## Você será capaz de:

- Utilizar os comandos do `npm` para criar e gerenciar pacotes e dependências;
- Utilizar o comando `node` para executar scripts;
- Utilizar o sistema de módulos do Node.js;
- Criar scripts simples, com interação do usuário, com Node.js.

## Por que isso é importante?

O conhecimento de Node.js enquanto ferramenta tem sido cada vez mais valorizado. A cada dia, mais empresas vem utilizando essa tecnologia para desenvolver suas aplicações Back-End. Como resultado, a demanda por pessoas capacitadas tem aumentado.
Não é à toa que a adoção do Node.js como tecnologia Back-end vem crescendo. O JavaScript, linguagem utilizada pela ferramenta, é extremamente versátil, simples e poderoso. A linguagem permite uma curva de aprendizado relativamente menor que outras grandes linguagens de propósito semelhante disponíveis no mercado e, ao mesmo tempo, agrega grande valor a quem a utiliza.
Sendo assim, o conhecimento da correta utilização do Node.js torna-se de extrema importância para pessoas que buscam fazer parte do mercado do desenvolvimento web moderno.
Vem bastante informação pela frente, mas não se preocupe! Conforme for avançando no conteúdo e aplicando os conceitos na prática, tudo vai fazer mais sentido.

---

## O que é Node.js?

Como dito anteriormente, o Node.js surgiu do V8, que é a ferramenta do Google Chrome responsável por ler e executar as instruções que escrevemos em JavaScript, processo comumente chamado de *interpretar* o código. Ao software responsável por interpretar o código dá-se o nome de interpretador , engine e, por vezes, de runtime . Por isso, é comum dizer que o NodeJS é um *runtime* JavaScript.
Apesar de ser baseado no V8, o Node.js possui algumas diferenças em relação ao interpretador que funciona nos navegadores. Dentre elas, as principais são a ausência de métodos para manipulação de páginas web e a presença de métodos que permitem acessar o sistema de arquivos e a rede mais diretamente.

## Por que usar Node.js

### Ele está por toda a parte

Como dito anteriormente, o uso do Node.js pelo mercado de tecnologia vem crescendo muito nos últimos anos. Dados do site ModuleCounts.com mostram que, atualmente, o NPM, que é onde os pacotes Node.js são disponibilizados, é o repositório com mais pacotes quando comparado a repositórios de outras grandes linguagens, como mostra o gráfico abaixo:

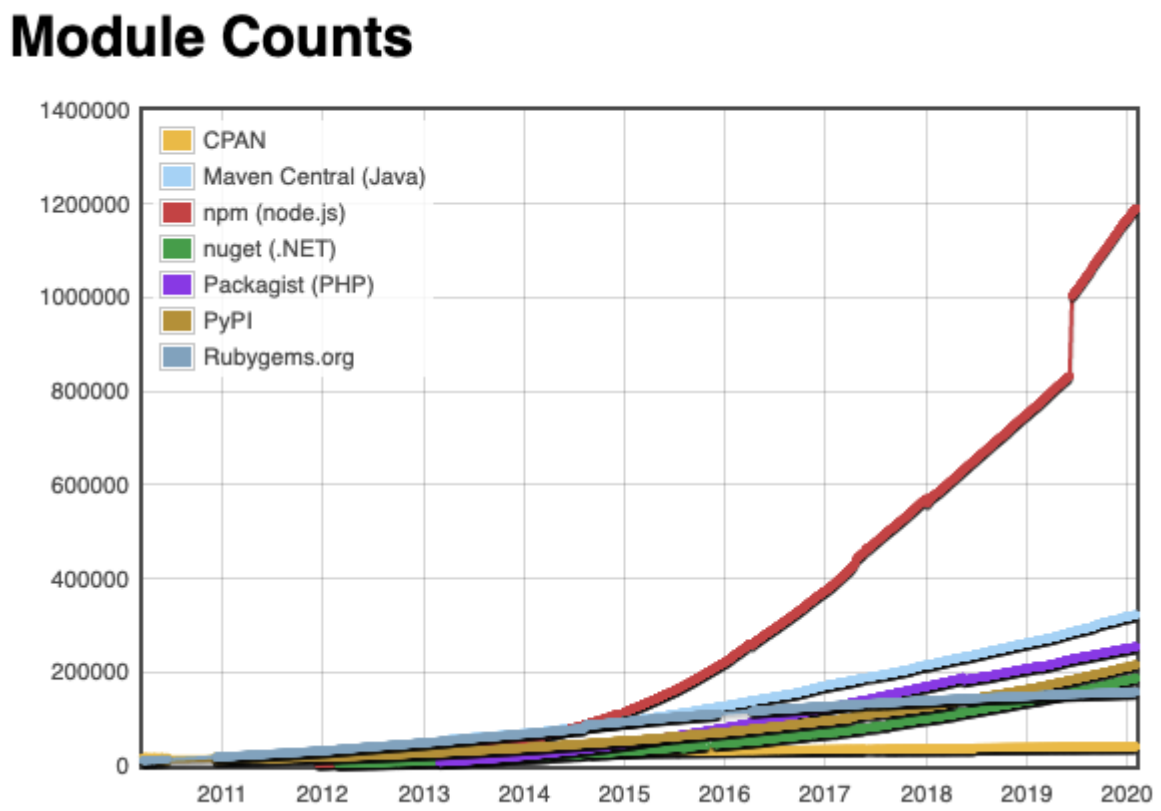


Gráfico extraído no dia 2 de fevereiro de 2020, mostrando a quantidade de pacotes em vários repositórios.

