

# INTRODUÇÃO AO NODEJS

Trabalhar com funções assíncronas tem ótimos benefícios quando se trata de processamento I/O no servidor. Isso acontece devido ao fato de que uma chamada de I/O é considerada uma tarefa muito custosa para um computador realizar. Tão custosa que chega a ser perceptível para um usuário.

Por exemplo, já percebeu que quando você tenta abrir um arquivo pesado de mais de 1 GB em um editor de texto o sistema operacional trava alguns segundos ou até minutos para abrir? Agora imagine esse problema no contexto de um servidor, que é responsável por abrir esse arquivo para milhares de usuários. Por mais potente que seja o servidor, eles terão os mesmos bloqueios de I/O que serão perceptíveis para o usuário, a diferença é que um servidor estará lidando com milhares usuários requisitando I/O (às vezes milhares requisitando ao mesmo tempo).

É por isso que o Node.js trabalha com assincronismo. Ele permite que você desenvolva um sistema totalmente orientado a eventos, tudo isso graças ao Event-loop.

O Event-loop é um mecanismo interno, que utiliza internamente as bibliotecas:

- [libev](#)
- [libeio](#)

Elas são nativas da linguagem C e responsáveis por prover a funcionalidade de assíncrono I/O para o Node.js.

## Como o Event-loop funciona?



### OS EVENTOS SÃO PROCESSADOS UM POR VEZ

Basicamente ele é um loop infinito, que em cada iteração verifica se existem novos eventos em sua fila de eventos. Tais eventos somente aparecem nessa fila quando são emitidos durante as emissões de eventos na aplicação.

O EventEmitter, é o módulo responsável por emitir eventos, e a maioria das bibliotecas do Node.js herdam desse módulo suas funcionalidades de emit e listen de eventos.

Quando um determinado código emite um evento, o mesmo é enviado para a fila de eventos para que o Event-loop execute-o, e em seguida retorne seu resultado em um callback. Tal callback geralmente é executado através de uma função de escuta, ou mais conhecida como funções de listen, semanticamente conhecida pelas funções: `on()`, `listen()` e outras.

Programar orientado a eventos vai manter sua aplicação mais robusta e estruturada para lidar com funções que são executadas de forma assíncrona não-bloqueantes.

## Recapitulando as bases do Javascript

### Comentários

- Comentários de única linha;
- Comentários de múltiplas linhas.

```
// Este comentário ocupa uma única linha
```

```
/* Já este comentário  
é mais longo e utiliza  
várias linhas */
```

### Declarando variáveis

- VAR
- LET
- CONST

```
function exemplo() {  
  //x poderia ser acessado aqui  
  for(var x = 0; x < 5; x++) {  
    //x existe aqui  
  };  
  //x está visível aqui novamente  
};
```

```
function exemplo() {  
  //x não existe aqui  
  for(let x = 0; x < 5; x++ ) {  
    //x existe aqui  
  };  
  //x não está visível aqui novamente  
};
```

## Funções

```
function nomeDaFuncao( /*parâmetros*/ ) {  
    /* código que será executado */  
  
    return /*Valor retornado*/;  
}
```

## Hierarquia do objeto

```
//Construtor  
function Exemplo() {  
    this.propriedade = 'Isso é uma propriedade',  
    this.metodo = function() {  
        return 'Isso é um método';  
    }  
}  
  
var objeto = new Exemplo(); //Instância do construtor "Exemplo"  
  
//Alerta os respectivos textos na tela  
alert(objeto.propriedade),  
alert(objeto.metodo());
```

## Bibliotecas e componentes externos:

### Exemplo de biblioteca para e-mail

```
var mail = {  
    from: "Fulano <suaconta@gmail.com>",  
    to: "destinatario@gmail.com",  
    subject: "Envio de email usando Node.js",  
    text: "Olá mundo!",  
    html: "<b>Olá mundo!</b>"  
}
```

O **require** existe só em CommonJS (a maneira que o Node.js criou para importar e exportar módulos dentro de uma aplicação)

O **import** é ES6, ou seja uma nova ferramenta que ambos JavaScript do browser e JavaScript do servidor (Node.js) podem usar.

```
import Library from 'some-library';
```

```
Library = require('some-library');
```

## TRABALHANDO COM ARRAYS

### Inserir Elementos em um Array

Até agora, vimos como declarar e inicializar um [Array \(vetor\)](#). Agora, vamos ver como podemos inserir dados em um array.

Como vimos anteriormente, um array é composto por índices que determinam a posição de um dado dentro daquele array. Quando vamos inserir um novo dado devemos colocá-lo no final do array.

A questão está em: como podemos achar último índice de um array?

De uma forma simples, podemos usar a nossa lógica e pensar um pouco.

Como cada posição do array é igual a *posição-1*, então, se temos um array de tamanho 5, o maior índice que possuímos é 4. Com isso podemos concluir que, o tamanho do array será sempre seu próximo índice.

Array	Último Índice	Tamanho	Próximo Índice
[0],[1],[2]	2	3	3
[0],[1],[2],[3],[4]	4	5	5
[0],[1],[2],[3],[4],[5],[6],[7]	7	8	8

Sabendo disso, usaremos a propriedade `length` do objeto Array para obter o tamanho do vetor e, assim, conseguirmos inserir mais um elemento.

```
var vetor = new Array ();
var i, proximo;
for (i = 1; i <= 5; i++) {
    proximo = vetor.length;
    /*
    obtendo o tamanho do array
    para descobrir qual o último índice
    */
    vetor[proximo] = "Dado " + i;
}
for (i=0; i<vetor.length; i++) {
    document.write(vetor[i] + "<BR>");
}
```

O que resulta em:

Dado 1  
Dado 2  
Dado 3  
Dado 4  
Dado 5

## Adicionando elementos com push()

O método `push()` adicionará um elemento ao final de um vetor, enquanto sua função dupla, o método `pop()`, removerá um elemento do final da matriz. Para adicionar um elemento ao final de um array usando `push()`, você precisa fazer isso:

```
1. var list = ["foo", "bar"];
2.
3. list.push("baz");
4.
5. ["foo", "bar", "baz"] // resultado
```

Você também pode adicionar vários elementos a um array usando `push()`, como mostrado abaixo:

```
1. var list = ["foo", "bar"];
2.
3. list.push("baz", "qux", "etcetera");
4.
5. ["foo", "bar", "baz", "qux", "etcetera"] // resultado
```

Se você precisar adicionar um elemento ou vários elementos ao final de uma matriz, o método `push()` será quase sempre sua opção mais simples e rápida.

## Adicionado elemento com unshift()

O método `unshift()` adicionará um elemento ao início de um vetor, enquanto sua função dupla, `shift()`, removerá um elemento do início do vetor. Para adicionar um elemento, execute o método da seguinte forma:

```
1. var list = ["foo", "bar"];
2.
3. list.unshift("baz");
4.
5. ["baz", "foo", "bar"] // resultado
```

Para adicionar vários elementos ao início de um array usando `unshift()`, tente isto:

```
1. var list = ["foo", "bar"];
2.
3. list.unshift("baz", "qux");
4.
5. ["baz", "qux", "foo", "bar"] // resultado
```

Se você precisar adicionar elementos ao início de uma matriz, o método `unshift()` será quase sempre sua opção mais simples e rápida.

## Adicionando elemento com splice()

O método splice() modifica o conteúdo de um array removendo elementos existentes e / ou adicionando novos elementos. Abaixo, você encontrará a sintaxe correta da função splice():

```
1. array.splice(inicio, quantidade [, item1 [, item2 [, ...]]])
```

Se você quiser inserir um elemento (ou elementos) em um determinado ponto em algum lugar dentro do array, além do começo ou fim, então você deve estar usando o método splice(). Para usar splice() seu código deve ficar assim:

```
1. var list = ["foo", "bar"];
2.
3. list.splice(1, 0, "baz"); // na posição de índice 1, remova 0 elementos e adicione "baz" a essa posição
4.
5. // o elemento "bar" agora será movido automaticamente para a posição de índice 2
6.
7. ["foo", "baz", "bar"] // resultado
```

Para adicionar vários elementos no meio de um array usando splice() tente isto:

```
1. var list = ["foo", "bar"];
2.
3. list.splice(1, 0, "baz", "qux");
4.
5. ["foo", "baz", "qux", "bar"] // resultado
```

O comando splice é como o canivete suíço de manipulação do vetor; no entanto, você deve primeiro tentar usar os comandos push ou unshift muito mais simples antes de usar splice() para adicionar a uma matriz.

## Adicionando elementos com concat()

O método concat() retorna uma novo array composto do array original mais a que for passado por parametro no método. Para adicionar alguns elementos a outro array usando concat(), faça o seguinte:

```
1. var list = ["foo", "bar"];
2.
3. var newlist = list.concat(["baz", "qux"]);
4.
5. ["foo", "bar", "baz", "qux"] // resultado da nova lista
```

Também é possível adicionar diretamente valores que não sejam vetores usando concat():

```
1. var list = ["foo", "bar"];
2.
3. var newlist = list.concat("baz", "qux");
4.
5. ["foo", "bar", "baz", "qux"] // resultado da nova lista
```

O método `concat()` é uma solução simples em situações em que você precisa combinar alguns arrays juntos, sem inchar seu código com loops como o “for” ou outros loops iterativos.

## Adicionando elementos usando o índice do vetor

Também podemos manipular diretamente o array, sem o uso de qualquer método, referindo-se à posição do índice dentro do array. Aqui nós adicionamos dois novos elementos em posições especificadas:

```
1. var list = ["foo", "bar"];
2.
3. list[2] = "baz"; // adiciona o elemento "baz" à posição do índice 2 na matriz
4.
5. list[3] = "qux";
6.
7. ["foo", "bar", "baz", "qux"] // resultado
```

Aqui nós adicionamos um elemento ao final do array:

```
1. var list = ["foo", "bar"];
2.
3. list[list.length] = "baz"; // adiciona o elemento "baz" ao final do array
4.
5. ["foo", "bar", "baz"] // resultado
```

O método de notação de índice geralmente é útil quando você sabe exatamente onde colocar um elemento na matriz, independentemente de onde essa posição possa estar. Essa situação geralmente se aplica a muitos tipos de implementações de algoritmos.

## Remover Elementos de um Array

Até agora, vimos como declarar e inicializar um [Array \(vetor\)](#) e como [inserir elementos em um Array](#) usando algoritmos ou métodos. Agora, vamos ver como podemos remover dados de um array.

### Método pop

#### Remover Elemento do Final de Um Array

O método `pop()` remove um elemento do final de um array e retorna o conteúdo do índice removido.

Não precisamos nos preocupar com o tamanho e nem com o último índice do array, pois `pop()` fará todo o trabalho sozinho.

```
1.<SCRIPT LANGUAGE="JavaScript" TYPE="text/javascript">
2.vetor = new Array ("José", "Maria", "João", "Patrícia");
3.document.write ("Este array possui "+vetor.length+" elementos.<br>");
4.document.write ("Vamos remover "+vetor.pop()+" da lista.<br>");
5.document.write ("Agora, temos "+vetor.length+" elementos.<br>"
6.+ "Que são: "+vetor.join(", ");
7.</SCRIPT>
```

Veja o que acontece quando este JavaScript é executado.

Este array possui 4 elementos.  
Vamos remover Patrícia da lista.

Agora, temos 3 elementos.  
Que são: José, Maria, João

## Método shift

### Remover Elemento do Início de Um Array

Outra forma de remover elementos é utilizar o método `shift()`.

