Node.js

O que é?

Um interpretador de JavaScript para executá-lo fora do navegador.

Quem usa?

- Paypal
- Netflix
- LinkedIn
- Uber
- eBay
- Yahoo
- Mozilla
- por aí vai.

Por que aprender?

É interpretado, então não tem encheção de saco com IDE e compilador específicos.

É JavaScript, então nada de sair muito da zona de conforto.

O que vamos fazer?

1. Aprender o que é Node.js, porque utilizar e os conceitos básicos.
2. Conhecer o npm.
3. Entender um pouco da arquitetura do Node.js e como ele funciona “debaixo dos panos”
4. Estudar o fluxo assíncrono do Node.js

Node.js – Que porra é essa?

Primeiro, vamos parar de frescura, fala só **Node** que é mais fácil.

O Node é um motor JavaScript que surgiu com base no **V8**, que é a ferramenta que o Google Chrome usa para ler e executar instruções em JavaScript.

Esse tipo de software normalmente é chamado de **interpretador, engine,** ou **runtime**.

Apesar de ter sido baseado no **V8**, o Node tem algumas diferenças do engine que roda nos navegadores. Por exemplo, não tem aqueles métodos usados para trabalhar com as paginas Web e nem métodos que permitem acessar o sistema de arquivos e a rede de forma mais direta.

Por que eu vou usar isso?

A comunidade do Node é gigante e muito ativa, então se você pensou em criar algo do zero, provavelmente já existe e você pode instalar usando **npm install**.

Outra coisa relevante é a…

Performance

O Node permite escrever softwares servidores de requisição HTTP de forma muito mais eficiente que várias linguagens. Isso porque ele consegue fazer operações de leitura e escrita, tanto no disco quanto na rede, de forma não bloqueante. Quando o servidor recebe uma requisição e precisa, por exemplo, buscar dados no banco de dados, as demais requisições não precisam esperar que essa primeira requisição termine para que elas possam ser atendidas, processando de forma **concorrente**.

Como as aplicações em Node são bem eficientes e otimizadas nesse sentido, elas acabam por consumir menos recursos dos servidores. Sabe o que isso significa?
Dindin, bufunfa, grana, capilé, cascalho, faz-me-rir…

Aplicações em tempo real

Node não ser bloqueante facilita a vida para implementar recursos que facilitam o trabalho das operações que acontecem em tempo real.

Bibliotecas como o socket.io [https://socket.io/] permitem que aplicações em tempo real relativamente complexas (como chats com multiplos usuários, chats privados e etc.) sejam escritas com poucas linhas de código, de maneira bem completa.

Suporte nativo a esse tipo de tecnologia é uma mão na roda.

Node é JavaScript

Nada de ter de se adaptar a uma nova linguagem, a curva de aprendizado é super tranquila, e hoje em dia se usa JavaScript desde Web e Mobile até IOT e Televisão.

Sistema de módulos

Como já foi dito, a comunidade do Node é gigante, e isso significa que existe uma infinidade de pacotes e bibliotecas disponíveis de graça.

Para poder utilizar desse recurso, precisamos entender o que é um **módulo** dentro no Node.

Um módulo é um pedaço de código que pode ser organizado em um ou mais arquivos, e que possui escopo próprio. Em resumo, um módulo é uma funcionalidade ou um conjunto de funcionalidades que está isolado do restante do código.

O Node possui três tipos de módulos.

1. Internos
2. Locais
3. De terceiros

Módulos internos

Os módulos internos (ou core modules) são os módulos que estão inclusos no Node por padrão e já vem instalados quando instalamos o engine.

Se quiser saber mais sobre módulos internos, pesquise sobre:

- *fs* – módulo para manipulação do sistema de arquivos (bem útil pra você)
- *url* – utilitário para ler e manipular URLs
- *os* – ferramentas relacionadas ao sistema operacional
- *util* – ferramentas que normalmente são úteis para programadores

E por aí vai. Uma googlada rápida já entrega tudo que você precisa pra utilizar esses módulos, pesquise **fs module node docs** e divirta-se

Módulos locais

São os módulos que você cria. São partes do seu programa que foram separados em arquivos diferentes. É possível publicar seus módulos no **NPM** para ajudar outras pessoas.

Módulos de terceiros

Os módulos de terceiros são os que outras pessoas criaram e disponibilizaram via **npm**. *Memorize isso, você vai usar muitos módulos de terceiros.*

Como importar e exportar módulos?

Para utilizar o conteúdo de um módulo de outro arquivo no Node, precisamos importar esse módulo para o contexto da nossa aplicação. Existem dois sistemas de módulos mais difundidos na comunidade de Javascript:

- **ES6**
- **CommonJS**

No ES6, os modulos são importados utilizando a palavra-chave **import** e exportados utilizando a palavra-chave **export** mas existe um problema:

O Node não tem suporte nativo ao ES6, o que nos obriga a utilizar transpiladores como o Babel.

> *Transpiladores são aplicações que lêem um código-fonte escrito em uma linguagem e produzem o código equivalente em outra linguagem.*

Dessa forma, a princípio utilizaremos o sistema **CommonJS**, afinal, é o sistema de módulos nativo do Node.

Agora que já temos uma visão geral, vamos aprender a **exportar** algo de um arquivo JavaScript.