Ter muitos pacotes publicados no repositório atual nos dá uma ideia de duas grandes vantagens que o Node.js tem sobre tecnologias concorrentes: uma comunidade extremamente ativa e uma grande quantidade de ferramentas para resolver os mais diversos tipos de problema.

#### Performance

Quando comparado a outras grandes tecnologias, o Node.js nos permite escrever softwares servidores de requisições HTTP de forma muito mais eficiente. Isso se dá pelo fato de que toda leitura e escrita que ele realiza,

tanto no disco quanto na rede, é feita de forma não bloqueante . Ou seja, quando o servidor recebe uma requisição e precisa, por exemplo, buscar dados no banco de dados, as demais requisições não precisam esperar que a primeira termine para que possam ser atendidas. Em outras palavras, o NodeJS realiza todas as suas operações de entrada e saída de dados de forma assíncrona , utilizando processamento concorrente (veremos mais sobre fluxo assíncrono nos próximos dias).
Por serem mais eficientes e otimizadas que outras tecnologias, as aplicações feitas em Node.js acabam por consumir menos recursos dos servidores que as executam, tornando o Node.js uma tecnologia, em geral, mais barata que suas concorrentes.

### Aplicações em tempo real

A natureza não bloqueante do Node.js permite que alguns recursos sejam implementados na plataforma para facilitar o trabalho com operações em tempo real.
Bibliotecas como o socket.io permitem que, com poucas linhas de código, aplicações em tempo real relativamente complexas (como chats com suporte a múltiplos usuários, conversas privadas e em grupo e outros recursos) sejam criadas por completo.
Num mundo em que a tecnologia está cada vez mais inserida em diversas áreas, ter suporte nativo da plataforma que utilizamos para aplicações em tempo real com certeza é muito bem-vindo.

### JavaScript

Muitas das vantagens do Node.js vêm do fato de que a linguagem executada por ele é o JavaScript.
Como já falamos antes, o JavaScript tem se mostrado uma linguagem extremamente versátil, estando presente em diversos ambientes, como a Web, Desktop, Mobile, dispositivos IoT (internet das coisas) e até mesmo em aparelhos televisores!
A versatilidade e baixa curva de aprendizado do JavaScript conferem ao Node um poder incrível para resolver as mais diversas situações.

### Então Node.js é a melhor tecnologia para tudo?

Você acabou de ver alguns dos motivos pelos quais o Node.js é a ferramenta ideal para vários tipos de projeto. No entanto, é importante lembrar que não existe *bala de prata* quando o assunto é tecnologia: a

melhor ferramenta sempre depende do caso de uso e dos recursos disponíveis para desempenhar uma determinada tarefa.

## Sistema de módulos

Já citamos acima que uma das maiores vantagens do Node.js é a grande quantidade de pacotes e bibliotecas disponíveis publicamente no `npm` ; agora chegou a hora de aprendermos o que é um pacote Node e como podemos utilizá-los no nosso código.
Pra começar, vamos entender o que é um módulo em Node.js: um módulo é um "pedaço de código" que pode ser organizado em um ou mais arquivos, e que possui escopo próprio, ou seja: suas variáveis, funções, classes e afins só estão disponíveis nos arquivos que fazem parte daquele módulo. Em outras palavras, um módulo é uma funcionalidade ou um conjunto delas que se encontram isoladas do restante do código.
O Node.js possui três tipos de módulos: internos, locais e de terceiros.

### Módulos internos

Os módulos internos (ou *core modules* ) são inclusos no Node.js e instalados junto com ele quando baixamos o *runtime* . Alguns exemplos de *core modules* são:
Clique no nome de cada módulo para abrir a documentação oficial (em inglês).

- `fs` : Fornece uma API para interagir com o sistema de arquivos de forma geral;
- `url` : Provê utilitários para ler e manipular URLs;
- `querystring` : Disponibiliza ferramentas para leitura e manipulação de parâmetros de URLs;
- `util` : Oferece ferramentas e funcionalidades comumente úteis a pessoas programadoras.

### Módulos locais

Módulos locais são aqueles definidos juntamente à nossa aplicação. Representam funcionalidades ou partes do nosso programa que foram separados em arquivos diferentes.
Módulos locais podem ser publicados no NPM para que outras pessoas possam baixá-los e utilizá-los, o que nos leva ao nosso próximo e último tipo de módulo.