O método `shift()` remove um elemento do início de um array e retorna o conteúdo do índice removido.

```
1.<SCRIPT LANGUAGE="JavaScript" TYPE="text/javascript">
2.vetor = new Array ("José", "Maria", "João", "Patrícia");
3.document.write ("Este array possui "+vetor.length+" elementos.<BR>");
4.document.write ("Vamos remover "+vetor.shift()+" da lista.<BR>");
5.document.write ("Agora, temos "+vetor.length+" elementos.<BR>")
6.+ "Que são: "+vetor.join(", ");
7.</SCRIPT>
```

Veja o resultado logo abaixo.

Este array possui 4 elementos.  
Vamos remover José da lista.  
Agora, temos 3 elementos.  
Que são: Maria, João, Patrícia

## Remover Elementos no Meio do Array Algoritmicamente

Como vimos, é possível remover elementos de um Array (vetor) tanto no final quanto no início. Agora, veremos como remover elementos que estão no meio do Array algoritmicamente usando os métodos apresentados até o momento.

Vamos pensar na seguinte situação: Temos um Array que contém todas as cartas do baralho, mas existe uma carta errada chamada R entre o 5 e o 6. Como iremos remover a carta R?

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
A	2	3	4	5	R	6	7	8	9	10	J	Q	K

Primeiro criaremos um loop começando da posição em que se encontra a carta R (posição 5) e iremos percorrer o Array até o penúltimo índice (tamanho-1).

```
1.for (i=5; i<vetor.length-1; i++)
```

Então, iremos atribuir a posição atual o valor da próxima posição. Isso irá sobrescrever o valor errado (R).

```
1.vetor[i] = vetor[i+1];
```

Isso irá resultar no seguinte Array.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
A	2	3	4	5	6	7	8	9	10	J	Q	K	K

Podemos perceber que o valor incorreto (R) não existe mais na posição 5. Porém, temos um valor repetido no final de nosso Array. Para removê-lo, usaremos o método `pop()`.

```
1.vetor.pop();
```

Pronto! Agora temos um Array de cartas de baralho correto. O código completo é demonstrado logo abaixo:

```
1.<SCRIPT LANGUAGE="JavaScript" TYPE="text/javascript">
2.vetor = new Array ("A", "2", "3", "4", "5", "R", "6",
3."7", "8", "9", "10", "J", "Q", "K");
4.for (i=5; i<vetor.length-1; i++) {
5.vetor[i] = vetor[i+1];
6.}
7.vetor.pop()
8.</SCRIPT>
```

Agora temos nosso Array da seguinte maneira na memória:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
A	2	3	4	5	6	7	8	9	10	J	Q	K



A	2	3	4	5	6	7	8	9	10	J	Q	K
---	---	---	---	---	---	---	---	---	----	---	---	---

## Junção e Concatenação de Arrays (Vetores)

Como vimos anteriormente, os vetores em JavaScript são constituídos pelo objeto [Array](#).

Há momentos que queremos unir um ou mais vetores. JavaScript nos possibilita fazer isso resultando duas formas diferentes.

### Método Join

O vetor construído com o objeto Array de JavaScript pode ter a junção de seus elementos.

A junção consiste em criar uma string única usando separadores.

O método `join()` nos possibilita a junção de forma simples, precisa e correta em uma string. Por exemplo, vamos imaginar que temos um vetor que possui os dados de uma data qualquer em seus índices e que queremos mostrar a saída dessa data inteira. Há duas formas que podemos fazer isso e são complicadas.

Uma delas é criar um loop entre os dois primeiros índices e depois mostrar apenas o último índice.

```
01.<script language="JavaScript" type="text/javascript">
02.var data = new Array (3);
03.data[0] = 27;
04.data[1] = 3;
05.data[2] = 2009;
06.for (i=0; i<data.length-1; i++){
07.document.write (data[i] + "/");
08.}
09.document.write (data[2]);
10.</script>
```

A outra forma, apesar de mais simples, ainda não é a mais indicada.

```
1.<script language="JavaScript" type="text/javascript">
2.var data = new Array (3);
3.data[0] = 27;
4.data[1] = 3;
5.data[2] = 2009;
6.document.write (data[0]+"/"+data[1]+"/"+data[2]);
7.</script>
```

Agora, usando o método `join()`, basta unir os elementos usando a barra como separador.

```
1.<script language="JavaScript" type="text/javascript">
2.var data = new Array (3);
3.data[0] = 27;
4.data[1] = 3;
5.data[2] = 2009;
6.document.write (data.join("/"));
7.</script>
```

Vale lembrar que o método `join` não modifica o vetor original, mas é possível guardar seu resultado em uma variável. Ex.: `dataCompleta = data.join("/")`.

### Método Concat

O método `concat()` consiste em unir um ou mais arrays (vetores).

Este método usa como argumento um objeto do tipo array. Se desejarmos unir mais de um vetor, cada objeto array do argumento deve vir separado por vírgula (,).

```
01.<script language="JavaScript" type="text/javascript">
02.var alunosAno1 = new Array ("Maria", "João", "Alexandre");
03.var alunosAno2 = new Array ("Everton", "Cláudia", "Vanessa");
04.var alunosAno3 = new Array ("Junior", "Edgar", "Paulo");
05.var alunosAno4 = new Array ("Regina", "Bárbara", "Juliana");
```

```

06.var todosAlunos = alunosAno1.concat(alunosAno2, alunosAno3,
alunosAno4);
07.document.write ("Esta escola tem " + todosAlunos.length + "
alunos.<br>"
08.+ "Que se chamam:<br>" + todosAlunos);
09.</script>

```

Aqui, o resultado obtido na variável todosAlunos é o nome contido nas quatro variáveis alunosAno.

O que resulta em:

Esta escola tem 12 alunos.

Que se chamam:

Maria,João,Alexandre,Everton,Claudia,Vanessa,Junior,Edgar,Paulo,Regina,Barbara,Juliana

Assim como no método join(), concat() também não altera o conteúdo original dos vetores.

## Organizar Arrays

Depois de vermos como [inserir elementos em um array](#) e como [remover elementos de um array](#), agora veremos como inverter e organizar um array.

### Reverse

O método reverse do objeto array serve para inverter a ordem dos elementos de um array.

Dessa forma podemos usar apenas uma forma de organização e depois inverter o array para conseguir a ordem desejada.

```

1.<SCRIPT LANGUAGE="JavaScript" TYPE="text/javascript">
2.var vetor = Array (1, 2, 3, 4, 5);
3.document.write ("Array inicial: " + vetor);
4.document.write ("<BR>Array invetido: " + vetor.reverse());
5.</SCRIPT>

```

Isso inverterá o array resultando em:

Array inicial: 1,2,3,4,5  
Array invetido: 5,4,3,2,1

### Sort

O método sort é o que realmente faz a organização do array. Porém, sort apenas organiza o array de forma alfabética, se quisermos organizar o array de forma numérica devemos criar uma função para indicar como parâmetro de sort.

Abaixo está um exemplo de uma organização simples feita em ordem alfabética.

```

1.<SCRIPT="JavaScript" TYPE="text/javascript">
2.var vetor = Array ("João", "Maria", "José", "Pedro");
3.document.write (vetor.sort());
4.</SCRIPT>

```

Veja que *vetor* é organizado corretamente de forma alfabética.

José,João,Maria,Pedro

Agora, podemos perceber que o resultado da organização de números não é dada de forma satisfatória.

```

1.<SCRIPT LANGUAGE="JavaScript" TYPE="text/javascript">
2.var vetor = Array (3000, 20, 100, 4);
3.document.write (vetor.sort());
4.</SCRIPT>

```

100,20,3000,4

Para organizar um array numérico usando o método sort devemos criar uma função com dois parâmetros que retorne um valor negativo ou positivo. Esse valor, servirá como base para o método sort determinar qual índice do array virá primeiro.

Para organizar a variável *vetor* criaremos duas funções: uma chamada crescente e outra chamada decrescente.

```

01.<SCRIPT LANGUAGE="JavaScript" TYPE="text/javascript">
02.function crescente (index1, index2){

```

```

03.return index1 - index2;
04.}
05.
06.function decrescente (index1, index2){
07.return index2 - index1;
08.}
09.
10.var vetorOriginal = Array (3000, 20, 100, 4);
11.var vetorCrescente = vetorOriginal; // copiando um vetor para o
outro
12.var vetorDecrescente = vetorOriginal; // copiando um vetor para o
outro
13.document.write ("Vetor em ordem crescente:<BR>");
14.document.write (vetorCrescente.sort(crescente));
15.document.write ("<BR>Vetor em ordem decrescente<BR>");
16.document.write (vetorDecrescente.sort(decrescente));
17.</SCRIPT>

```

O resultado é mostrado abaixo:

```

Vetor em ordem crescente:
4,20,100,3000
Vetor em ordem decrescente
3000,100,20,4

```

## TRABALHANDO COM ARQUIVOS - REQUIRE(FS)

A maneira de se trabalhar com arquivos externos no node é o uso do file system.

Para usar este módulo temos que importar o ('fs'). Todos os métodos têm formas síncronas e assíncronas.

### Abrindo e fechando um arquivo

```

const fs = require('fs');
fs.open('novo.txt', 'r', (err, fd) => {
  if (err) throw err;
  fs.close(fd, (err) => {
    if (err) throw err;
  });
});

```

### Lendo um arquivo.

O primeiro método que falarei é o readFile, que serve para ler um arquivo.

```

var fs = require('fs');
fs.readFile('teste.txt', 'utf-8', function (err, data) {
  if(err) throw err;
  console.log(data);
});

fs.readFile('novo.txt', 'utf-8', function(err, data){
  var linhas = data.split(/\r?\n/);
  linhas.forEach(function(linha) {
    console.log(linha); // aqui testar cada linha
  })

```

```
}}
```

Na primeira linha eu criei uma variável `fs` atribui para ela o módulo `'fs'`(File System).

Dentro do File System temos o método `readFile`, que recebe 3 parâmetros, (arquivo, parâmetros opcionais, função callback).

No primeiro parâmetro eu passei o meu caminho com o arquivo que eu quero que seja lido, o segundo parâmetro é a codificação do arquivo e o terceiro é a função callback, que recebe também dois parâmetros, um de erro e um de dados, este ultimo é o valor do arquivo que quero ler.

## Escrevendo em um arquivo.

Para escrever em um arquivo também não existem mistérios, utilizamos o método `writeFile`.

```
fs.writeFile('teste.txt', 'Hello World!\n', {encoding: 'utf-8', flag:
'a'}, function (err) {
  if (err) throw err;
  console.log('Arquivo salvo!');
});
```

Este método assim como o anterior também recebe o caminho do arquivo, porém agora recebe o que iremos escrever como segundo parâmetro, um terceiro parâmetro que é opcional, e a função callback.

Como terceiro parâmetro passei um objeto com a codificação que desejo e com a forma que quero que seja escrito o arquivo, na mensagem passei um `\n` ao final do Hello World, para que toda vez que escreva no arquivo comece na próxima linha.

