

E-BOOK

DO ESTUDANTE

>>>> NODE + REST API

Servi-dor



Todos os direitos reservados
©2023 Resilia Educação

RESILIA

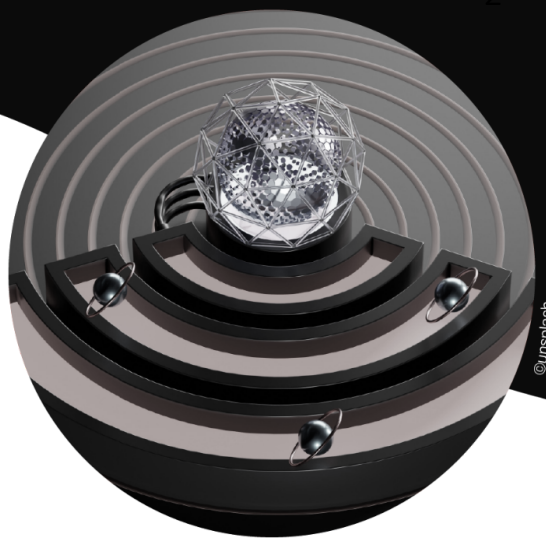
SUMÁRIO

Node.js **3**

EXPRESS **8**

NODE + REST API

Servi-dor



Neste percurso, vamos conhecer a tecnologia Node.js, que é **essencial para podermos navegar pelo universo do *back-end***. Essa área da TI tem como foco criar e desenvolver servidores e API que se comunicam com as interfaces gráficas. Pense no *back-end* como aquele que entrega os dados e no *front-end* como aquele que exibe os dados.

No caso do Node.js, temos um *software* de código aberto e multiplataforma baseado no interpretador v8 do Google, permitindo, assim, a execução de código Javascript fora de um navegador web.

Antes do Node.js, os códigos Javascript eram rodados através de um arquivo *script* utilizando o navegador *web*, ou seja, era necessário utilizar um arquivo HTML juntamente com o navegador para executar um código Javascript. Com o surgimento do Node.js, isso deixou de ser necessário, pois tornou-se possível executar códigos, por exemplo, através do nosso terminal, dentro da nossa máquina, sem precisar da interface gráfica do navegador. Isso certamente nos possibilitou e nos possibilita diariamente uma maior flexibilidade.

Neste *e-book*, vamos **aprender como criar e utilizar códigos com o Node.js e entender como ele viabiliza a utilização de bibliotecas e a criação de servidores *web***, mais conhecidos como APIs Rest. Vamos lá?

CONTEXTUALIZANDO

Sabemos que, quando tentamos desenvolver algo com Javascript, é necessário criar um arquivo HTML para abrir no navegador e só então poder, em seguida, testar o Javascript. O Node é uma *engine* (um motor) de Javascript que nos permite executar o Javascript fora da *web*.

O surgimento de *engines* trouxe muitas mudanças e abriu possibilidades para o desenvolvimento de Javascript, permitindo a criação de APIs em Javascript, entre

outras ferramentas. Foi com ele que o Javascript passou a existir ao lado do *back-end*, dando então uma maior importância à linguagem e também uma maior popularidade, ou seja, o Node fez com que o Javascript se tornasse uma linguagem “*full Stack*” para que, assim como você, outros desenvolvedores possam utilizar a linguagem para desenvolver no *back-end*.

A partir dessa contextualização, fica nítida a importância de conhecer o Node.js para nos tornarmos programadores *full Stack* em uma linguagem flexível, a qual nos permite desenvolver para múltiplos sistemas, como a *web*, *mobile*, *desktop*, entre outros.

Além disso, vale ressaltar que a linguagem Javascript é popular não só entre os desenvolvedores, mas também entre as empresas. Vamos então conhecer um pouco mais sobre o Node.js e o Express?

Glossário

Engine/Motor: Um *engine* ou motor é o núcleo de uma aplicação que fornece as funcionalidades básicas e as ações de um *software*. O Node.js, por exemplo, utiliza o *engine* **V8**, desenvolvido pelo Google. O *engine* é responsável por interpretar e executar o código JavaScript no lado do servidor, o que nos permite executar o Javascript fora de um navegador.

Node.js



Para utilizar o Node.js, é necessário baixar e instalar o seguinte programa em sua máquina:

[Node.js \(nodejs.org\)](https://nodejs.org)



Após baixar o programa, os passos de instalação são sempre pressionar o NEXT.