Módulos de terceiros

Módulos de terceiros são aqueles criados por outras pessoas e disponibilizados para uso através do `npm` . Conforme mencionado, nós também podemos criar e publicar nossos próprios módulos, quer seja para utilizarmos em diversas aplicações diferentes, quer seja para permitir que outras pessoas os utilizem. Veremos como fazer isso ainda hoje.
Nós também podemos criar e publicar módulos, seja para nós mesmos utilizarmos em diversas aplicações diferentes, ou para permitir que outras pessoas os utilizem em suas respectivas aplicações. Veremos como fazer isso ainda hoje.
Ufa! São alguns conceitos diferentes pra se acostumar, né? Mas tudo bem, você está indo bem. Resumindo o que acabamos de ver sobre módulos no Node.js: módulos são "caixinhas" que isolam suas funções, variáveis e afins de outras partes do código; eles podem ser internos (que vêm com o Node.js), locais (criados por nós na nossa máquina) ou de terceiros (baixados do NPM). Além disso, o NPM é o site em que publicamos nossos pacotes, e `npm` é a ferramenta de linha de comando que realiza o gerenciamento desses pacotes pra nós.

## Importando e exportando módulos

Quando queremos utilizar o conteúdo de um módulo ou pacote de outro arquivo no Node.js, precisamos importar esse módulo / pacote para o contexto atual no qual estamos.
Existem dois sistemas de módulos difundidos na comunidade JavaScript:

- Módulos ES6 ;
- Módulos CommonJS .

ES6

O nome ES6 vem de ECMAScript 6, que é a especificação seguida pelo JavaScript.
Na especificação do ECMAScript 6, os módulos são importados utilizando a palavra-chave `import` , tendo como contrapartida a palavra-chave `export` para exportá-los.
O Node.js não possui suporte a módulos ES6 por padrão, sendo necessário o uso de transpiladores, como o Babel, ou supersets da linguagem, como o TypeScript, para que esse recurso esteja disponível.

Transpiladores são ferramentas que leêm o código-fonte escrito em uma linguagem como o Node.js e produz o código equivalente em outra linguagem. Supersets são linguagens que utilizam um transpilador para adicionar novas funcionalidades ao JavaScript.
Para saber mais sobre módulos ES6 e transpiladores, dê uma olhada na seção Recursos Adicionais.

### CommonJS

O CommonJS é o sistema de módulos implementado pelo Node.js nativamente e, portanto, o sistema que utilizaremos daqui pra frente. Veja as próximas seções para entender como ele funciona
Vamos dar uma olhada, primeiramente, em como exportamos algo de um módulo ou arquivo JavaScript.

### Exportando módulos

Para exportar algo no sistema CommonJS, utilizamos a variável global `module.exports` , atribuindo a ela o valor que desejamos exportar:

Copiar

```
// brlValue.js
const brl = 5.37;

module.exports = brl;
```

Note como utilizamos as palavras-chave `module.exports` . Como vimos anteriormente, um módulo possui um escopo isolado, ou seja, suas funções, variáveis, classes e demais definições existem somente dentro do módulo. O `module.exports` nos permite definir quais desses "objetos" internos ao módulo serão exportados , ou seja, serão acessíveis a arquivos que importarem aquele módulo. O `module.exports` pode receber qualquer valor válido em JavaScript, incluindo objetos, classes, funções e afins.
No arquivo acima estamos exportando do nosso módulo o valor da constante `brl` , que é `5.37` . Ao importarmos esse módulo, receberíamos o valor `5.37` como resposta. Apesar de isso funcionar, exportar um único valor constante assim não é comum. Vamos observar um caso que acontece com mais frequência:

Copiar

```
// brlValue.js
const brl = 5.37;
```

```
const usdToBrl = (valueInUsd) => valueInUsd * brl;

module.exports = usdToBrl;
```

Agora estamos exportando uma função de forma que podemos utilizá-la para converter um valor em dólares para outro valor, em reais.
O uso desse nosso módulo se daria da seguinte forma:

Copiar

```
// index.js
const convert = require('./brlValue');

const usd = 10;
const brl = convert(usd);

console.log(brl) // 53.7
```

Perceba que podemos dar o nome que quisermos para a função depois que a importamos, independente de qual o seu nome dentro do módulo.

---

Suponhamos agora que seja desejável exportar tanto a função de conversão quanto o valor do dólar (a variável `brl` ). Para isso, podemos exportar um objeto contendo as duas constantes da seguinte forma:

Copiar

```
// brlValue.js
const brl = 5.37;

const usdToBrl = (valueInUsd) => valueInUsd * brl;

module.exports = {
  brl,
  usdToBrl,
};
```

Dessa forma, ao importarmos o módulo, receberemos um objeto como resposta:

Copiar

```
// index.js
const brlValue = require('./brValue');

console.log(brlValue); // { brl: 5.37, usdToBrl: [Function:
usdToBrl] }

console.log(`Valor do dólar: ${brlValue.brl}`); // Valor do dólar:
5.37
```

```
console.log(`10 dólares em reais: ${brlValue.usdToBrl(10)}`); // 10 dólares em reais: 53.7
```

Por último, como estamos lidando com um objeto, podemos utilizar *object destructuring* para transformar as propriedades do objeto importado em constantes no escopo global:

Copiar

```
const { brl, usdToBrl } = require('./brValue');

console.log(`Valor do dólar: ${brl}`); // Valor do dólar: 5.37
console.log(`10 dólares em reais: ${usdToBrl(10)}`); // 10 dólares em reais: 53.7
```

---

Agora que você já viu como exportar valores de módulos, vamos mergulhar em como podemos importar módulos no Node.js:

### Importando módulos

Você verá, a seguir, como utilizar o `require` para importar cada tipo de módulo.