### Flags para operações de leitura / gravação são:

#### FLAG DESCRIÇÃO

- r** Abre o arquivo para leitura. Uma exceção ocorre se o arquivo não existe.
- r+** Abre o arquivo para leitura e escrita. Uma exceção ocorre se o arquivo não existe.
- rs** Arquivo aberto para leitura no modo síncrono.
- rs+** Arquivo aberto para leitura e escrita, contando a OS para abri-lo de forma síncrona.
- w** Abre o arquivo para escrita. O arquivo é criado (se não existir) ou truncado (se existir).
- wx** Como `'w'`, mas não consegue se existe caminho.
- w+** Abre o arquivo para leitura e escrita. O arquivo é criado (se não existir) ou truncado (se existir).
- wx+** Como `'w+'`, mas não consegue se existe caminho.
- a** Abre o arquivo para acrescentar. O arquivo é criado se ele não existe.
- ax** Como `'a'`, mas não consegue se existe caminho.
- a+** Abre o arquivo para leitura e acrescentando. O arquivo é criado se ele não existe.
- ax+** Como `'a+'`, mas não consegue se existe caminho.

## Excluir o arquivo

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

fs.unlink('teste.txt', (err) => {
  if (err) throw err;
  console.log('arquivo removido com sucesso');
});
```

## Validar a exclusão

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

try {
  fs.unlinkSync('teste2.txt');
  console.log('arquivo removido com sucesso');
} catch (err) {
  console.log('arquivo não encontrado');
}
```

## Renomear o arquivo

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

fs.rename('teste.txt', 'novo.txt', (err) => {
  if (err) throw err;
  console.log('nome de arquivo alterado');
});
```

## Copiar o arquivo

```
const fs = require('fs')
const readableStream = fs.createReadStream('novo.txt');
var writableStream = fs.createWriteStream('copy.txt');
readableStream.pipe(writableStream);
```

## Verificar se existe o arquivo

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

fs.readFile('novo.txt', 'utf8', function(err,data){
  if(err) {
    console.error("Arquivo nao encontrado: %s", err);
    process.exit(1);
  }

  console.log(data);
});
```

## Verificar se existe a pasta

```
// Carregando o File System
var fs = require("fs");
// Lê o conteúdo do diretório retornando um array de string de arquivos.
// Obs.: Essa leitura é Não-Bloqueante, por isso retorna via callback.
fs.readdir("home", function(err, files){
  console.log(files);
});

// A mesma função, executada de forma Bloqueante.
var files = fs.readdirSync("home");
```

```
console.log(files);
```

## Lendo o conteúdo dos arquivos

```
let arquivo = []; // Vetor para armazenar todos os nomes dos arquivos lidos

// Lendo todos os arquivos existentes na pasta files de forma síncrona
fs.readdirSync('home').forEach(arquivo => {
  // Efetuando a leitura do arquivo
  fs.readFile('home/' + arquivo, 'utf8', function(err, data) {
    // Enviando para o console o resultado da leitura
    console.log(data);
  });
});
```

## Callbacks em Node

Callbacks são funções que serão executadas de modo assíncrono, ou posteriormente. Ao passo que o código for lido de cima para baixo de modo processual, programas assíncronos possivelmente executam funções em tempos diferentes baseado na ordem e velocidade em que requisições http ou o trabalho de arquivamento do sistema acontecem.

A diferença pode ser confusa sendo determinada quando uma função é assíncrona ou não depende da execução de um grande contexto. Seque um simples exemplo de código assíncrono:

```
var myNumber = 1
function addOne() { myNumber++ } // define a função
addOne() // executa a função
console.log(myNumber) // mostra na saída padrão o número 2
```

O código acima define uma função e na linha seguinte chama essa função, sem esperar por nada. Onde ela é chamada imediatamente adicionando 1 à variável **myNumber**, então depois que a função foi chamada você espera que o número seja 2 na variável **myNumber**. Supondo que precisamos armazenar nosso número em um arquivo chamado **number.txt**:

```
var fs = require('fs') // require é uma função especial do node para requisitar os módulos
var myNumber = undefined // não sabemos ainda qual valor está armazenado no arquivo
```

```
function addOne() {
  fs.readFile('./number.txt', function doneReading(err, fileContents) {
    myNumber = parseInt(fileContents)
    myNumber++
  })
}
```

```
addOne()
```

```
console.log(myNumber) // mostra na saída padrão `undefined`
```

Porque é que temos **undefined** quando pedimos para mostrar o número dessa vez? Nesse código usamos o método **fs.readFile**, que acontece de modo assíncrono. Usualmente você tem que se comunicar com os discos rígidos ou redes bem de modo assíncrono. Se for somente para acesso à memória ou fazer algo na CPU isso é feito bem de modo síncrono.

A razão pela qual fazer o I/O assíncrono é porque o acesso ao disco é 100,000 vezes mais devagar do que comunicar-se com a memória (RAM).

Onde você executa um programa onde as funções são imediatamente definidas, mas não executa elas imediatamente. Este é um conceito fundamental para entender sobre a programação assíncrona. Onde `addOne` é chamada fora de `fs.readFile` e move-se para a próxima tarefa pronta para ser executada e não esperar o disco responder igual a `fs.readFile`. Se não tem nada para executar o node espera por pendências de operações fs/network terminarem ou pararem de executar saindo da linha de comando.

Onde `fs.readFile` esta completa para ler o arquivo (talvez isso demore milissegundos ou segundos e até mesmo minutos dependendo de quanto o disco é rápido) executando a função `doneReading` e dando um erro (claro que se algum tipo de erro acontecer) e o conteúdo do arquivo.

A razão pela qual `undefined` foi mostrado no código acima é que ele é chamado dentro de `console.log` e fora de `fs.readFile` mostrando o valor anterior de `myNumber` e não o conteúdo do arquivo.

Se você tem um pedaço de código que precisa ser executado várias e várias vezes ou um tempo depois, o primeiro passo é colocar esse pedaço de código dentro de uma função. Aonde você podera chamar sem precisar escrever ele em todas as partes que for necessário e nomeando da maneira que fique claro aquilo que esta sendo feito. Isso ajuda a dar nomes descritivos para as funções.

Callbacks são somente funções que são executadas de forma tardia. A chave para a compreensão de callbacks é perceber que eles são utilizados quando você não sabe **quando** alguma operação assíncrona estará completa, mas você sabe **quando** se completará – a ultima linha da função assíncrona! A ordem de cima-para-baixo que você declara para callbacks isso não é necessariamente importante, somente a ordem lógica/hierárquica de assentamento do código. Primeiro divida o código em funções e use callbacks para declarar se uma depende da outra função para encerrar.

O método `fs.readFile` é fornecido pelo node no módulo `fs`, é assíncrono e acontece quando se tem um tempo curto para finalizar. Considerando o que ele faz: ele vai até o sistema operacional, que por sua vez esta rodando em um disco rígido isso pode ou não estar girando a milhares de rotações por minuto. Então ele tem que usar o laser para ler os dados e enviar através das camadas de comunicação do sistema de volta para o seu programa em javascript. Você da ao `fs.readFile` uma função (conhecida como callback) que sera chamada depois de recuperar os dados do sistema de arquivos. Ela coloca os dados recuperados em uma variavel do javascript e passa para sua função (callback) com essa variável, neste caso a variavel se chama `fileContents` porque ela contém o conteúdo do arquivo que foi lido.

Pense no restaurante do exemplo do tutorial lá do inicio. Em muitos restaurantes você pega um número e coloca em sua mesa e espera por sua comida. Eles são como os callbacks. Isso deixa claro para o servidor a quem chamar quando o cheeseburger estiver pronto.

Colocando `console.log` em uma função e passando ele para um callback.

```
var fs = require('fs')
var myNumber = undefined

function addOne(callback) {
  fs.readFile('./number.txt', function doneReading(err, fileContents) {
    myNumber = parseInt(fileContents)
    myNumber++
    callback()
  })
}

function logMyNumber() {
  console.log(myNumber)
```

```
}
```

```
addOne(logMyNumber)
```

Agora a função `logMyNumber` que é passada como argumento se torna a variável `callback` dentro da função `addOne`. Depois que `fs.readFile` terminar a variável `callback` é chamada (`callback()`). Somente funções podem ser chamadas, então se você passar qualquer outra coisa diferente de uma função, causará um erro. Onde uma função é chamada no javascript o código dentro dessa função é imediatamente executado. Neste caso nosso `log` é executado onde `callback` é atualmente a função `logMyNumber`. Lembre-se, somente se você *definiu* uma função não significa que ela será executada. Você tem que *chamar* uma função para ela acontecer. Para quebrar esse exemplo em mais pedaços, aqui tem uma linha do tempo de eventos que acontecem quando o seu programa é executado:

- 1: o código é analisado, isso significa que se existir algum erro de sintaxe o programa quebrará e será apontado aonde isso aconteceu.
- 2: `addOne` será chamado, onde `logMyNumber` será passado com uma função chamada `callback`, que é o que precisa ser chamado quando `addOne` estiver completa. Imediatamente disparando o método assíncrono `fs.readFile`. Essa parte do programa leva um tempo para terminar.
- 3: com nada para fazer, o node espera por um tempo até o `fs.readFile` encerrar a sua execução.
- 4: `fs.readFile` termina e chama o callback, `doneReading`, que incrementa o número e imediatamente chama a função `logMyNumber` contida na variável `callback`.

Talvez a parte mais confusa de se programar com callbacks é que as funções são somente objetos armazenados em variáveis e passadas em todo o programa com diferentes nomes. Dando nomes simples e descritivos para suas variáveis faz seu código ser mais legível para outros. Geralmente falando em programas no node onde você enxerga uma variável como `callback` ou `cb` você assume ela como uma função.

Você talvez tenha escutado alguns termos como `programação evencionada` ou `ciclo de eventos`. Onde é referenciado da mesma maneira que `fs.readFile` foi implementada. Node primeiramente despacha a operação `fs.readFile` e espera por `fs.readFile` enviar um evento para concluir. Enquanto a resposta é esperada o node vai buscando checar outras coisas. Dentro do node há uma lista de coisas a serem feitas mas não informaram ainda, então o ciclo do node acaba e retorna para a lista várias vezes checando se o que estava sendo processado terminou. Depois do término ele pega o que foi 'processado', ex. callbacks que dependem desse término são chamados.