Uma vez instalado o Node.js, já podemos utilizá-lo. Primeiramente, precisamos criar um arquivo Javascript e permitir que o Node execute o código dentro desse arquivo criado. Para executar um arquivo usando o Node.js, chamamos em nosso terminal “`node /localizaçãoDoArquivo`”. Para abrir um terminal dentro da pasta-projeto, por exemplo, seria necessário utilizar o seguinte código para executar o arquivo “main.js”: **`node ./main.js`**.

Vamos colocar em prática o que acabamos de aprender?



Atividade: Somando

Crie um arquivo Javascript e, dentro dele, escreva uma variável com o valor 10 e uma variável com o valor 90. Em seguida, apresente o valor da soma das duas variáveis no console utilizando o Node.js.

Outra funcionalidade importante dentro do Node.js é nos permitir “instalar” uma biblioteca (lib) feita por outros programadores. Dessa forma, podemos utilizar diferentes bibliotecas dentro do nosso projeto, sendo cada uma para uma funcionalidade específica. Assim, podemos economizar tempo de produção.

Para entender a importância dessas bibliotecas, imagine que estamos fazendo um desenho de um corpo humano e que ficamos em dúvida sobre como desenhar as mãos, o nariz, entre outras partes. Não seria muito produtivo, nesse caso, se pudéssemos utilizar essas partes do corpo já desenhadas e criadas por outras pessoas e ter o acesso a essas partes de forma simplificada? É isso que o Node.js nos permite fazer, uma vez que, com ele, podemos instalar em nosso projeto os recursos que outros desenvolvedores já desenvolveram.

É possível, por exemplo, pegar uma *lib* de tradução para traduzir um *site* em múltiplos idiomas ou, ainda, utilizar uma biblioteca para alternar entre modo escuro e claro... Tudo isso já está pronto e disponível para utilizarmos em nossos projetos! Assim, cada biblioteca que usamos nos ajuda a desenvolver um projeto de forma rápida.

Para instalar tais bibliotecas, é necessário utilizar gerenciadores que procurem, baixem, descompactem e extraiam o pacote dessas bibliotecas. Tais tecnologias já existem, e uma delas é o NPM (*Node Package Manager*). Ele nos permite utilizar bibliotecas e gerenciá-las, de modo que tudo fique mais compacto e seja tudo realizado em comandos no terminal.

O NPM já vem de forma nativa instalada juntamente com o Node, por ser o gerenciador de bibliotecas padrão do Node.

Podemos pensar no NPM como se fosse uma máquina de lavar roupa: primeiro colocamos a roupa dentro dela, depois adicionamos o sabão e, por fim, programamos a lavagem através dos comandos disponíveis no painel para que ela lave a roupa. A máquina, durante a lavagem, faz diversas operações e, após finalizar, entrega a roupa lavada. Esse processo simplifica aquele que antes era feito manualmente, e isso também acontece com o NPM: ele executa diversas operações através do comando que usamos e, no final, ele nos entrega um pacote instalado.

Glossário

NPM: NPM é o acrônimo de *Node Package Manager*. Ele é um gerenciador de pacotes para o Node.js que nos permite instalar, compartilhar e gerenciar pacotes de bibliotecas de *software*.

Para procurar por uma biblioteca, devemos acessar o site [npm \(npmjs.com\)](https://npmjs.com)

Para instalar as bibliotecas, devemos utilizar o seguinte comando:

```
npm install [nome-da-biblioteca]
```

Outros comandos úteis são:

COMANDO	FINALIDADE
<code>npm init</code>	Inicia um repositório Node dentro da pasta
<code>npm install</code>	Instala as bibliotecas vigentes que estejam sinalizadas no package.json (o arquivo de configuração)
<code>npm uninstall [nome-da-biblioteca]</code>	Desinstala um pacote
<code>npm install -g [nome-da-biblioteca]</code>	Instala um pacote de forma global em seu computador

Confira outros comandos úteis acessando o *link* a seguir:

[Conheça um pouco mais sobre o sistema de módulos em Node.JS e NPM](#)

Vamos agora realizar um passo a passo para colocar em prática a instalação e a utilização de uma biblioteca e entender melhor como isso é feito:

1. Para iniciar, vamos criar uma nova pasta para o nosso projeto (aqui nomeada de “colorindo”).
2. Dentro da pasta, vamos dar início ao nosso projeto criando um arquivo Javascript de nome “main.js”.