#### Módulos locais

Quando queremos importar um módulo local, precisamos passar para o `require` o caminho do módulo, seguindo a mesma assinatura. Por exemplo, `require('./meuModulo')` . Note que a extensão ( `.js` ) não é necessária: por padrão, o Node já procura por arquivos terminados em `.js` ou `.json` e os considera como módulos.
Além de importarmos um arquivo como módulo, podemos importar uma pasta. Isso é útil, pois muitas vezes um módulo está dividido em vários arquivos, mas desejamos importar todas as suas funcionalidades de uma vez só. Nesse caso, a pasta precisa conter um arquivo chamado `index.js` , que importa cada um dos arquivos do módulo e os exporta da forma mais conveniente.
Por exemplo:

Copiar

```
// meuModulo/funcionalidade-1.js

module.exports = function () {
  console.log('funcionalidade1');
}
```

Copiar

```
// meuModulo/funcionalidade-2.js

module.exports = function () {
  console.log('funcionalidade2');
}
```

Copiar

```
// meuModulo/index.js
const funcionalidade1 = require('./funcionalidade-1');
const funcionalidade2 = require('./funcionalidade-2');

module.exports = { funcionalidade1, funcionalidade2 };
```

Note que utilizamos a palavras-chave `module.exports` . Confome já vimos, um módulo possui um escopo isolado, ou seja, suas funções, variáveis, classes e demais definições existem somente dentro do módulo. O `module.exports` nos permite definir quais desses "objetos" internos ao módulo serão exportados , ou seja, estarão acessíveis para arquivos que importarem aquele módulo. O `module.exports` pode receber qualquer valor válido em JavaScript, incluindo objetos, classes, funções e afins. No exemplo acima, isso quer dizer que, quando importarmos o módulo `meuModulo` , teremos um objeto contendo duas propriedades, que são as funcionalidades que exportamos dentro de `meuModulo/index.js` .
Para importarmos e utilizarmos o módulo `meuModulo` , basta passar o caminho da pasta como argumento para a função `require` , assim:

Copiar

```
// minha-aplicacao/index.js
const meuModulo = require('./meuModulo');1

console.log(meuModulo); // { funcionalidade1: [Function:
funcionalidade1], funcionalidade2: [Function: funcionalidade2] }

meuModulo.funcionalidade1();
```

### Módulos internos

Para utilizarmos um módulo interno, devemos passar o nome do pacote como parâmetro para a função `require` . Por exemplo, `require('fs')` importa o pacote `fs` , responsável pelo sistema de arquivos.
Uma vez que importamos um pacote, podemos utilizá-lo no nosso código como uma variável, dessa forma:

Copiar

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

fs.readFileSync('./meuArquivo.txt');
```

Repare que o nome da variável pode ser qualquer um que escolhermos. O que importa mesmo é o nome do pacote que passamos como parâmetro para o `require` .

### Módulos de terceiros

Módulos de terceiros são importados da mesma forma que os módulos internos: passando seu nome como parâmetro para a função `require` . A diferença é que, como esses módulos não são nativos do Node.js, precisamos primeiro instalá-los na pasta do projeto em que queremos utilizá-los. O registro oficial do Node.js, em que milhares de pacotes estão disponíveis para serem instalados, é o `npm` . Além disso, `npm` também é o nome da ferramenta de linha de comando (CLI - *command line interface* ) responsável por baixar e instalar esses pacotes. O CLI `npm` é instalado juntamente com o Node.js.
Quando importamos um módulo que não é nativo do Node.js, e não aponta para um arquivo local, o Node inicia uma busca por esse módulo. Essa busca acontece na pasta `node_modules` . Caso um módulo não seja encontrado na `node_modules` mais próxima do arquivo que o chamou, o Node procurará por uma pasta `node_modules` na pasta que contém a pasta atual. Caso encontrado, o módulo é carregado. Do contrário, o processo é repetido em um nível de pasta acima. Isso acontece até que o módulo seja encontrado, ou até que uma pasta `node_modules` não exista no local em que o Node está procurando.
Aproveitando que estamos falando sobre módulos de terceiros, vamos falar melhor do NPM: você entenderá melhor o que ele é e como utilizar os principais comandos do seu CLI. Bora lá!

## NPM

O NPM (sigla para *Node Package Manager* ) é, como dito no tópico anterior, o repositório oficial para publicação de pacotes NodeJS. É para ele que realizamos upload dos arquivos de nosso pacote quando queremos disponibilizá-lo para uso de outras pessoas ou em diversos projetos. Atualmente, uma média de 659 pacotes são publicados no NPM todos os dias, segundo o site ModuleCounts.com

Um pacote é um conjunto de arquivos que exportam um ou mais módulos Node. Nem todo pacote Node é uma biblioteca, visto que uma API desenvolvida em Node também tem um pacote.
Você entenderá mais sobre o que compõe um pacote mais à frente.

### Utilizando o CLI `npm`

O CLI do `npm` é uma ferramenta oficial que nos auxilia no gerenciamento de pacotes, sejam eles dependências da nossa aplicação ou nossos próprios pacotes. É através dele que criamos um projeto, instalamos e removemos pacotes, publicamos e gerenciamos versões dos nossos próprios pacotes. Publicar um pacote público no `npm` é gratuito e não há um limite de pacotes que se pode publicar. Existem, no entanto, taxas de assinaturas, caso desejemos hospedar pacotes de forma privada, ou seja, pacotes sob os quais só nós temos o controle de acesso.
Para entender melhor o `npm` e seu uso na prática, assista ao vídeo a seguir:
Como você pode ver no vídeo, o `npm` nos provê alguns comandos importantes, vários dos quais já temos usado há bastante tempo no curso! Você terá acesso ao *Cheat Sheet* dos comandos neste repositório para consultas rápidas. No entanto, vamos passar por alguns deles para uma explicação mais aprofundada:

### npm init

O comando `npm init` nos permite criar, de forma rápida e simplificada, um novo pacote Node.js na pasta onde é executado.
Ao ser executado, o comando pede para quem executou algumas informaçãoes sobre o pacote como nome, versão, nome das pessoas autoras e afins. Caso desejemos utilizar as respostas padrão para todas essas perguntas, podemos utilizar o comando com a flag `-y` , ou seja, `npm init -y` ; dessa forma, nenhuma pergunta será feita, e a inicialização será realizada com os valores padrão.
Para criar um novo pacote Node.js, o `npm init` simplesmente cria um arquivo chamado `package.json` com as respostas das perguntas realizadas e mais alguns campos de metadados. Para o Node.js, qualquer pasta contendo um arquivo `package.json` válido é um pacote.
Dentro do `package.json` é onde podemos realizar algumas configurações importantes para o nosso pacote como nome, versão, dependências e scripts .
Falando em scripts, vejamos um pouco mais sobre eles:

### npm run

O comando `run` faz com que o `npm` execute um determinado script configurado no `package.json` . Scripts são "atalhos" que podemos definir para executar determinadas tarefas relacionadas ao pacote atual.
Para criar um script, precisamos alterar o `package.json` e adicionar uma nova chave ao objeto `scripts` . O valor dessa chave deve ser o comando que desejamos que seja executado quando chamarmos aquele script.
Por exemplo, para ter um script chamado `lint` que execute a ferramenta de linter que usamos aqui na Trybe, o ESLint, nossa chave `scripts` fica assim:

Copiar

```
{
  "scripts": {
    "lint": "eslint ."
  }
}
```

Perceba que `lint` é o nome do script que digitamos no terminal para executar o ESLint na pasta atual.
Agora, com um script criado, podemos utilizar o comando `npm run <nome do script>` para executá-lo. No nosso caso, o nome do script é `lint` , então o comando completo fica assim:

Copiar

```
npm run lint
```

Você pode criar quantos scripts quiser, para realizar quais tarefas quiser. Inclusive, pode criar scripts que chamam outros scripts, criando assim "pipelines". Esse tipo de coisa é muito útil, por exemplo, quando trabalhamos *supersets* do JavaScript como o `TypeScript` , ou transpiladores como o `Babel` , pois ambos exigem que executemos comandos adicionais antes de iniciar nossos pacotes.
Agora que vimos o que são scripts, vamos falar de um script especial: o `start` . Esse script é especial pois é utilizado por um comando do npm diferente do `npm run` : o `npm start` . Entenda sobre ele a seguir:

### npm start

O comando `npm start` age como um "atalho" para o comando `npm run start` , uma vez que sua função é executar o script `start` definido no `package.json` .

Como convenção, todo pacote que pode ser executado pelo terminal (como CLIs, APIs e afins) deve ter um script `start` com o comando necessário para executar a aplicação principal daquele pacote.
Por exemplo, se tivermos um pacote que calcula o IMC (Índice de Massa Corporal) de uma pessoa cujo código está no arquivo `imc.js` , é comum criarmos o seguinte script:

Copiar