Aqui temos uma versão de um pseudocódigo do exemplo acima:

```
function addOne(thenRunThisFunction) {  
  waitAMinute(function waitedAMinute() {  
    thenRunThisFunction()  
  })  
}
```

```
addOne(function thisGetsRunAfterAddOneFinishes() {})
```

Imagine que tenha 3 funções assíncronas `a`, `b` e `c`. Para cada uma leva-se 1 minuto de execução e depois de terminado elas chamam um callback (que é passado como primeiro argumento). Se você tem que falar para o node 'comece executando a, depois b depois que a terminar, e executar c então b termina' isso passa a ser:

```
a(function() {  
  b(function() {  
    c()  
  })  
})
```



Onde esse código será executado, **a** é imediatamente executado, um minuto depois que ele terminar e chama **b**, outro minuto depois que ele terminar e chamar **c** e finalmente depois que 3 minutos se passaram o node para de executar desde que não tenha mais nada para fazer. Isso é definitivamente a maneira mais elegante para escrever o código acima, mas o ponto é esse, se você tem esse código que espera por uma porção de código assíncrono terminar, onde é expressado essas dependências colocando seu código em funções, isso é passado às outras funções como callbacks. A projeção do node requer que você pense de modo não-linear. Considerando essa lista de operações:

```
ler um arquivo
processar um arquivo
```

Se você ingênuamente transforma-se isso em um pseudocódigo você teria este resultado:

```
var file = readFile()
processFile(file)
```

Esse tipo de linearidade no código (passo-a-passo, em ordem) não é o modo como o node trabalha. Se esse código fosse executado onde **readFile** e **processFile** estão sendo chamados ao mesmo tempo. Não faria o menor sentido porque **readFile** leva um tempo para completar sua execução. Ao passo que você precisa expressar **processFile** que depende do **readFile** completo. Esta é a exata finalidade dos callbacks! E por causa da forma que o JavaScript trabalha você pode escrever dependências de diferentes maneiras:

```
var fs = require('fs')
fs.readFile('movie.mp4', finishedReading)

function finishedReading(error, movieData) {
  if (error) return console.error(error)
  // faça algo com os dados em movieData
}
```

Mas você também pode estruturar o código dessa maneira que irá funcionar:

```
var fs = require('fs')

function finishedReading(error, movieData) {
  if (error) return console.error(error)
  // faça algo com os dados em movieData
}

fs.readFile('movie.mp4', finishedReading)
```

Ou até mesmo assim:

```
var fs = require('fs')

fs.readFile('movie.mp4', function finishedReading(error, movieData) {
  if (error) return console.error(error)
  // faça algo com os dados em movieData
})
```

## Criando uma aplicação WEB com NodeJS usando o servidor HTTP

A função de cada parte do código para criar um servidor http:

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

**Nota:** Existem outros módulos que criam servidores como é caso de "net", "tcp" e "tls".

A partir deste momento temos uma variável http que na realidade é um objeto, sobre o que podemos invocar métodos que estavam no módulo requerido. Por exemplo, uma das tarefas implementadas no módulo HTTP é a de criar um servidor, que se faz com o módulo "createServer()". Este método receberá um *callback* que será executado cada vez que o servidor receba uma solicitação.

```
var server = http.createServer(function (peticao, resposta){  
  resposta.end("Ola CriarWeb.com");  
});
```

A função callback que enviamos a createServer() recebe dois parâmetros que são a solicitação e a resposta. Não usamos a solicitação por agora, mas contém dados da solicitação realizada. Usaremos a resposta para enviar dados ao cliente que fez a solicitação. De modo que "resposta.end()" serve para terminar a solicitação e enviar os dados ao cliente. Agora vou dizer ao servidor que se ponha em funcionamento porque até o momento só criamos o servidor e escrevemos o código a ser executado quando se produza uma solicitação, mas não o iniciamos.

```
server.listen(3000, function(){  
  console.log("seu servidor está pronto em " + this.address().port);  
});
```

Com isto dizemos ao servidor que escute no porto 3000, embora pudéssemos ter posto qualquer outro porto que gostássemos. Ademais "listen()" recebe também um função callback que realmente não seria necessária, mas que nos serve para fazer coisas quando o servidor tenha sido iniciado e esteja pronto. Simplesmente, nessa função callback indico que estou pronto e escutando no porto configurado.

Código completo de servidor HTTP em node.JS

Como você pode ver, em muitas poucas linhas de código geramos um servidor web que está escutando em um porto dado. O código completo é o seguinte:

```
var http = require("http");
var server = http.createServer(function (request, response){
  response.end("Ola CriarWeb.com");
});
server.listen(3000, function(){
  console.log("seu servidor está pronto em " + this.address().port);
});
```

## Colocar em execução o arquivo com Node.JS para iniciar o servidor

Agora podemos executar com Node o arquivo que criamos. Vamos da linha de comandos à pasta onde salvamos o arquivo servidor.js e executamos o comando "node" seguido do nome do arquivo que pretendemos executar:

```
node servidor.js
```

Então no console de comandos nos deve aparecer a mensagem que informa que nosso servidor está escutando no porto 3000.

O modo de comprovar se realmente o servidor está escutando a solicitações de clientes no tal porto é acessar com um navegador. Deixamos ativa essa janela de linha de comandos e abrimos o navegador.

Acessamos a:

```
http://localhost:3000
```

## Elementos naturais de HTTP do Node.js

O módulo http do Node.js fornece funções úteis e classes para construir um servidor HTTP. Ele é um módulo-chave para os recursos de rede do Node.

Ele pode ser facilmente incluído em um módulo Node.js usando:

```
const http = require('http')
```

O objeto http declarado ali possui algumas propriedades e métodos, além e algumas classes.

### http.METHODS

Esta 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**

```
> 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.createServer()**

Retorna uma nova instância da classe `http.Server` e seu uso é muito simples:

```
const server = http.createServer((req, res) => {
  //cada requisição recebida dispara este callback
})
```

### **http.request()**

Realiza uma requisição HTTP para um servidor, criando uma instância da classe `http.ClientRequest`.

### **http.get()**

Similar ao `http.request()`, mas automaticamente define o método HTTP como GET e já finaliza a requisição com `req.end()` automaticamente.

O módulo HTTP também fornece cinco classes:

- `http.Agent`
- `http.ClientRequest`

- `http.Server`
- `http.ServerResponse`
- `http.IncomingMessage`

Um objeto **`http.ClientRequest`**, por exemplo, é criado quando chamamos `http.request()` ou `http.get()`. Quando uma resposta é recebida o evento `response` é chamado com a resposta, passando uma instância de **`http.IncomingMessage`** como argumento.

Os dados retornados por uma resposta podem ser lidos de duas maneiras:

- Você pode chamar o método `response.read()`
- No event handler `response` você pode configurar um listener para o evento `data`, assim você pode receber os dados em formato de stream (bytes)
- Já a classe **`http.Server`** é comumente instanciada e retornada quando criamos um novo servidor usando `http.createServer()`. Uma vez que você tenha um objeto `server`, você pode acessar seus métodos:
- **`close()`** que encerra as atividades do servidor, não aceitando mais novas requisições;
- **`listen()`** que inicia o servidor HTTP e espera novas conexões;
- A classe **`http.ServerResponse`** é muito utilizada como argumento nos callbacks que tratam a resposta de requisições, a famosa variável `'res'` presente em muitos callbacks, como abaixo:
- É importante sempre salientar que após utilizarmos o objeto `res` devemos chamar o método `end()` para fechar a resposta e enviar a mesma para o cliente que fez a requisição.
- Objetos `res/response` possuem alguns métodos para interagir com os cabeçalhos HTTP:
- `getHeaderNames()` traz a lista de nomes dos header presentes na resposta
- `getHeaders()` traz uma cópia dos headers presentes
- `setHeader('nome', valor)` altera o valor de um header
- `getHeader('nome')` retorna o valor de um header
- `removeHeader('nome')` remove um header
- `hasHeader('nome')` retorna true se a response possui este header
- `headersSent()` retorna true se os headers já foram enviados ao cliente

Depois de fazer as alterações que desejar nos cabeçalhos do `response`, você pode enviá-los ao cliente usando `response.writeHead()`, que aceita o `statusCode` como o primeiro parâmetro, a mensagem opcional e os cabeçalhos.

```
response.statusCode = 500
response.statusMessage = 'Internal Server Error'
```

Ok, mas nem só de cabeçalhos vive a resposta, certo? Para enviar dados ao cliente no corpo da requisição, você usa `write()`. Ele vai enviar dados bufferizados para a stream de resposta.

E por fim, a classe **`http.IncomingMessage`** é criada nas requisições.

Node.js é uma plataforma multiprotocolo, ou seja, com ele será possível trabalhar com HTTP, DNS, TCP, WebSockets e muito mais. Porém um dos protocolos mais usados para desenvolver sistemas web é o protocolo HTTP, de fato é o protocolo com a maior quantidade de módulos disponíveis para trabalhar no Node.js.

Hoje apresentarei um pouco sobre como desenvolver uma aplicação HTTP, na prática desenvolveremos um simples sistema web utilizando o módulo nativo HTTP e também apresentando alguns módulos mais estruturados para desenvolver aplicações complexas.

Toda aplicação web necessita de um servidor web em execução para disponibilizar todos os seus recursos, na prática você irá desenvolver uma **aplicação servidora**, ou seja, além de programar todas funcionalidades da sua aplicação você também terá que configurar na própria aplicação aspectos sobre como **ela servirar seus recursos para o cliente** quando for executá-la. Essas **configurações são conhecidas como middleware**, é claro que é um trabalho dobrado no começo, mas isso traz a liberdade de configurar cada mínimo detalhe do sistema, ou seja, permite desenvolver algo mais performático e controlado pelo programador. Caso performance não seja prioridade no desenvolvimento do seu sistema, recomendo que utilize alguns módulos famosos que já vem com o mínimo necessário de configurações prontas para não perder tempo trabalhando sobre esse aspecto, alguns módulos conhecidos são: [Connect](#), [Express](#), [Geddy](#) e [muito mais aqui](#). Esses módulos já são preparados para trabalhar desde uma **infraestrutura mínima e básica (Microframeworks)** até uma **infraestrutura mais enxuta com padrões do tipo MVC (Model-View-Controller)** e outros padrões de projetos (MVC Frameworks).

### Módulo nativo HTTP:

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

var server = http.createServer(function(request, response){

  response.writeHead(200, {"Content-Type": "text/html"});

  response.write("<html><body><h1>Olá Node.js!</h1></body></html>");

  response.end();

});

server.listen(3000, function(){

  console.log("Executando Servidor HTTP");

});
```

Esse é um exemplo clássico e simples de um servidor web sendo executado na **porta 3000**, respondendo por padrão na **rota raiz "/"** um resultado em **formato html** com a mensagem **Olá Node.js!**.

Agora complicando mais, vamos adicionar duas rotas nesse sistema, uma rota para página de erro e também um link em cada página html para intergir uma com a outra:

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

var server = http.createServer(function(request, response){

  response.writeHead(200, {"Content-Type": "text/html"});

  if(request.url == "/"){

    response.write("<html><body><h1>Olá Node.js!</h1>");

    response.write("<a href='/bemvindo'>Bem vindo</a>");
```

```

    response.write("</body></html>");
} else if (request.url == "/bemvindo"){
    response.write("<html><body><h1>Bem-vindo ao Node.js!</h1>");
    response.write("<a href='/'>Olá Node.js</a>");
    response.write("</body></html>");
} else{
    response.write("<html><body><h1>Página não encontrada!</h1>");
    response.write("<a href='/'>Voltar para o início</a>");
    response.write("</body></html>");
}
response.end();
});
server.listen(3000, function(){
    console.log('Executando Servidor HTTP');
});

```

Toda leitura de url é obtida através do método **request.url** que retorna uma string sobre o que foi digitado no endereço url do seu browser. Endereços urls do protocolo http possui alguns padrões como **query strings** (**?nome=joao**) e **pathnames** (**/admin**) e sinceramente tratar toda string url seria trabalhoso demais, sendo que já existem diversos exemplos prontos para isso. No Node.js existe o **módulo chamado url** responsável por realizar um **parser e formatação de strings url**, veja como um exemplo abaixo:

```

var http = require('http');
var url = require('url');
var server = http.createServer(function(request, response){
    // Faz um parse da string url digitada.
    var result = url.parse(request.url, true);
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("<html><body>");
    response.write("<h1>Dados da query string</h1>");

```



```
// Itera o resultado de parâmetros passados via query string.
for(var key in result.query){
    response.write("<h2>" + key + " : " + result.query[key] + "</h2>");
}
response.write("</body></html>");
response.end();
});
server.listen(3000, function(){
    console.log('Executando Servidor HTTP');
});
```

Digite no seu browser a url: <http://localhost:3000/?nome=joao&idade=22&email=joao@mail.net> para ver os resultados tratados pelo **parse de url**.

Agora vamos separar o código HTML do código Node.js em arquivos distintos, para isso utilizaremos o **módulo nativo FS (File System)** que faz tratamento de arquivos, que no nosso caso será leitura de arquivo HTML.