3. Em seguida, vamos abrir um terminal dentro da pasta. Se for com o *Windows* terminal, basta clicar com o botão direito do *mouse* e ir em “abrir terminal *windows* aqui”. Se for com o Visual studio code, é preciso ir até a aba superior em terminal e em “new terminal” ou “novo terminal”.
4. Após abrir o nosso terminal, vamos então executar o seguinte comando:

```
npm install colors
```

Este comando serve para informar ao NPM que ele irá instalar a biblioteca *colors* dentro do nosso projeto (da nossa pasta nomeada de “colorindo”).

Note que, após executar o comando, foram criados dois arquivos e uma pasta, Mas o que eles fazem?

- **node_modules** → Esta pasta contém o código de todas as bibliotecas instaladas. Ela é criada pelo NPM para armazenar a biblioteca baixada, permitindo, assim, que ela seja utilizada em nosso projeto.
- **package.json** → Este arquivo contém todas as informações dos nossos projetos. Ele também é criado pelo NPM, mas sua função é informar o Node sobre configurações e informações do projeto.
- **package-lock.json** → Este arquivo contém todas as informações de cada pacote instalado. Também criado pelo NPM, ele serve para informar o Node sobre os pacotes instalados.

Assim, vemos que cada arquivo e cada pasta criados pelo NPM são necessários para que o Node entenda a situação do projeto e das dependências (bibliotecas instaladas).

Após instalada a biblioteca dentro do nosso projeto, vamos entrar em nosso arquivo “main.js” e vamos dar início ao nosso projeto, assim como é pontuado na documentação da biblioteca.

Lembre-se sempre de olhar a **documentação** de cada biblioteca a ser instalada para saber quais são os *status*, como usar, para que ela serve, entre outras informações importantes que a documentação pode conter.

Em seguida, para importar a biblioteca que foi instalada, usamos o seguinte código:

```
var colors = require('colors');
```

Toda e qualquer biblioteca que nós instalamos precisa ser importada para podermos utilizá-la. Isso acontece porque o Javascript nos permite criar módulos, ou seja, partes de nossos códigos, e importar esses módulos em qualquer lugar e para os utilizar.

Pense na seguinte analogia: quando estamos arrumando a casa, por exemplo, sempre vamos de um cômodo para o outro, pois seria muito trabalhoso arrumar todos ao mesmo tempo. É a mesma coisa com o Javascript: dividimos os códigos em partes, separadas visualmente, para que tenhamos uma melhor performance ao programar. Porém, o Node, no final de tudo, junta todas as partes em um único arquivo Javascript, permitindo, assim, que ele execute o nosso código.

Vamos, então, finalizar nosso projeto. Após importar a biblioteca com o `require('colors');` começamos a utilizar a biblioteca.

Na documentação, encontramos alguns exemplos de utilização, os quais vamos utilizar para entender o que essa biblioteca faz:

```
var colors = require("colors");
```

```
console.log("hello".green); → Gera no console um texto na cor verde.
```

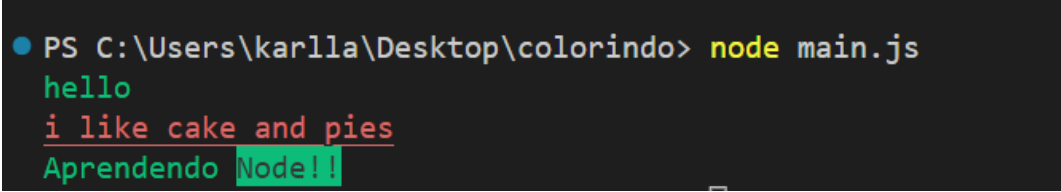
```
console.log("i like cake and pies").underline.red) → Gera no console um texto com underline em vermelho.
```

```
console.log("Aprendendo".green, "Node!!".bgGreen) → Gera no console uma junção de dois textos, um na cor verde e outro com a cor de fundo verde.
```

Note que, sempre que colocamos um `console.log`, estamos utilizando o ponto em um texto para aplicarmos um efeito no texto. Em `"Aprendendo".green`, estamos adicionando a cor verde ao texto "Aprendendo".

Vamos agora verificar o resultado. Para isso, executamos o comando:

```
node main.js
```



```
● PS C:\Users\karlla\Desktop\colorindo> node main.js
hello
i like cake and pies
Aprendendo Node!!
```

Vemos, por fim, a mudança de cor no console, o que significa que essa biblioteca nos ajudou a colorir o console. Ela é utilizada em diversas plataformas e sistemas, tanto para alertar erros quanto para informar o usuário de alguma possível infração, entre outras possibilidades.