```
{
  // ...
  "scripts": {
    "start": "node imc.js"
  }
  // ...
}
```

Dessa forma, qualquer pessoa que for utilizar seu script vai ter certeza de como inicializá-lo, pois só vai precisar executar `npm start` .

### npm install

Você provavelmente já utilizou esse comando durante o módulo de Front-End. Ele é o responsável por baixar e instalar pacotes Node.js do NPM para o nosso pacote, e existem algumas formas de usá-lo:

- `npm install <nome do pacote>` : Baixa o pacote do registro do NPM e o adiciona ao objeto `dependencies` do `package.json`
- `npm install -D <nome do pacote>` : É semelhante ao comando anterior. Baixa o pacote do registro do NPM, porém o adiciona ao objeto `devDependencies` do `package.json` , indicando que o pacote em questão não é necessário para executar a aplicação, mas é necessário para desenvolvimento, ou seja, para alterar o código daquela aplicação. Isso é muito útil ao colocar a aplicação no ar, pois pacotes marcados como `devDependencies` podem ser ignorados, já que vamos apenas executar a aplicação, e não modificá-la.
- `npm install` : Baixa e instala todos os pacotes listados nos objetos de `dependencies` e `devDependencies` do `package.json` . Sempre deve ser executado ao clonar o repositório de um pacote para garantir que todas as dependências desse pacote estão instaladas.

---

Ufa! Até aqui, foi bastante informação, né? Antes de prosseguir, dê uma pausa, pegue (ou reabasteça) sua substância energizante preferida

(Cafézinho ☕? Energético ⚡? Água 🥤? Você que manda 😄), respire e, depois, continue a leitura.
Não se preocupe se não estiver decorando tudo. Você vai exercitar muito todos esses conceitos ao longo do restante do módulo de Back-End 😉.

## Criando um script simples

Agora que chegamos até aqui, vamos criar o famoso Hello World em Node.js. Vem com a gente!

### Criando o pacote Node.js

Vamos começar criando uma nova pasta, chamada `hello-world` , onde colaremos nosso código.
Uma vez dentro dessa pasta, execute o comando `npm init` . Deixe todas as perguntas com o valor padrão, a não ser o nome da pessoa autora ( `author:` ), onde você colocará seu nome.
Pronto! Nosso pacote está criado. Abra a pasta `hello-world` no VSCode e vamos começar a criar nosso script.

### Criando o código do Hello, world!

Dentro da pasta `hello-world` , crie um arquivo chamado `index.js` . Por padrão, esse é o arquivo principal de qualquer aplicação Node.js, e é comum darmos esse nome ao arquivo que deve ser executado para iniciar nosso programa. Sendo assim, por convenção, todo pacote Node.js deve ter um arquivo index.js , salvo exceções, que devem ser documentadas no README do repositório.
Dentro do `index.js` , adicione o seguinte código:

Copiar

```
console.log('Hello, world!');
```

E pronto, nosso script de "Hello, world!" está criado! Mas nosso pacote ainda não está pronto. Vamos criar um script `start` para estarmos aderentes às convenções do Node.js.

### Criando o script `start`

Como você viu anteriormente, para criar um script, precisamos alterar o `package.json` da nossa aplicação. Sendo assim, abra o `package.json` da pasta `hello-world` e altere a linha destacada para criar o script start dessa forma:

Copiar

```
// {
//   "name": "hello-world",
//   "version": "1.0.0",
//   "description": "",
//   "main": "index.js",
//   "scripts": {
//     "test": "echo \"Error: no test specified\" && exit 1",
       "start": "node index.js"
//   },
//   "author": "Seu nome",
//   "license": "ISC"
// }
```

### Executando o script

Agora que temos tudo criado e configurado, chegou a hora de executar nosso Hello, world! Para isso, navegue até a pasta `hello-world` no terminal e execute `npm start` .
E pronto! Temos nosso primeiro "Hello, world!" sendo executado com Node.js!
Mas, cá entre nós, essa coisa de "Hello, world!" simples já tá um pouco batida, né? 😝
Vamos, então, dar uma incrementada nesse script, adicionando o nome e sobrenome da pessoa que chamou nosso script!

### Lendo input do terminal

Para podermos ler o nome e sobrenome da pessoa que executou o script, vamos utilizar um pacote de terceiros: o `readline-sync` .
Por tratar-se de um módulo de terceiros, precisamos primeiro instalar o `readline-sync` pra podermos utilizá-lo no código.
Para fazer isso, basta executarmos, dentro da pasta `hello-world` , o comando `npm i readline-sync` . A letra `i` aqui é um atalho para `install` . Ela também funciona com a flag `-D` para `devDependencies` , e sem parâmetro nenhum, para instalar as dependências listadas no `package.json` .
Uma vez instalado o pacote, podemos utilizá-lo em nosso script. Para isso, precisamos, primeiro, importá-lo:

Copiar

```
const readline = require('readline-sync');

// console.log('Hello, world!');
```

Perceba que, apesar do pacote chamar-se `readline-sync` , podemos dar qualquer nome para a `const` que usamos para importá-lo.

---

Agora, com o pacote em mãos, podemos utilizar as funções `question` e `questionInt` disponibilizadas por ele para perguntar à pessoa usuária seu nome e idade:

Copiar

```
// const readline = require('readline-sync');

const name = readline.question('Qual seu nome? ');
const age = readline.questionInt('Qual sua idade? ');

// console.log('Hello, world!');
```

A função `question` interpreta a resposta como uma string comum, ao passo que a função `questionInt` converte a resposta para número utilizando o `parseInt` e retorna um erro caso a pessoa tente inserir algo que não seja um número válido.

---

Pronto, o próximo e último passo é utilizarmos essas novas variáveis para compor nossa mensagem de olá.

Copiar

```
// const readline = require('readline-sync');

