream](#) provê a fundação da qual todas APIs de streaming são feitas. Todas streams são instâncias do [EventEmitter](#)

Porquê streams

Streams basicamente nos dão duas grandes vantagens sobre outros métodos de manipulação de dados:

- **Eficiência de memória:** você não precisa carregar grandes quantidades de dados em memória antes de ser capaz de processá-los
- **Eficiência temporal:** menos tempo é requerido para começar a processar dados, uma vez que você pode começar a processar assim que os tiver, em vez de esperar até que toda carga útil de dados esteja disponível

Um exemplo de stream

Um exemplo típico é ler arquivos de um disco.

Usando o módulo `fs` do Node.js, você pode ler um arquivo, e servi-lo com HTTP quando uma nova conexão é estabelecida no seu servidor HTTP:

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

const server = http.createServer(function(req, res) {
  fs.readFile(__dirname + '/data.txt', (err, data) => {
    res.end(data)
  })
})
server.listen(3000)
```

`readFile()` lê todo conteúdo do arquivo, e invoca uma função callback quando terminar.

`res.end(data)` na callbac vai retornar o conteúdo do arquivo para o client HTTP.

Se o arquivo for grande, a operação vai demorar um pouco. Aqui temos a mesma funcionalidade usando streams:

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

const server = http.createServer((req, res) => {
  const stream = fs.createReadStream(__dirname + '/data.txt')
  stream.pipe(res)
})
server.listen(3000)
```

Em vez de esperar até que o arquivo seja completamente lido, nós começamos a “streamá-lo” para o client HTTP assim que tivermos chunks de dados prontos para serem enviados.

pipe()

O exemplo acima usa `stream.pipe(res)`: o método `pipe()` é chamado na stream de arquivo.

O que esse código faz? Ele pega a origem, e canaliza para um destino.

Você chama isso na stream origem, e nesse caso, a stream de arquivo é canalizada para a resposta HTTP.

O valor retornado pelo método `pipe()` é a stream de destino, que é algo muito conveniente pois nos permite encadear múltiplas chamadas `pipe()`, desse jeito:

```
src.pipe(dest1).pipe(dest2)
```

Isso faz o mesmo que

```
src.pipe(dest1)
dest1.pipe(dest2)
```

APIs Node.js baseadas em streams

Dadas as suas vantagens, muitos dos módulos principais do Node.js fornecem recursos nativos para manipulação de streams, sendo os mais notáveis:

- `process.stdin` retorna uma stream conectada ao stdin

- `process.stdout` retorna uma stream conectada ao stdout
- `process.stderr` retorna uma stream conectada ao stderr
- `fs.createReadStream()` cria uma stream de leitura para um arquivo
- `fs.createWriteStream()` cria uma stream de escrita para um arquivo
- `net.connect()` inicia uma stream baseada em conexão
- `http.request()` retorna uma instância da classe `http.ClientRequest`, que é uma stream de escrita
- `zlib.createGzip()` comprime dados usando gzip (um algoritmo de compressão) em uma stream
- `zlib.createGunzip()` descomprime uma stream gzip.
- `zlib.createDeflate()` comprime dados usando deflate (um algoritmo de compressão) em uma stream
- `zlib.createInflate()` descomprime uma stream deflate.

Diferentes tipos de streams

Existem quatro classes de streams:

- **Readable**: uma stream de onde você pode canalizar, mas não canalizar dados nela (você pode receber dados, mas não pode enviar dados). Quando você coloca dados em uma stream de leitura, os dados são bufferizados, até que um consumer comece a ler os dados.
- **Writable**: uma stream em que você pode canalizar dados, mas não obter dados dela (você pode enviar dados, mas não pode receber).
- **Duplex**: uma stream em que você pode tanto canalizar dados quanto obtê-los, basicamente uma combinação de uma stream Readable e uma Writable

- **Transform**: uma stream Transform é similar a uma Duplex, mas a saída é um transform das entradas

Como criar uma stream de leitura (readable stream)

Nós obtêmos a stream de leitura pelo [módulo stream](#), a inicializamos e implementamos o método `readable._read()`.

Primeiro crie um objeto stream:

```
const Stream = require('stream')
const readableStream = new Stream.Readable()
```

então implemente o `_read`:

```
readableStream._read = () => {}
```

Você também pode implementar o `_read` usando a opção `read`:

```
const readableStream = new Stream.Readable({
  read() {}
})
```

Agora que a stream está inicializada, nós podemos enviar dados a ela:

```
readableStream.push('hi!')
readableStream.push('ho!')
```

Como criar uma stream de escrita (writable stream)

Para criar uma stream de escrita nós extendemos o objeto base `Writable`, e implementamos seu método `_write()`.