O Node, portanto, permite-nos utilizar diversas bibliotecas, com diversos recursos que possam nos ajudar a desenvolver uma aplicação.

Vamos treinar o que vimos até o momento! Na atividade a seguir, repetiremos os processos utilizando uma biblioteca que nos proporciona um recurso diferente.

Vamos lá?



Atividade: Traduzindo

Instale a biblioteca e a documentação disponibilizadas abaixo.
Depois, traduza a palavra “hello node” em quatro línguas diferentes.

[translate - npm \(npmjs.com\)](https://www.npmjs.com/translate)

Dica: utilize a *engine* do Google.



Para refletir

- De que forma o Node.js ajuda os programadores?
- Como era a utilização do Javascript antes do Node.js?
- O que é uma biblioteca? Qual é a importância dela em nossos projetos?

EXPRESS



Aprendemos até o momento a utilizar o Node e as suas bibliotecas (conhecidas também com o nome de *packages* ou pacotes). Agora vamos aprender a construir uma API Rest utilizando a biblioteca Express.

O Express é um *framework* que funciona para aplicativo da web do Node.js mínimo e flexível, que nos permite acessar um vasto conjunto de recursos para as nossas aplicações. Ou seja, o Express nos permite desenvolver uma API Rest que se comunica através da *web*, de modo que a enviar dados de um lado para o outro.

Glossário

API: API é o acrônimo de *Application Programming Interface*. Ela é um conjunto de rotinas, protocolos e ferramentas para construir um *software*. As APIs permitem que diferentes aplicativos possam se comunicar entre si e compartilhar dados.

Uma API envia dados através da *web* por meio de requisições. É como se a API fosse o correio que está com a sua carta e que, posteriormente, envia essa carta para você. Tal ato de enviar a carta na *web* é chamado de requisição.

Ao abrir uma rede social, por exemplo, a API trabalha com requisições *web* para enviar dados e para aparecerem os *cards*, fazendo com que o usuário tenha uma boa experiência. Quase todos os *sites* da *web* atualmente utilizam esse recurso.

Podemos pensar na API como o coração da aplicação, pois é ela que vai permitir que o *site* tenha o dado de cadastro dos usuários, as mensagens, os *status*, entre outras informações. Ou seja, quem gerencia tudo é a API. Agora que entendemos esse conceito, vamos avançar e aprender a criar uma API.

Para saber mais sobre como funciona uma API, leia o e-book “Requisições Json (API) Enviando Cartas”

Para criar uma API com o Express, primeiramente é necessário criar uma pasta para o nosso projeto. Vamos criar a pasta de nome “TesteApi”. Dentro dela, em seguida, é preciso abrir um terminal e executar o seguinte comando:

```
npm init
```

Esse comando nos permite iniciar um projeto *npm* dentro da pasta, e ele geralmente solicita os seguintes dados: o “package name”, a “version”, a “description”, o “entry point”, o “test command”, o “git repository”, as “keywords”, o “author” e a “license”.

Todos esses dados podem ser preenchidos com seus dados pessoais, mas não é obrigatório o preenchimento. Após isso, conferimos todos os dados e digitamos “yes”. Em seguida, um arquivo irá surgir, o *package.json*, e dentro dele estarão todos os dados que preenchemos.

Essa etapa é necessária porque, em muitos casos, algumas bibliotecas não criam esse arquivo, o que gera erros ao rodar o Node. Para prevenir, então, rodamos esse comando e criamos de forma manual esse arquivo.

Agora que já temos nosso arquivo de configuração, vamos então iniciar a nossa API. Para isso, precisamos instalar o “*package*” (pacote ou biblioteca) do Express. Este comando utiliza uma sintaxe minimalista para instalar o pacote Express:

```
npm i express
```

Em seguida, vamos criar o arquivo principal chamado “*app.js*”, onde vamos desenvolver a nossa API.

Após a instalação da biblioteca, precisamos importá-la em nosso arquivo “app.js” e iniciar a instância de nossa API, como podemos observar no código a seguir:

```
var express = require("express");  
const app = express();
```

Note que nós temos uma variável de nome “express” que está importando o Express através do **require(“express”)**, permitindo utilizar tudo que ele nos disponibiliza.

Temos, então, a nossa **const app**, que agora contém toda a instância e todo o serviço da nossa API. O que precisamos fazer na sequência é configurar as nossas rotas com o método GET, que é o método que representa o Reader do CRUD, ou seja, ele simboliza *ler ou pegar algum dado*.