```
// app.js
var http = require('http');
var fs = require('fs');
var server = http.createServer(function(request, response){
    fs.readFile(__dirname + '/index.html', function(err, html){
        response.writeHead(200, {'Content-Type': 'text/html'});
        response.write(html);
        response.end();
    });
});
server.listen(3000, function(){
    console.log('Executando Servidor HTTP');
});
<!-- index.html -->
<!DOCTYPE html>
<html>
<head>
```

```
<title>Hello World</title>

</head>

<body>

  <h1>Hello World</h1>

</body>

</html>
```

Neste código temos dois detalhes interessantes a citar, primeiro é a constante global chamada **\_\_dirname** que retorna uma string referente ao endereço raiz da aplicação, é uma variável muito útil, pois através dela podemos referenciar pastas e arquivos internos.

Outro detalhe é o método **fs.readFile()**, repare que o resultado da leitura do **arquivo index.html** é enviado via **função de callback**, ou seja através da **function(erro, html)** **que é um parâmetro da função fs.readFile()**, na prática diversos módulos trabalham dessa forma no Node.js, pois o retorno de resultados através de funções callbacks são tratados de forma assíncrona (**característica principal do Javascript**) e isso é algo muito interessante pois permite tratar a execução das rotinas da aplicação de forma paralela, e isso você usará frequentemente no Node.js. Alguns módulos apresentam em suas documentações duas alternativas de trabalhar com uma mesma função, são elas conhecidas como **execução síncrona e assíncrona**. Sempre que puder **utilize as versões assíncronas** em seus projetos, pois as **execuções assíncronas são mais performáticas** e normalmente **não bloqueiam** a execução de outras **rotinas síncronas** do seu sistema, mesmo quando ocorrem problemas durante suas execuções.

## Padrões de Desenvolvimento em Node

### *Javascript*

Por padrão **Node.js** compila código **Javascript** que é a sua **DSL nativa**. É claro que esse **Javascript** vem acompanhado com uma vasta lista de APIs e adaptações preparadas para trabalhar no server-side de uma aplicação. Abaixo segue um exemplo de um servidor HTTP apresentando o clássico **Hello World**:

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

var server = http.createServer(

  function (req, res) {

    res.writeHead(200, {'Content-Type': 'text/plain'});

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

  }

);

server.listen(4000);

console.log('Hello World executando em http://localhost:4000');
```

## CoffeeScript

Se você adora trabalhar com uma sintaxe mais enxuta, inspirada nas linguagens **Ruby e Python**, com certeza **CoffeeScript** será ideal para você. Ele na prática apenas compila e traduz código **CoffeeScript** para **Javascript** no final. É uma boa escolha nos casos em que você precisa desenvolver aplicações grandes e complexas, dando preferência em trabalhar de forma organizada e com diversas funcionalidades extras que visam facilitar o seu dia-a-dia de programador, tanto na produtividade quanto manutenção de código. Veja abaixo o mesmo exemplo de **Hello World** em versão **CoffeeScript**:

```
http = require 'http'

http.createServer (req, res) ->

  res.writeHead 200, 'Content-Type': 'text/plain'

  res.end 'Hello, World!'

.listen 4000

console.log 'Hello World executando em http://localhost:4000'
```

## HaxeNode

Confesso que essa DSL me surpreendeu, totalmente nova e veio com intuito de cativar os programadores do **Java, C++ e C#**, pois o objetivo do **HaxeNode** é fornecer as principais características do **paradigma orientado à objetos** para o Node.js.

De fato, com ele você conseguirá utilizar **Generics, Tipagem forte de variáveis, Enumerators, declaração e pacotamento de classes, Iterators, Classes Inline e Interfaces**. Para finalizar segue abaixo um exemplo de Hello World feito com HaxeNode:

```
import js.Node;

class Hello {

  public static function main() {

    var server = Node.http.createServer(function(req:NodeHttpServerReq, res:NodeHttpServerResp) {

      res.setHeader("Content-Type","text/plain");

      res.writeHead(200);

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

    });

    server.listen(4000,"localhost");

    trace('Hello World executando em http://127.0.0.1:1337/');

  }

}
```

# Módulos essenciais para Node.js

## *Connect*

Ele é criado em cima do [módulo http](#) nativo do Node.js, com ele será possível criar sistemas com alto nível de configurações de servidor. Ele é base em diversos módulos para **desenvolvimento Web MVC**. Ele é recomendado para o desenvolvimento de projetos pequenos, que utilizam poucas rotas, ou projetos que utilizam diversos middlewares de configuração do servidor, por exemplo: **Logger, Gzip, JSON Parser, Session, Cookie, Static Cache** e muito mais. Um bom exemplo são os sistemas **single-page**, onde o foco de interação do sistema é centralizado em uma única página dinâmica ou um servidor de widgets. Site: <http://www.senchalabs.org/connect>

## *Express*

Ele utiliza o Connect por trás dos panos. Por isso é possível utilizar todo o poder do **Connect**, junto ao um roteador de url robusto. Sua simplicidade é muito semelhante ao [Sinatra do Ruby](#) e existem diversos projetos na web feitos por ele ([MySpaces](#), [LearnBoost](#), [Storify](#), [TreinoSmart](#) e [outros](#)). Ele é um framework ideal para criação de projetos pequeno à grande porte, e toda a organização dos códigos fica a critério do desenvolvedor. Ou seja, você pode montar um projeto em **MVC (Model-View-Controller)**, **MVR (Model-View-Routes)**, **Single-page** e entre outros patterns. Site: <http://expressjs.com>

## *Sails*

Ele é mais novo framework MVC lançado para Node.js, sua estrutura e filosofia de desenvolvimento foi inspirado pelo [Rails](#), porém ele é focado no desenvolvimento de **APIs e comunicação real-time**, ou seja, suas rotas retornam tanto resultados no formato JSON (típicos de **API RESTful**) como também são acessadas via **WebSockets** permitindo uma comunicação bi-direcional com servidor. Ele apesar de ser novo, é perfeito para desenvolver projetos de médio a grande porte. Site: <http://sailsjs.org>

## *Meteor*

Este é um framework totalmente inovador focado no desenvolvimento de sistemas real-time. Ele utiliza o **Express + Socket.IO + Mongoose** internamente, permitindo o acesso direto ao banco de dados tanto no back-end como no front-end. Com apenas 3 arquivos (**JS/HTML/CSS**) já é possível criar uma mini aplicação 100% real-time nesta plataforma. Recomendado para o desenvolvimento de aplicações single-page que necessitam de interações em tempo real com o usuário. Site: <http://meteor.com>

## *Socket.IO*

Com a popularização do **HTML5 WebSockets** tornou-se viável criar sistemas real-time. Porém o grande problema é que não são todos os browsers compatíveis com esta tecnologia. Foi a partir deste problema que nasceu o Socket.IO, um módulo **cross-browser** que permite comunicar-se entre cliente e servidor em real-time de uma forma mágica! Além do WebSockets, ele possui outros **transporters**: **FlashSocket, Ajax Long Polling, Forever Iframe, JSONP Polling**. Estes recursos visam manter uma comunicação real-time quando o browser não possuir nativamente o WebSockets, garantindo que até no **Internet Explorer 6** tenha interação em tempo real. Site: <http://socket.io>

## Engine.IO

Este é a versão minimalista do **Socket.IO**. Sua interface foi projetada para ser o mais próximo do padrão **WebSockets** do **HTML5**. Ele possui apenas 3 **transporters**: **WebSockets**, **FlashSockets** e **JSONP Long Polling**. Este é o mais recente módulo real-time para Node.js criado pelos mesmos desenvolvedores do Socket.IO. Ele é um módulo **leve e de baixo nível**. Uma boa comparação é que assim como **Connect** esta para o **Express**, o **Engine.IO** esta para o **Socket.IO**. Site: <https://github.com/LearnBoost/engine.io>

## Underscore

Este módulo é o famoso **kit de ferramentas para o Node.js**. Com ele é possível manipular estruturas complexas com muita facilidade. Sua documentação tem diversas funções que fazem manipulações milagrosas com Arrays e objetos JSON. É um módulo recomendado para estar presente em qualquer projeto que exija manipulações complexas de estrutura de dados. Ele é compatível tanto no **Node.js** como no **Javascript client-side**. Site: <http://underscorejs.org>

## Moment

Se você precisa trabalhar com **parsers, validações e formatações de datas** ou até mesmo lidar com manipulações temporais. Este é o módulo perfeito para ti! Ele faz possui uma interface muito simples para lidar com o objeto Date do Javascript. Assim como o Underscore, ele é compatível tanto no **Node.js** como **Javascript client-side**. Esta é uma biblioteca recomendada para qualquer tipo de sistemas, afinal quem nunca precisou apresentar de forma amigável uma data no sistema? Site: <http://momentjs.com>

# NPM

Node Package Manager, sua instalação já vem quando se instala o [Node.js](#) e utilizá-lo é muito simples.

Obs.: Por padrão cada módulo é instalado em modo local, ou seja, é apenas inserido dentro do diretório atual do projeto. Caso queria instalar módulos em módulo global apenas inclua o parâmetro **-g** em cada comando.

Por exemplo: **npm install -g nome\_do\_módulo**

Abaixo mostrarei alguns comandos básicos e essenciais para você sobreviver gerenciando módulos em um projeto node:

- **npm install nome\_do\_módulo**: instala um módulo no projeto.
- **npm install nome\_do\_módulo --save**: instala o módulo e adiciona-o na lista de dependências do **package.json** do projeto.
- **npm list**: lista todos os módulos existentes no projeto.
- **npm list -g**: lista todos os módulos globais.
- **npm remove nome\_do\_módulo**: desinstala um módulo do projeto.
- **npm update nome\_do\_módulo**: atualiza a versão do módulo.
- **npm -v**: exibe a versão atual do npm.
- **npm adduser nome\_do\_usuario**: cria um usuário no site <https://npmjs.org> para publicar seu módulo na internet.
- **npm whoami**: exibe detalhes do seu perfil público do npm (é necessário criar um usuário com o comando anterior).
- **npm publish**: publica o seu módulo, é necessário ter uma conta ativa no <https://npmjs.org>.

# Package Express

## Instalação

Assumindo que já tenha instalado o [Node.js](#), crie um diretório para conter o seu aplicativo, e torne-o seu diretório ativo.

Use o comando `npm init` para criar um arquivo `package.json` para o seu aplicativo. Para obter mais informações sobre como o `package.json` funciona, consulte [Detalhes do tratamento de package.json do npm](#).

```
$ npm init
```

Este comando solicita por várias coisas, como o nome e versão do seu aplicativo. Por enquanto, é possível simplesmente pressionar RETURN para aceitar os padrões para a maioria deles, com as seguintes exceções:

```
entry point: (index.js)
```

Insira `app.js`, ou qualquer nome que deseje para o arquivo principal. Se desejar que seja `index.js`, pressione RETURN para aceitar o nome de arquivo padrão sugerido.

Agora instale o Express no diretório `app` e salve-o na lista de dependências. Por exemplo:

```
$ npm install express --save
```

Para instalar o Express temporariamente não o inclua na lista de dependências, omita a opção `--save`:

```
$ npm install express
```

Módulos do Node instalados com a opção `--save` são incluídas na lista `dependencies` no arquivo `package.json`. Posteriormente, executando `npm install` no diretório `app` irá automaticamente instalar os módulos na lista de dependências.