Primeiro crie um objeto stream:

```
const Stream = require('stream')
const writableStream = new Stream.Writable()
```

e então implemente o `_write`:

```
writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString())
  next()
}
```

Agora você pode canalizar uma stream de leitura nele:

```
process.stdin.pipe(writableStream)
```

Como obtêr dados de uma stream de leitura

Como nós lemos dados de uma stream de leitura? Usando uma stream de escrita:

```
const Stream = require('stream')

const readableStream = new Stream.Readable({
  read() {}
})
const writableStream = new Stream.Writable()

writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString())
```

```
    next()
}

readableStream.pipe(writableStream)

readableStream.push('hi!')
readableStream.push('ho!')
```

Você também pode consumir uma stream de leitura diretamente, usando o evento `readable`:

```
readableStream.on('readable', () => {
  console.log(readableStream.read())
})
```

Como enviar dados para uma stream de escrita

Usando o método `write()` da stream:

```
writableStream.write('hey!\n')
```

Sinalizando a uma stream de escrita que você terminou de escrever

Use o método `end()`:

```
const Stream = require('stream')

const readableStream = new Stream.Readable({
  read() {}
})
```

```
const writableStream = new Stream.Writable()

writableStream._write = (chunk, encoding, next) => {
  console.log(chunk.toString())
  next()
}

readableStream.pipe(writableStream)

readableStream.push('hi!')
readableStream.push('ho!')

writableStream.end()
```

A diferença entre development e production

Você pode ter configurações diferentes para os ambientes de produção e desenvolvimento.

O Node.js assume que você está sempre rodando de um ambiente de desenvolvimento.

Você pode sinalizar ao Node.js que está rodando em produção definindo a variável de ambiente `NODE_ENV=production`.

Isso normalmente é feito executando o comando

```
export NODE_ENV=production
```

no shell, mas é melhor colocar no seu arquivo de configuração shell (`bash_profile` com Bash shell), porque senão a configuração não é persistida em caso de reinício do sistema.

Você também pode aplicar a variável de ambiente prefixando no comando de inicialização da sua aplicação:

```
NODE_ENV=production node app.js
```

Essa variável de ambiente é uma convenção amplamente usada por bibliotecas também.

Definir o ambiente para `production` geralmente garante que

- é mantido o mínimo de logs, apenas o essencial
- mais níveis de cache são usados para otimizar a performance

Tomando como exemplo o Pug, a biblioteca de templates usada pelo Express, compila em modo debug se `NODE_ENV` não estiver definida como `production`. No ambiente de desenvolvimento as views do Express são compiladas em cada requisição, enquanto que em produção elas são cacheadas. Há muitos outros exemplos além desse.

Você pode usar condicionais para executar códigos em ambientes diferentes:

```
if (process.env.NODE_ENV === "development") {  
  //...  
}  
if (process.env.NODE_ENV === "production") {  
  //...  
}  
if(['production', 'staging'].indexOf(process.env.NODE_ENV) >= 0) {  
  //...  
})
```

Por exemplo, em uma aplicação Express, você pode usar isso para definir o `errorHandler` de acordo com o ambiente:

```
if (process.env.NODE_ENV === "development") {  
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }))  
  
if (process.env.NODE_ENV === "production") {  
  app.use(express.errorHandler())  
})
```

Error handling no Node.js

Erros no Node.js são tratados por meio de exceções.

Criando exceções

Uma exceção é criada usando a palavra reservada `throw`:

```
throw value
```

Assim que o JavaScript executa essa linha, o fluxo normal do programa é parado e o controle é retomado pelo **tratamento de exceção** mais próximo.

Normalmente no código client-side o `value` pode ser qualquer valor JavaScript, incluindo uma string, um número ou um objeto.

No Node.js, nós não lançamos strings, só lançamos objetos de erro.

Objetos de erro

Um objeto de erro é um objeto que é uma instância do objeto `Error`, ou que estenda a classe `Error`, fornecida pelo [módulo nativo `Error`](#):

```
throw new Error('Ran out of coffee')
```

ou

```
class NotEnoughCoffeeError extends Error {  
    // ...  
}  
throw new NotEnoughCoffeeError()
```

Tratando exceções

Um bloco `try/catch` é um tratador de exceções.

Qualquer exceção lançada nas linhas que incluem o bloco `try` serão tratadas no bloco `catch` correspondente:

```
try {  
    // linhas de código  
} catch (e) {}
```

`e` nesse exemplo é o valor da exceção.

Você pode adicionar múltiplos blocos, que podem tratar diferentes tipos de erros.

Pegando exceções não capturadas

Se uma exceção não capturada for lançada durante a execução do seu programa, ele irá crashar.