// const name = readline.question('What is your name? ');
// const age = readline.questionInt('How old are you? ');

console.log(`Hello, ${name}! You are ${age} years old!`);
```

E, agora, se executarmos novamente, veremos o resultado: perguntamos qual o nome e idade da pessoa e, depois, exibimos uma mensagem personalizada.
Execute novamente o script com `npm start` para vê-lo em ação!

# Exercícios

back-end

## Antes de começar: versionando seu código

Para versionar seu código, utilize o seu repositório de exercícios. 😉
Abaixo você vai ver exemplos de como organizar os exercícios do dia em uma `branch`, com arquivos e `commits` específicos para cada exercício. Você deve seguir este padrão para realizar os exercícios a seguir.

1. Abra a pasta de exercícios:
2. Copiar
3. `$ cd ~/trybe-exercicios`
4. Certifique-se de que está na branch main e ela está sincronizada com a remota. Caso você tenha arquivos modificados e não comitados, deverá fazer um commit ou checkout dos arquivos antes deste passo.
5. Copiar

```
$ git checkout main
```

6. `$ git pull`
7. A partir da main, crie uma `branch` com o nome `exercicios/26.1` *(bloco 26, dia 1)*
8. Copiar
9. `$ git checkout -b exercicios/26.1`
10. Caso seja o primeiro dia deste módulo, crie um diretório para ele e o acesse na sequência:
11. Copiar

```
$ mkdir back-end
```

12. `$ cd back-end`
13. Caso seja o primeiro dia do bloco, crie um diretório para ele e o acesse na sequência:
14. Copiar

```
$ mkdir bloco-26-introducao-ao-desenvolvimento-web-com-nodejs
```

15. `$ cd bloco-26-introducao-ao-desenvolvimento-web-com-nodejs`
16. Crie um diretório para o dia e o acesse na sequência:
17. Copiar

```
$ mkdir dia-1-nodejs-introducao
```

18. `$ cd dia-1-nodejs-introducao`
19. Os arquivos referentes aos exercícios deste dia deverão ficar dentro do diretório `~/trybe-exercicios/back-end/block-26-introducao-ao-desenvolvimento-web-com-nodejs/dia-1-nodejs-introducao`. Lembre-se de fazer commits pequenos e com mensagens bem descritivas, preferencialmente a cada exercício resolvido.

    Verifique os arquivos alterados/adicionados:
20. Copiar

```
$ git status
On branch exercicios/26.1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)
```

21. `  modified:   exercicio-1`

    Adicione os arquivos que farão parte daquele commit:
22. Copiar

```
# Se quiser adicionar os arquivos individualmente
$ git add caminhoParaArquivo

# Se quiser adicionar todos os arquivos de uma vez, porém, atente-se
para não adicionar arquivos indesejados acidentalmente
```

23. `$ git add --all`

    Faça o commit com uma mensagem descritiva das alterações:
24. Copiar
25. `$ git commit -m "Mensagem descrevendo alterações"`
26. Você pode visualizar o log de todos os commits já feitos naquela branch com `git log`.
27. Copiar

```
$ git log
```

```
commit 100c5ca0d64e2b8649f48edf3be13588a77b8fa4 (HEAD -> exercicios/26.1)
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 17:48:01 2019 -0300

    Exercicio 2 - mudando o evento de click para mouseover, tirei o alert e coloquei pra quando clicar aparecer uma imagem do lado direito da tela

commit c0701d91274c2ac8a29b9a7fbe4302accacf3c78
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 16:47:21 2019 -0300

    Exercicio 2 - adicionando um alert, usando função e o evento click

commit 6835287c44e9ac9cdd459003a7a6b1b1a7700157
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 15:46:32 2019 -0300
```

28. `Resolvendo o exercício 1 usando eventListener`
29. Agora que temos as alterações salvas no repositório local precisamos enviá-las para o repositório remoto. No primeiro envio, a branch `exercicios/26.1` não vai existir no repositório remoto, então precisamos configurar o `remote` utilizando a opção `--set-upstream` (ou `-u`, que é a forma abreviada).
30. Copiar
31. `$ git push -u origin exercicios/26.1`
32. Após realizar o passo 9, podemos abrir a `Pull Request` a partir do link que aparecerá na mensagem do push no terminal, ou na página do seu repositório de exercícios no GitHub através de um botão que aparecerá na interface. Escolha a forma que preferir e abra a Pull Request. De agora em diante, você repetirá o fluxo a partir do passo 7 para cada exercício adicionado, porém como já definimos a branch remota com `-u` anteriormente, agora podemos simplificar os comandos para:
33. Copiar