## Usando o Express

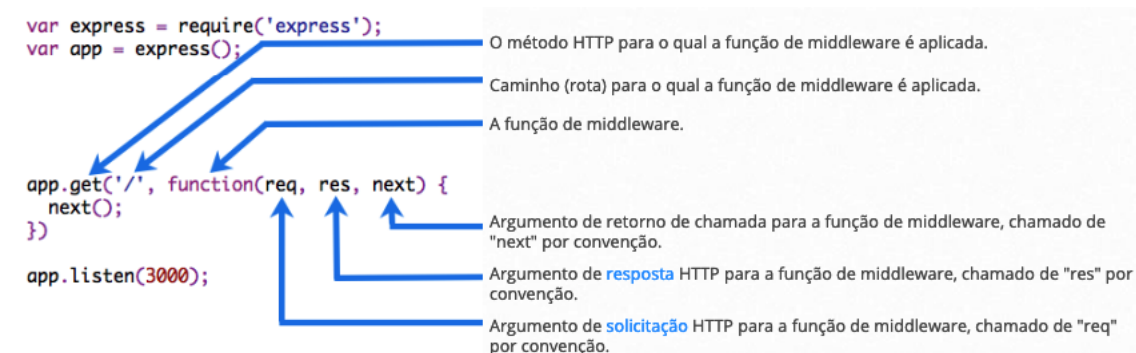
Funções de **Middleware** são funções que tem acesso ao [objeto de solicitação](#) (`req`), o [objeto de resposta](#) (`res`), e a próxima função de middleware no ciclo solicitação-resposta do aplicativo. A próxima função middleware é comumente denotada por uma variável chamada `next`.

Funções de middleware podem executar as seguintes tarefas:

- Executar qualquer código.
- Fazer mudanças nos objetos de solicitação e resposta.
- Encerrar o ciclo de solicitação-resposta.
- Chamar o próximo middleware na pilha.

Se a atual função de middleware não terminar o ciclo de solicitação-resposta, ela precisa chamar `next()` para passar o controle para a próxima função de middleware. Caso contrário, a solicitação ficará suspensa.

O exemplo a seguir mostra os elementos de uma chamada de função de middleware:



## Exemplo Hello World

Este é essencialmente o aplicativo mais simples do Express que é possível criar. Ele é um aplicativo de arquivo único — **não** é o que você iria obter usando o [Gerador Express](#), que cria a estrutura para um aplicativo completo com inúmeros arquivos JavaScript, modelos Jade, e subdiretórios para vários propósitos.

Primeiro crie um diretório chamado `myapp`, mude para ele e execute o `npm init`. Em seguida instale o `express` como uma dependência, de acordo com o [guia de instalação](#).

No diretório `myapp`, crie um arquivo chamado `app.js` e inclua o seguinte código:

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

O aplicativo inicia um servidor e escuta a porta 3000 por conexões. O aplicativo responde com “Hello World!” à solicitações para a URL raiz (/) ou **rota**. Para todos os outros caminhos, ele irá responder com um **404 Não Encontrado**.

O req (solicitação) e res (resposta) são os mesmos objetos que o Node fornece, para que seja possível chamar o req.pipe(), req.on('data', callback), e qualquer outra coisa que desejaria fazer sem o envolvimento do Express.

Execute o aplicativo com o seguinte comando:

```
$ node app.js
```

Em seguida, carregue <http://localhost:3000/> em um navegador para visualizar a saída

## Desenvolvimento

Aqui está um exemplo simples de uma função de middleware chamada “myLogger”. Esta função apenas imprime “LOGGED” quando uma solicitação para o aplicativo passa por ela. A função de middleware é designada para uma variável chamada myLogger.

```
var myLogger = function (req, res, next) {
  console.log('LOGGED');
  next();
};
```

Observe a chamada acima para next(). A chamada desta função chama a próxima função de middleware no aplicativo. A função next() não faz parte do Node.js ou da API Express, mas é o terceiro argumento que é passado para a função de middleware. A função next() poderia ter qualquer nome, mas por convenção ela é sempre chamada de “next”. Para evitar confusão, sempre use esta convenção.

Para carregar a função de middleware, chame app.use(), especificando a função de middleware. Por exemplo, o código a seguir carrega a função de middleware do myLogger antes da rota para o caminho raiz (/).

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



```

var app = express();

var myLogger = function (req, res, next) {
  console.log('LOGGED');
  next();
};

app.use(myLogger);

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000);

```

Sempre que o aplicativo recebe uma chamada, ele imprime a mensagem “LOGGED” no terminal.

A ordem de carregamento do middleware é importante: funções de middleware que são carregadas primeiro também são executadas primeiro.

Se `myLogger` é carregada após a rota para o caminho raiz, a chamada nunca chegará a ela e o aplicativo não imprimirá “LOGGED”, pois o manipulador de rota do caminho raiz encerra o ciclo de solicitação-resposta.

A função de middleware `myLogger` simplesmente imprime uma mensagem, e em seguida passa a solicitação para a próxima função de middleware na pilha chamando a função `next()`.

O próximo exemplo inclui uma propriedade chamada `requestTime` ao objeto da solicitação. Iremos chamar esta função de middleware de “requestTime”.

```

var requestTime = function (req, res, next) {
  req.requestTime = Date.now();
  next();
};

```

O aplicativo agora usa a função de middleware `requestTime`. Além disso, a função de retorno de chamada do caminho raiz usa a propriedade que a função de middleware inclui no `req` (o objeto da solicitação).

```

var express = require('express');
var app = express();

var requestTime = function (req, res, next) {
  req.requestTime = Date.now();
  next();
};

```

```
app.use(requestTime);

app.get('/', function (req, res) {
  var responseText = 'Hello World!';
  responseText += 'Requested at: ' + req.requestTime + ' ';
  res.send(responseText);
});

app.listen(3000);
```

Ao fazer uma solicitação para a raiz do aplicativo, o aplicativo agora exibe o registro de data e hora da sua solicitação no navegador.

Como você tem acesso ao objeto da solicitação, ao objeto de resposta, à próxima função de middleware na pilha, e à API completa do Node.js, as possibilidades com as funções de middleware são ilimitadas.

## Roteamento

O **Roteamento** refere-se à determinação de como um aplicativo responde a uma solicitação do cliente por um endpoint específico, que é uma URI (ou caminho) e um método de solicitação HTTP específico (GET, POST, e assim por diante).

Cada rota pode ter uma ou mais funções de manipulação, que são executadas quando a rota é correspondida.

A definição de rotas aceita a seguinte estrutura:

```
app.METHOD(PATH, HANDLER)
```

Onde:

- `app` é uma instância do `express`.
- `METHOD` é um [método de solicitação HTTP](#).
- `PATH` é um caminho no servidor.
- `HANDLER` é a função executada quando a rota é correspondida.

Este tutorial assume que uma instância de `express` chamada `app` está criada e o servidor está em execução. Caso não tenha familiaridade com a criação e inicialização de um aplicativo, consulte o [exemplo Hello world](#).

Os seguintes exemplos ilustram a definição de rotas simples.

Responder com `Hello World!` na página inicial:

```
app.get('/', function (req, res) {
  res.send('Hello World!');
});
```

```
});
```

Responder a uma solicitação POST na rota raiz (/) com a página inicial do aplicativo:

```
app.post('/', function (req, res) {  
  res.send('Got a POST request');  
});
```

Responder a uma solicitação PUT para a rota /user:

```
app.put('/user', function (req, res) {  
  res.send('Got a PUT request at /user');  
});
```

Responder a uma solicitação DELETE para a rota /user:

```
app.delete('/user', function (req, res) {  
  res.send('Got a DELETE request at /user');  
});
```

O código a seguir é um exemplo de uma rota muito básica.

```
var express = require('express');  
var app = express();  
  
// respond with "hello world" when a GET request is made to the homepage  
app.get('/', function(req, res) {  
  res.send('hello world');  
});
```

## Métodos de roteamento

Um método de roteamento é derivado a partir de um dos métodos HTTP, e é anexado a uma instância da classe `express`.

o código a seguir é um exemplo de rotas para a raiz do aplicativo que estão definidas para os métodos GET e POST.

```
// GET method route  
app.get('/', function (req, res) {
```

```
res.send('GET request to the homepage');
});
// POST method route
app.post('/', function (req, res) {
  res.send('POST request to the homepage');
});
```

O Express suporta os seguintes métodos de roteamento que correspondem aos métodos HTTP: get, post, put, head, delete, options, trace, copy, lock, mkcol, move, purge, propfind, proppatch, unlock, report, mkactivity, checkout, merge, m-search, notify, subscribe, unsubscribe, patch, search, e connect.

Para métodos de rota que são traduzidos para nomes de variáveis inválidas no Javascript, use a notação de colchetes. Por exemplo, `app['m-search']('/', function ...`

Existe um método de roteamento especial, `app.all()`, que não é derivado de nenhum método HTTP. Este método é usado para carregar funções de middleware em um caminho para todos os métodos de solicitação.

No exemplo a seguir, o manipulador irá ser executado para solicitações para “/secret” se você estiver usando GET, POST, PUT, DELETE, ou qualquer outro método de solicitação HTTP que é suportado no [módulo http](#).

```
app.all('/secret', function (req, res, next) {
  console.log('Accessing the secret section ...');
  next(); // pass control to the next handler
});
```

## Caminhos de rota

Caminhos de rota, em combinação com os métodos de solicitação, definem os terminais em que as solicitações podem ser feitas. Caminhos de rota podem ser sequências de caracteres, padrões de sequência, ou expressões regulares.

O Express usa o [path-to-regexp](#) para verificar a correspondência de caminhos de rota; consulte a documentação do path-to-regexp para obter todas as possibilidades nas definições de caminhos de rota.

O [Express Route Tester](#) é uma ferramenta útil para testar rotas básicas do Express, apesar de não suportar a correspondência de padrões.

Sequências de consulta não fazem parte dos caminhos de rota.

Aqui estão alguns exemplos de caminhos de rota baseados em sequências de caracteres

Este caminho de rota corresponde a solicitações à rota raiz, /.

```
app.get('/', function (req, res) {
  res.send('root');
```

```
});
```

Este caminho de rota irá corresponder a solicitações ao `/about`.

```
app.get('/about', function (req, res) {  
  res.send('about');  
});
```

Este caminho de rota irá corresponder a solicitações ao `/random.text`.

```
app.get('/random.text', function (req, res) {  
  res.send('random.text');  
});
```

Aqui estão alguns exemplos de caminhos de rota baseados em padrões de sequência

Este caminho de rota irá corresponder ao `acd` e `abcd`.

```
app.get('/ab?cd', function (req, res) {  
  res.send('ab?cd');  
});
```

Este caminho de rota irá corresponder ao `abcd`, `abxcd`, `abbbcd`, e assim por diante.

```
app.get('/ab+cd', function (req, res) {  
  res.send('ab+cd');  
});
```

Este caminho de rota irá corresponder ao `abcd`, `abxcd`, `abRABDOMcd`, `ab123cd`, e assim por diante.

```
app.get('/ab*cd', function (req, res) {  
  res.send('ab*cd');  
});
```

Este caminho de rota irá corresponder ao `/abe` e `/abcde`.

```
app.get('/ab(cd)?e', function (req, res) {  
  res.send('ab(cd)?e');  
});
```

Os caracteres `?`, `+`, `*`, e `()` são subconjuntos de suas contrapartes em expressões regulares. O hífen `-` e o ponto `.` são interpretados literalmente por caminhos baseados em sequências de caracteres.