Para resolver isso, escute o evento `uncaughtException` no objeto `process`:

```
process.on('uncaughtException', err => {
  .console.error('There was an uncaught error', err)
  .process.exit(1) // obrigatório (conforme as docs do Node.js)
})
```

Você não precisa importar o módulo nativo `process`, ele é automaticamente injetado.

Exceções com promises

Usando promises você pode encadear diferentes operações, e tratar erros no final:

```
doSomething1()
  .then(doSomething2)
  .then(doSomething3)
  .catch(err => console.error(err))
```

Como você sabe quando o erro ocorreu? Você na verdade não sabe, mas você pode tratar erros em cada uma das funções que chamar (`doSomethingX`), e dentro do catch lançar um novo erro, que chamará o `catch` exterior:

```
const doSomething1 = () => {
  //...
  try {
    //...
  } catch (err) {
    //... trata o erro localmente
    throw new Error(err.message)
  }
  //...
}
```

Para ser possível tratar erros localmente sem tratá-los na função que chamamos, nós podemos quebrar a corrente, você pode criar uma função em

cada `then` e processar a exceção:

```
doSomething1()
  .then(() => {
    return doSomething2().catch(err => {
      // trata o erro
      throw err // quebra a corrente!
    })
  })
  .then(() => {
    return doSomething2().catch(err => {
      // trata o erro
      throw err // quebra a corrente!
    })
  })
  .catch(err => console.error(err))
```

Tratando erros com `async/await`

Usando `async/await`, você ainda precisa tratar erros, e você pode fazer isso dessa forma:

```
async function someFunction() {
  try {
    await someOtherFunction()
  } catch (err) {
    console.error(err.message)
  }
}
```

Como fazer log de um objeto no Node.js

Quando você digita `console.log()` em um programa JavaScript e o roda no browser, isso vai criar uma entrada bem legal no Console do Browser:

```
> console.log(obj)
▶ {name: "Flavio", age: 35}
```

Ao clicar na flecha, o log é expandido, e você pode ver claramente as propriedades do objeto:

```
> console.log(obj)
▼ {name: "Flavio", age: 35} ⓘ
  age: 35
  name: "Flavio"
▶ __proto__: Object
```

No Node.js, o mesmo acontece.

Nós não temos tanto luxo quando logamos algo no console, porque isso vai retornar o objeto para o shell se você rodar o programa Node.js manualmente, ou para um arquivo de log. Você obtém uma representação do objeto em formato de string.

Agora, tudo fica bem até um certo nível de aninhamento. Depois de dois níveis de aninhamento, o Node.js desiste e imprime `[Object]` como um placeholder:

```
const obj = {
  name: 'joe',
  age: 35,
  person1: {
```

```
    name: 'Tony',
    age: 50,
    person2: {
      name: 'Albert',
      age: 21,
      person3: {
        name: 'Peter',
        age: 23
      }
    }
}
console.log(obj)
```

```
{
  name: 'joe',
  age: 35,
  person1: {
    name: 'Tony',
    age: 50,
    person2: {
      name: 'Albert',
      age: 21,
      person3: [Object]
    }
  }
}
```

Como você pode imprimir o objeto inteiro?

A melhor maneira de fazer isso, preservando a beleza da saída, é usar:

```
console.log(JSON.stringify(obj, null, 2))
```

sendo **2** o número de espaços usados para identação.

Outra opção é usar

```
require('util').inspect.defaultOptions.depth = null  
console.log(obj)
```

mas o problema é que objetos aninhados com mais de 2 níveis ficam planificados, e isso pode ser um problema com objetos complexos.

Referências

Revisado, Editado e baseado na documentação oficial do NodeJS, da qual possui licença MIT, que permite a livre redistribuição.

Agradecimento especial ao meu estagiário [Matheus Rocha](#) que trabalhou duro para esta apostila sair!

Meu objetivo é levar conhecimento e acessibilidade a todos, muito obrigado por ter lido até aqui.

Se tiver qualquer dúvida me pergunte qualquer coisa no [Instagram](#), faço mentoria coletiva (gratuita) para programadores iniciantes que se sentem perdidos, você será extremamente bem vindo por lá (é só clicar na imagem).

**QUER UMA
DIREÇÃO
NA
CARREIRA
COMO
<DEV/>?**

Clique aqui.

A small white icon of a hand with fingers pointing upwards is located in the top-left corner of the ad area.

Um grande abraço, Paulo Luan.



Se você quiser dar o proximo passo para ser um dev profissional, eu posso te mentorar! [é só você clicar aqui!](#)

[← Como estudar Ciências da Computação \(sem vestibular e de graça\)](#)

[Como fazer perguntas sobre programação na internet? \(e aumentar 5x as](#)

chances de ter uma resposta) →

© 2021, Sua melhor escolha para aprender programação: [Reativa Tecnologia](#)