Exportando Módulos

Para exportar algo no sistema CommonJS, utilizamos uma variavel global que existe nativamente no ambiente Node:

> module.exports

E atribuimos a ela o valor que desejamos exportar. Segue o exemplo:

```
1  //name.js
2  const name = 'Rogério';
3
4  module.exports = name;
5
```

Dessa forma, estamos exportando a variável **name** presente no arquivo **name.js**

Suponhamos que agora queremos exportar também outra variável com o sobrenome:

```
1  //name.js
2  const name = 'Rogério';
3  const lastName = 'Serpa';
4
5  module.exports = {
6    name,
7    lastName
8  };
9
```

Aqui enviamos ela dentro de um objeto com as chaves **name** e **lastName**.

Você pode exportar qualquer coisa: Desde variáveis simples, até funções ou mesmo classes.

Importando módulos

Ok, exportamos os dados, mas como vamos ter acesso a eles em outro arquivo? Importando.

Utilizando os dados do arquivo **name.js** criado anteriormente, importaremos os dados para dentro de uma variável utilizando a função nativa **require()**:

```
1  //index.js
2  const data = require('./name.js');
3
4  console.log(data.name); // Saída: Rogério
5  console.log(data.lastName); // Saída: Serpa
6
```

Como o que foi exportado do arquivo **name.js** foi um objeto com duas posições, quando a variável **data** recebeu o retorno da importação com o **require('./name.js')**, ela recebeu exatamente este objeto com duas posições.

Sendo um objeto, podemos também utilizar a técnica de *destructuring:*

```
1  //index.js
2  const {name, lastName} = require('./name.js');
3
4  console.log(name); // Saída: Rogério
5  console.log(lastName); // Saída: Serpa
6
```

Como vimos, para uma importação local, basta utilizarmos a função **require** passando como argumento o caminho relativo do caminho do seu arquivo.

> *NOTA: O **require** funciona apenas com arquivos de extenção **.json** e **.js***

Para importarmos módulos internos, basta passarmos como argumento para o **require** o nome do pacote:

```
1  const fs = require('fs');
2
```

O nome da variavel que vai receber o **require** pode ser qualquer coisa, o importante é o nome do pacote passado pro **require**.

Importando módulos de terceiros

O processo de importação de módulos de terceiros é o mesmo de módulos internos, passar o nome padrão do pacote como parâmetro para a função **require**, mas como eles não são nativos, é necessário instalá-los no projeto onde serão necessários.

O registro oficial do Node onde encontramos estes pacotes é o **npm**. O **npm** também é o nome da interface de linha de comando responsável por baixar e instalar esses pacotes. Essa interface é instalada junto com o Node.

Quando importamos um módulo que não é nativo do Node e não aponta para um arquivo local, o Node busca esse módulo no diretório **node_modules** mais próximo do arquivo que chamou essa importação. Se o módulo for encontrado, ele é carregado, caso contrário o processo é repetido um

diretório ‘acima’. Isso acontece até o módulo ser encontrado ou não existir mais nenhum diretório **node_modules** no local em que o Node está procurando.

E já que entramos no assunto, vamos falar do diabo do **npm**.

NPM

O **npm** (sigla pra **node package manager**) é o repositório oficial para publicação de pacotes Node. Um pacote é um conjunto de arquivos que exporta um ou mais módulos.

O CLI (command line interface) do **npm** é uma ferramenta que serve nos ajudar a gerenciar pacotes, sejam dependencias do nosso projeto ou nossos próprios pacotes. É com ele que criamos um projeto, instalamos e removemos pacotes, além de publicarmos e gerenciarmos versões dos nossos próprios pacotes.

Publicar um pacote no npm é gratuito, somente se open-source. Se quiser hospedar um pacote de forma privada, é necessário pagar uma assinatura.

Os principais comandos do npm são:

npm init – cria um pacote Node na pasta onde é executado. Ao ser executado, pede algumas informações que podem ser deixadas como padrão, apenas apertando **Enter**. Quando executado, cria um arquivo chamado **package.json** com alguns metadados sobre seu pacote, além de nome, versão, dependencias e scripts.

npm run – executa um script configurado no **package.json**, um exemplo é o **npm run start** que é usado para iniciar um live-server de uma aplicação em **React.JS**

npm install – baixa e instala todos os pacotes listados nos campos *dependencies* e *devDependencies* do **package.json**. Pode ser executado passando o nome de um pacote de terceiros para instalar a dependencia no seu projeto também, por exemplo: *npm install express*.

DESAFIO

Você deve criar uma calculadora que recebe 2 números, qual a operação que deverá ser feita (adição, subtração, multiplicação, divisão) e ela deve imprimir o resultado na tela.

Iremos utilizar modulos locais e de terceiros nessa atividade.

Requisitos:

1 – O campo ‘description’ deve ser preenchido com uma descrição semântica quando executar o npm init. Os outros campos podem ser deixados como padrão ou preenchidos a sua escolha.

> *NOTA: É recomendado deixar o campo ‘repository’ vazio até que você crie o repositório no github. Você pode editar isso mais adiante no arquivo* ***package.json***

2 – Criar um script que inicie sua aplicação dentro do **package.json**

> *NOTA: O padrão para o nome do arquivo que irá iniciar sua aplicação é* ***index.js***

3 - O arquivo com as funções que executam as operações deve ser um módulo local que exporta essas funções.

4 – A aplicação deve solicitar o primeiro numero, depois o segundo, a operação, e ao final exibir o resultado.

Dicas:

Para receber uma entrada no terminal através do Node, recomendo o pacote readline-sync. É simples de usar e bem explicado na documentação. [https://www.npmjs.com/package/readline-sync]