Exemplos de caminhos de rota baseados em expressões regulares:

Este caminho de rota irá corresponder a qualquer coisa com um “a” no nome.

```
app.get(/a/, function(req, res) {  
  res.send('/a/');  
});
```

Este caminho de rota irá corresponder a `butterfly` e `dragonfly`, mas não a `butterflyman`, `dragonfly man`, e assim por diante.

```
app.get(/.*fly$/, function(req, res) {  
  res.send('/.*fly$/');  
});
```

## Manipuladores de rota

É possível fornecer várias funções de retorno de chamada que se comportam como [middleware](#) para manipular uma solicitação. A única exceção é que estes retornos de chamada podem chamar `next('route')` para efetuar um bypass nos retornos de chamada da rota restantes. É possível usar este mecanismo para impor pré-condições em uma rota, e em seguida passar o controle para rotas subsequentes se não houveram razões para continuar com a rota atual.

Manipuladores de rota podem estar na forma de uma função, uma matriz de funções, ou combinações de ambas, como mostrado nos seguintes exemplos.

Uma única função de retorno de chamada pode manipular uma rota. Por exemplo:

```
app.get('/example/a', function (req, res) {  
  res.send('Hello from A!');  
});
```

Mais de uma função de retorno de chamada pode manipular uma rota (certifique-se de especificar o objeto `next object`). Por exemplo:

```
app.get('/example/b', function (req, res, next) {
```

```

    console.log('the response will be sent by the next function ...');
    next();
  }, function (req, res) {
    res.send('Hello from B!');
  });

```

Uma matriz de funções de retorno de chamada podem manipular uma rota. Por exemplo:

```

var cb0 = function (req, res, next) {
  console.log('CB0');
  next();
}

var cb1 = function (req, res, next) {
  console.log('CB1');
  next();
}

var cb2 = function (req, res) {
  res.send('Hello from C!');
}

app.get('/example/c', [cb0, cb1, cb2]);

```

Uma combinação de funções independentes e matrizes de funções podem manipular uma rota. Por exemplo:

```

var cb0 = function (req, res, next) {
  console.log('CB0');
  next();
}

var cb1 = function (req, res, next) {
  console.log('CB1');
  next();
}

app.get('/example/d', [cb0, cb1], function (req, res, next) {
  console.log('the response will be sent by the next function ...');
  next();
}, function (req, res) {
  res.send('Hello from D!');
});

```

# Métodos de resposta

Os métodos do objeto de resposta (`res`) na seguinte tabela podem enviar uma resposta ao cliente, e finalizar o ciclo solicitação-resposta. Se nenhum destes métodos forem chamados a partir de um manipulador de rota, a solicitação do cliente será deixada em suspenso.

Método	Descrição
<a href="#">res.download()</a>	Solicita que seja efetuado o download de um arquivo
<a href="#">res.end()</a>	Termina o processo de resposta.
<a href="#">res.json()</a>	Envia uma resposta JSON.
<a href="#">res.jsonp()</a>	Envia uma resposta JSON com suporte ao JSONP.
<a href="#">res.redirect()</a>	Redireciona uma solicitação.
<a href="#">res.render()</a>	Renderiza um modelo de visualização.
<a href="#">res.send()</a>	Envia uma resposta de vários tipos.
<a href="#">res.sendFile</a>	Envia um arquivo como um fluxo de octeto.
<a href="#">res.sendStatus()</a>	Configura o código do status de resposta e envia a sua representação em sequência de caracteres como o corpo de resposta.

## app.route()

É possível criar manipuladores de rota encadeáveis para um caminho de rota usando o `app.route()`. Como o caminho é especificado em uma localização única, criar rotas modulares é útil, já que reduz redundâncias e erros tipográficos. Para obter mais informações sobre rotas, consulte: [documentação do Router\(\)](#).

Aqui está um exemplo de manipuladores de rotas encadeáveis que são definidos usando `app.route()`.

```
app.route('/book')
  .get(function(req, res) {
    res.send('Get a random book');
  })
  .post(function(req, res) {
    res.send('Add a book');
  })
  .put(function(req, res) {
```



```
res.send('Update the book');
});
```

## express.Router

Use a classe `express.Router` para criar manipuladores de rota modulares e montáveis. Uma instância de `Router` é um middleware e sistema de roteamento completo; por essa razão, ela é frequentemente referida como um “mini-aplicativo”

O seguinte exemplo cria um roteador como um módulo, carrega uma função de middleware nele, define algumas rotas, e monta o módulo router em um caminho no aplicativo principal.

Crie um arquivo de roteador com um arquivo chamado `birds.js` no diretório do aplicativo, com o seguinte conteúdo:

```
var express = require('express');
var router = express.Router();

// middleware that is specific to this router
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now());
  next();
});

// define the home page route
router.get('/', function(req, res) {
  res.send('Birds home page');
});

// define the about route
router.get('/about', function(req, res) {
  res.send('About birds');
});

module.exports = router;
```

Em seguida, carregue o módulo roteador no aplicativo:

```
var birds = require('./birds');
...
app.use('/birds', birds);
```

O aplicativo será agora capaz de manipular solicitações aos caminhos `/birds` e `/birds/about`, assim como chamar a função de middleware `timeLog` que é específica para a rota.

```
var express = require('express');
var router = express.Router();

// Home page route.
router.get('/', function (req, res) {
  res.send('Wiki home page');
})

// About page route.
router.get('/about', function (req, res) {
  res.send('About this wiki');
})

module.exports = router;
```

## Organizando Rotas

Usando o `Router()`, podemos dividir a aplicação em partes. Isso significa que podemos criar um `basicRouter` para o roteamento básico do nosso *frontend*. Podemos, também, ter um `adminRouter` para rotas de administração protegidas por uma autenticação. Roteando nossa aplicação dessa maneira podemos separar cada parte. Isso nos dá a flexibilidade que precisamos em aplicações e APIs mais complexas. Dessa maneira, mantemos a aplicação limpa e organizada, movendo cada definição de roteamento para seu respectivo arquivo e apenas o instanciando e passando para o `app.use()` dessa maneira:

```
app.use('/', basicRoutes)
app.use('/admin', adminRoutes)
app.use('/api', apiRoutes)
```

## Rotas com Parâmetros: `/users/:name`

Para adicionar uma rota com parâmetros na aplicação, precisamos de uma rota tipo `/admin/users/:name` onde passamos um nome na URL e a aplicação imprime um "*Faalaaa name!*". Veja como essa rota seria:

```
// rota com parâmetros (http://localhost:8000/admin/users/:name)
adminRouter.get('/users/:name', (req, res) => {
  res.send('Faalaaa ' + req.params.name + '!')
})
```

Agora quando visitarmos <http://localhost:8000/admin/users/dev> veremos um "*Faalaaa dev!*", o `req.params` armazena todos os parâmetros passados na requisição.

[Express Router Parameters]

Mais na frente, vamos usar isso para retornar os dados do usuário informado e construir um painel de controle para o gerenciamento de usuários.

Agora, vamos dizer que precisemos validar esse nome de alguma forma. Talvez para ter certeza de que não é um nome impróprio. Então, faríamos isso em um *middleware* e vamos usar um especial pra isso...

## Middleware para Parâmetros: .param()

Vamos usar o *middleware* `.param()` do Express. Isso cria um *middleware* que será executado para um parâmetro específico. Nesse caso, para o `:name` nessa rota. Novamente, precisamos ter certeza de colocá-lo antes da definição da rota e ficará dessa maneira:

```
// middleware de validação para 'name'
adminRouter.param('name', (req, res, next, name) => {
  // faça a validação do 'name' aqui
  // validação blah blah
  // logar alguma coisa pra sabermos se funciona
  console.log('validando o nome: ' + name)

  // quando a validação acabar, salve o novo nome na requisição
  req.name = name
  // vá para a próxima coisa a fazer
  next()
})

// roteamento com parâmetro (http://localhost:8000/admin/users/:name)
adminRouter.get('/users/:name', (req, res) => {
  res.send('Faalaaa ' + req.name + '!!')
})
```

Agora, quando a rota `/users/:name` for acessada, o *middleware* será executado. Dessa forma, podemos executar quaisquer validações e então passar a nova variável a rota `.get` armazenando-a no `req` (request). Então, podemos acessá-la trocando o `req.params.name` pelo `req.name` já que pegamos ela do `req.params.name` e colocamos no `req.name` dentro do *middleware*.

Quando visitarmos <http://localhost:8000/admin/users/lucas> no *browser* veremos a requisição logada no console.

[Express Router Parameter Middleware]

O *middleware* para parâmetros é usado para validar os dados enviados para a aplicação. Se precisarmos criar uma API RESTful, precisamos então validar um *token* e garantir que o usuário tem acesso a informação. Até agora, tudo que estamos fazendo em Node nos levará a construção da API RESTful que vai ser o *backend/server-side* que falamos no início do curso quando falamos sobre o modelo *client-server*.

A última *feature* do Express que vamos ver é como usar o `app.route()` para definir múltiplas rotas.

## Rotas de Autenticação: app.route()

Podemos definir as rotas diretamente no `app`. Isso é parecido com o uso do `app.get`, mas vamos usar o `app.route`. Ele é basicamente um atalho para o Express Router. Em vez de chamar `express.Router()`, usamos o `app.route` e começamos a definir as rotas. Usando o `app.route` podemos definir múltiplas ações para a mesma rota. E nesse caso, vamos precisar definir uma rota GET para mostrar o formulário de login e uma POST para processar a autenticação. Isso fica assim:

```
app.route('/login')
  // exibe o form (GET http://localhost:8000/login)
  .get((req, res) => {
    res.send('this is the login form')
  })
  // processa o form (POST http://localhost:8000/login)
  .post((req, res) => {
    console.log('processing')
    res.send('processing the login form!')
  })
```

Isso define duas ações diferentes para a rota `/login` de maneira simples e clara. Isso foi aplicado diretamente em nosso objeto `app` no `server.js`, mas poderia ser definido no objeto `adminRouter` que vimos antes.

## Entregando arquivos estáticos no Express

Para entregar arquivos estáticos como imagens, arquivos CSS, e arquivos JavaScript, use a função de middleware `express.static` integrada no Express.

Passe o nome do diretório que contém os ativos estáticos para a função de middleware `express.static` para iniciar a entregar os arquivos diretamente. Por exemplo, use o código a seguir para entregar imagens, arquivos CSS, e arquivos JavaScript em um diretório chamado `public`:

```
app.use(express.static('public'));
```

Agora, é possível carregar os arquivos que estão no diretório `public`:

```
http://localhost:3000/images/kitten.jpg
http://localhost:3000/css/style.css
http://localhost:3000/js/app.js
http://localhost:3000/images/bg.png
http://localhost:3000/hello.html
```

O Express consulta os arquivos em relação ao diretório estático, para que o nome do diretório estático não faça parte da URL.

Para usar vários diretórios de ativos estáticos, chame a função de middleware `express.static` várias vezes:

```
app.use(express.static('public'));
app.use(express.static('files'));
```

O Express consulta os arquivos na ordem em que você configurar os diretórios estáticos com a função de middleware `express.static`.