```
# Quando quiser enviar para o repositório remoto
$ git push
```

```
# Caso você queria sincronizar com o remoto, poderá utilizar apenas
$ git pull
```

34.
35. Quando terminar os exercícios, seus códigos devem estar todos commitados na branch `exercicios/26.1`, e disponíveis no repositório remoto do `GitHub`. Pra finalizar, compartilhe o link da Pull Request no canal de Code Review para a monitoria e/ou colegas revisarem. Faça review você também, lembre-se que é muito importante para o seu desenvolvimento ler o código de outras pessoas. 👊🤛

## Agora a prática

Antes de começar, crie uma nova pasta e, dentro dela, crie um pacote Node.js com o `npm init` chamado `my-scripts` . Realize os exercícios dentro desse pacote.

1. Crie um script para calcular o Índice de Massa Corporal(IMC) de uma pessoa.
    1. A fórmula para calcular o IMC é `peso / altura ^ 2` .
       Obs: Lembre-se que a altura é em metros, caso deseje usar em centímetros a conversão para metros será necessária.
    2. Comece criando um novo pacote node com `npm init` e respondendo às perguntas do `npm` .
    3. Por enquanto, não se preocupe em pedir input da pessoa usuária. Utilize valores fixos para `peso` e `altura` .
    4. Armazene o script no arquivo `imc.js` .
2. Agora, permita que o script seja executado através do comando `npm run imc`
    1. O novo script criado deve conter o comando que chama o `node` para executar o arquivo `imc.js` .
3. Chegou a hora de tornar nosso script mais interativo! Vamos adicionar input de quem usa.
    1. Você já utilizou o pacote `readline-sync` para esse fim. Que tal utilizar o mesmo pacote?
    2. Substitua os valores fixos de `peso` e `altura` por dados informados pela pessoa ao responder as perguntas "Qual seu peso?" e "Qual sua altura?" no terminal.
4. Agora temos um problema: `peso` não é um número inteiro! Isso quer dizer que precisamos mudar um pouco a forma como solicitamos o input desse dado.

    1. O pacote `readline-sync` possui uma função específica para tratar esses casos. Consulte a documentação do pacote e encontre o método adequado para realizar input de floats .
    2. Encontrou a função? Show! Agora utilize-a para solicitar o input de `peso` .
5. Vamos sofisticar um pouco mais nosso script. Além de imprimir o IMC na tela, imprima também em qual categoria da tabela abaixo aquele IMC se enquadra:
- Considere a seguinte tabela para classificar a situação do IMC:
- Copiar

```
| IMC                                     | Situação                 |
| --------------------------------------- | ------------------------ |
| Abaixo de 18,5                          | Abaixo do peso (magreza) |
| Entre 18,5 e 24,9                       | Peso normal              |
| Entre 25,0 e 29,9                       | Acima do peso (sobrepeso) |
| Entre 30,0 e 34,9                       | Obesidade grau I         |
| Entre 35,0 e 39,9                       | Obesidade grau II        |
```

- `| 40,0 e acima                            | Obesidade graus III e IV |`
6. Vamos criar mais um script. Dessa vez, para calcular a velocidade média de um carro numa corrida
    1. A fórmula para calcular velocidade média é `distância / tempo` .
    2. Armazene o script no arquivo velocidade.js.
    3. Agora, permita que o script seja executado através do comando `npm run velocidade` . Para isso, crie a chave `velocidade` dentro do objeto `scripts` no `package.json` .
    4. Utilize o `readline-sync` para solicitar os dados à pessoa.
    5. Considere a distância em metros e o tempo em segundos. Repare que, agora, estamos trabalhando com números inteiros.
7. Crie um "jogo de adivinhação" em que a pessoa ganha se acertar qual foi o número aleatório gerado
    1. O script deve ser executado através do comando `npm run sorteio` .
    2. Utilize o `readline-sync` para realizar input de dados.
    3. Armazene o script em `sorteio.js` .
    4. O número gerado deve ser um inteiro entre 0 e 10.
    5. Caso a pessoa acerte o número, exiba na tela "Parabéns, número correto!".
    6. Caso a pessoa erre o número, exiba na tela "Opa, não foi dessa vez. O número era [número sorteado]".

    7. Ao final, pergunte se a pessoa deseja jogar novamente. Se sim, volte ao começo do script.
8. Crie um arquivo `index.js` que pergunta qual script deve ser executado
    1. O script deve ser acionado através do comando `npm start` .
    2. Utilize o `readline-sync` para realizar o input de dados
    3. Quando executado, o script deve exibir uma lista numerada dos scripts disponíveis.
    4. Ao digitar o número de um script e pressionar *enter* , o script deve ser executado.
    5. Você pode utilizar o `require` para executar o script em questão.

## Bônus

1. Crie um script que realize o fatorial de um número `n` .
   O fatorial é a multiplicação de um número por todos os seus antecessores até chegar ao número 1.
    1. Armazene o script no arquivo `fatorial.js` .
    2. Utilize o `readline-sync` para realizar o input de dados.
    3. O script deve ser acionado através do comando `npm run fatorial`
    4. Adicione o script ao menu criado no exercício 5.
    5. Adicione validações para garantir que o valor informado seja um inteiro maior que 0.
2. Crie um script que exiba o valor dos `n` primeiros elementos da sequência de fibonacci.
   A sequência de fibonacci começa com 0 e 1 e os números seguintes são sempre a soma dos dois números anteriores.
    1. Armazene o script no arquivo `fibonacci.js` .
    2. Utilize o `readline-sync` para realizar o input de dados.
    3. O script deve ser acionado através do comando `npm run fibonacci`
    4. Adicione o script ao menu criado no exercício 5.
    5. Não imprima o valor `0` , uma vez que ele não está incluso na sequência;
    6. Quando `n = 10` , exatamente 10 elementos devem ser exibidos;
    7. Adicione validações para garantir que o valor informado é um inteiro maior que 0.