Já que vamos chamar a instância e o verbo http, então precisamos definir a rota ou o caminho em que iremos escutar. Considere que esses caminhos sejam pastas: para cada pasta (caminho), você consegue definir um CRUD (*Create, Reader, Update e Delete*).

Após definirmos o caminho, temos que definir a ação: se o usuário acessar aquele caminho com o verbo configurado, será realizada uma ação. Nesse caso, chamamos a *request* (todo e qualquer dado passado através da requisição) e a *response* (aquilo que iremos retornar/devolver). Para isso, utilizamos a seguinte sintaxe:

```
const { response } = require("express");  
var express = require("express");  
const app = express();  
app.get("/", (request, response) => {  
    response.send("Olá mundo")  
})
```

Note que, na sintaxe acima, temos a chamada da instância (*app*), o verbo (*get*), o nosso caminho “/” (definido como caminho principal de uma rota) e temos a nossa ação (*arrow function*), a qual está recebendo dois parâmetros, sendo eles a *request* e a *response*.

Dentro da nossa ação, estamos enviando uma resposta (*send response*) com a palavra “Olá mundo”. Porém, ainda não iniciamos nossa API. Para iniciá-la e poder testar essa rota, precisamos definir um ouvinte para ficar escutando uma porta em nosso computador. Vale ressaltar que portas são serviços como o *Bluetooth* e o *Wi-Fi*, os quais são portas de comunicação onde conseguimos transmitir dados, sendo, por esse motivo, definidos como serviços.

Para definir um ouvinte, iremos utilizar a sintaxe a seguir:

```
const porta = 3000
app.listen(porta, () => {
  console.log('Server listening on port 3000')
})
```

Na execução a seguir, vemos nossa constante *porta* que contém um número de uma porta (3000) e, logo abaixo, vemos a chamada da nossa instância juntamente com o ouvinte, ou seja, nossa API ficará ouvindo requisições que forem feitas na porta 3000 do nosso *localhost* (o *host local* é uma rota interna de nosso navegador, sem acesso à internet, que permite acessar as portas dentro do nosso computador).

Quando a API começa a escutar a porta, ela passa a exibir no console a notificação “API online”. Para executar a API, precisamos então executar o seguinte comando:

```
node ./app.js
```

Para visualizar a resposta, utilize o programa Insomnia:

[Download - Insomnia](#)

Ou você pode somente abrir o *link* da API:

<http://localhost:3000>

Vimos, neste *e-book*, como utilizar o Node, suas bibliotecas, entre outras funções que nos ajudam a desenvolver uma API utilizando o Express, que é o *framework* de Javascript usado para desenvolver APIs que se comunicam através da *web*. Entendemos, também, como uma API funciona e como tudo isso se conecta com o mercado de tecnologia atualmente, já que sistemas, aplicativos, *sites*, entre outras possibilidades usam uma API para conseguir captar dados e, consecutivamente, conseguir fazer o *layout* planejado funcionar conforme o esperado.

Para refletir



- Para que utilizamos o Express? Por que ele é considerado um framework Node?
- Por que a web precisa da API? Qual é a importância de o front-end ter uma API?

Para ir além



- Sabemos que o Node.js revolucionou o jeito de utilizar o Javascript, permitindo a utilização fora da web e de apenas um terminal local em nossa máquina. Por isso, é fundamental conhecer bem os seus conceitos. Para saber mais, confira os links a seguir:
 - <[O Que É npm? Introdução Básica para Iniciantes](#) >
 - <[JavaScript Package Managers: NPM Vs YARN Vs PNPM](#)>
 - <[How different is CommonJs require from ES6 import? - DEV Community](#)>
- Estes dois vídeos apresentam, de modo mais visual, informações e conteúdos para você aprofundar seus conhecimentos sobre Node e Express:
 - ▶ [Curso de Node.JS - O que é Node.JS #01](#)
 - ▶ [Curso de Node.js - Introdução ao Express #06](#)
- Como vimos, APIs são bastante importantes para a web e para as aplicações *mobile*. Pensando nisso, este e-book fez uma introdução ao mundo da API. Para se aprofundar mais em criações de API usando Express, consulte os seguintes sites:
 - <[Introdução Express/Node - Aprendendo desenvolvimento web](#) >
 - <[Como criar API simples NodeJS + Express](#) >
- O livro indicado ajuda a entender como cada elemento funciona dentro de uma API com Node e Express, além de explicar a importância de conhecer e utilizar todos os recursos disponíveis que o Express nos oferece.

Livro: Programação web com Node e Express

NUVEM DE PALAVRAS