Para criar um prefixo de caminho virtual (onde o caminho não existe realmente no sistema de arquivos) para arquivos que são entregues pela função `express.static`, [especifique um caminho de montagem](#) para o diretório estático, como mostrado abaixo:

```
app.use('/static', express.static('public'));
```

Agora, é possível carregar os arquivos que estão no diretório `public` a partir do prefixo do caminho `/static`.

```
http://localhost:3000/static/images/kitten.jpg
http://localhost:3000/static/css/style.css
http://localhost:3000/static/js/app.js
http://localhost:3000/static/images/bg.png
http://localhost:3000/static/hello.html
```

Entretanto, o caminho fornecido para a função `express.static` é relativa ao diretório a partir do qual você inicia o seu `node` de processo. Se você executar o aplicativo `express` a partir de outro diretório, é mais seguro utilizar o caminho absoluto do diretório para o qual deseja entregar.

```
app.use('/static', express.static(__dirname + '/public'));
```

## Usando o EJS (Engine JavaScript)

O EJS é uma engine de visualização, com ele conseguimos de uma maneira fácil e simples transportar dados do back-end para o front-end, basicamente conseguimos utilizar códigos em javascript no html de nossas páginas.

Vamos criar uma pequena aplicação que utilizará o EJS para transportarmos dados para o nosso front-end e também vamos ver uma maneira de reaproveitar algumas partes do nosso html .

Para isso vamos precisar da seguinte estrutura de arquivos:

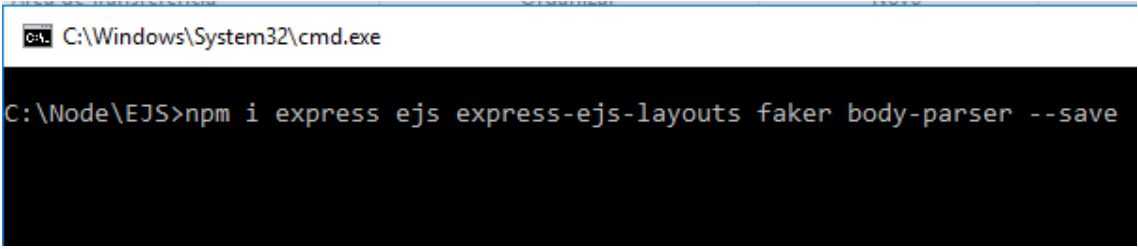
```
- public/
```

```
---- css/
----- styles.css
- views/
---- layout.ejs
---- pages/
----- about.ejs
----- contact.ejs
----- home.ejs
- package.json
- server.js
```

Instalando os pacotes do npm.

`npm i` ou `install express ejs --save`

Dessa forma o npm vai baixar os pacotes e já inserir eles nas dependências do nosso projeto, dentro do `package.json`



```
C:\Windows\System32\cmd.exe

C:\Node\EJS>npm i express ejs express-ejs-layouts faker body-parser --save
```

`express` É o pacote mais simples para criarmos as rotas do nosso app.

`ejs` É o pacote responsável pela engine EJS.

`express-ejs-layouts` Usamos ele para conseguirmos enviar dados para nossas páginas ejs pelo `express`.

`faker` Usamos ele para gerar algumas informações aleatórias como Nome, email, imagens. (Útil para testes)

`nodemon` Pacote usado para subir sua aplicação com a vantagem de que a cada vez que alterar ou criar um arquivo js ele reinicia automaticamente.

Após a instalação dos pacotes seu arquivo `package.json` ficará da seguinte forma:

```
...
{
  "dependencies": {
    {
      "body-parser": "^1.15.2",
      "ejs": "^2.5.2",
```

```
    "express": "^4.14.0",  
    "express-ejs-layouts": "^2.2.0",  
    "faker": "^3.1.0"  
  }  
}
```

## Variáveis que vamos utilizar.

No início do seu arquivo `server.js` adicione:

```
var express = require('express')  
var faker = require('faker')  
var bodyParser = require('body-parser')  
var expressLayouts = require('express-ejs-layouts')  
var app = express()  
var port = 3000
```

## Configurando o `server.js`

Logo após adicionarmos as variáveis, vamos configura-lás.

```
app.set('view engine', 'ejs')    // Setamos que nossa engine será o ejs  
app.use(expressLayouts)          // Definimos que vamos utilizar o express-  
    ejs-layouts na nossa aplicação  
app.use(bodyParser.urlencoded()) // Com essa configuração, vamos conseguir  
    parsear o corpo das requisições
```

## Criando as rotas.

Vamos utilizar as seguintes rotas

- GET /

```
app.get('/', (req, res) => {  
  res.render('pages/home')  
})
```

- Aqui o `server` recebeu uma requisição do `client` e devolveu o arquivo `home.ejs`, com isso o EJS renderiza ele para o `client` em forma de uma página html.
- GET /about

```
app.get('/about', (req, res) => {  
  var users = [{  
    name: faker.name.findName(),  
    email: faker.internet.email(),  
    avatar: 'http://placekitten.com/300/300'  
  }, {  
    name: faker.name.findName(),  
    email: faker.internet.email(),  
    avatar: 'http://placekitten.com/400/300'  
  }, {  
    name: faker.name.findName(),  
    email: faker.internet.email(),  
    avatar: 'http://placekitten.com/500/300'  
  }]  
  
  res.render('pages/about', { usuarios: users })  
})
```

O que o `server` faz aqui é praticamente a mesma coisa que foi feita na rota `home`, porém aqui nós devolvemos para o `client` além da página `about`, uma variável chamada `usuarios`.

- GET /contact

```
app.get('/contact', (req, res) => {  
  res.render('pages/contact')  
})
```

- POST /contact

```
app.post('/contact', (req, res) => {  
  res.send('Obrigado por entrar em contato conosco, ' + req.body.name + '!  
  Responderemos em breve!')  
})
```



O que o `server` faz aqui é recepcionar o conteúdo da requisição e devolver uma mensagem para o `client` aproveitando o nome da pessoa que enviou a mensagem para nós. Essa mensagem que vem no corpo da requisição, só é possível ser manipulada utilizando o pacote [body-parser](#).

No final o nosso arquivo `server.js` ficará assim:

```
var express = require('express')
var faker = require('faker')
var bodyParser = require('body-parser')
var expressLayouts = require('express-ejs-layouts')
var app = express()
var port = 3000

// Definimos que vamos utilizar o ejs
app.set('view engine', 'ejs')
app.use(expressLayouts)
app.use(bodyParser.urlencoded())

// ROTAS
app.get('/', (req, res) => {
  res.render('pages/home')
})

app.get('/about', (req, res) => {
  var users = [{
    name: faker.name.findName(),
    email: faker.internet.email(),
    avatar: 'http://placekitten.com/300/300'
  }, {
    name: faker.name.findName(),
    email: faker.internet.email(),
    avatar: 'http://placekitten.com/400/300'
  }, {
    name: faker.name.findName(),
    email: faker.internet.email(),
    avatar: 'http://placekitten.com/500/300'
  }]

  res.render('pages/about', { usuarios: users })
})
```

```

app.get('/contact', (req, res) => {
  res.render('pages/contact')
})

app.post('/contact', (req, res) => {
  res.send('Obrigado por entrar em contato conosco, ' + req.body.name + '! Responderemos em breve!')
})

app.use(express.static(__dirname + '/public'))
app.listen(port)
console.log('Servidor iniciado em http://localhost:' + port)

```

## Arquivos ejs

Agora vamos ver como vão ficar nossas páginas e também entender como trabalhar com elas.

- Layout

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Utilizando o EJS</title>
  <link rel="stylesheet" type="text/css"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
  <link rel="stylesheet" type="text/css" href="/css/styles.css">
</head>
<body>
  <header>
    <nav class="navbar navbar-inverse">
      <ul class="nav navbar-nav">
        <li><a href="/">Home</a></li>
        <li><a href="/about">About</a></li>
        <li><a href="/contact">Contact</a></li>
      </ul>
    </nav>
  </header>

  <main>

```

```
<div class="container">
  <%- body %>
</div>
</main>

<footer>
  EJS
</footer>
</body>
</html>
```

Aparentemente é uma página em html apenas, porém quando iniciamos nosso aplicativo e requisitamos alguma rota que devolve o arquivo `ejs`, o [express-ejs-layouts](#) procura o arquivo `layout.ejs` dentro da pasta `views`, então onde está o código `<%- body %>` será carregado o conteúdo de outro arquivo `ejs`. No próximo passo você vai entender como funciona essa ligação entre os arquivos.

- Home

```
<%- contentFor('body') %>
<div class="jumbotron text-center">
  Página Inicial
</div>
```

No arquivo `layout.ejs` vimos que o código `<%- body %>` seria substituído por outro assim que a página fosse renderizada, agora no arquivo `home.ejs` temos `<%- contentFor('body') %>`, é aqui que ocorre a ligação entre os dois arquivos, no lugar de `<%- body %>` ficará essa `div`.

- About

```
<%- contentFor('body') %>
<div class="jumbotron text-center">
  <h1>O Time!</h1>
  <div class="row">
    <% for (usuario of usuarios){%>
      <div class="col-sm-4">
        <h2><%= usuario.name %>
        
      </div>
    <% } %>
  </div>
</div>
```

```
</div>
</div>
```

O que fazemos aqui é renderizar para o `client` todos os nossos usuários, isso é possível pois fizemos um loop com o `ejs`, para capturarmos o valor de cada usuário que existe na variável `usuarios` que a rota `/about` devolveu.

- Contact

```
<div class="jumbotron text-center">
  <div class="row">
    <div class="col-sm-6 col-sm-offset-3">
      <h2>Entre em contato!</h2>
      <form action="/contact" method="POST">
        <div class="form-group">
          <label>Nome</label>
          <input type="text" name="name" class="form-
control">

        </div>
        <div class="form-group">
          <label>Email</label>
          <input type="text" name="email" class="form-
control">

        </div>
        <div class="form-group">
          <label>Sua mensagem</label>
          <input type="text" name="message" class="form-
control">

        </div>
        <div class="form-group text-right">
          <button type="submit" class="btn btn-primary
btn-lg">Enviar</button>
        </div>
      </form>
    </div>
  </div>
</div>
```

Nessa página usamos apenas o html para enviar um `POST` para a rota `/contact`.

## Nodemon

O pacote nodemon nos auxilia no momento do nosso desenvolvimento. Com ele, nós não precisamos dar stop e subir novamente a nossa APP. Ele verifica que

ocorreu uma alteração e já faz o refresh automaticamente. Para instalá-lo, execute o comando a baixo na sua console:

```
npm install -g nodemon
```

Você também pode instalar o nodemon como uma dependência de desenvolvimento:

```
npm install --save-dev nodemon
```

Com uma instalação local, o nodemon não estará disponível no caminho do seu sistema. Em vez disso, a instalação local do nodemon pode ser executada chamando-a a partir de um script npm (como npm start) ou usando npx nodemon.

O nodemon envolve seu aplicativo, para que você possa transmitir todos os argumentos que normalmente passaria para seu aplicativo:

```
nodemon [seu aplicativo]
```

Para opções de CLI, use o argumento -h(ou --help):

```
nodemon -h
```

Usar o nodemon é simples, se meu aplicativo aceitasse um host e uma porta como argumentos, eu começaria da seguinte forma:

```
nodemon ./server.js localhost 3000
```

Qualquer saída deste script é prefixada com [nodemon], caso contrário, todas as saídas de seu aplicativo, incluindo os erros, serão ecoadas como esperado.

Se nenhum script for fornecido, o nodemon testará o package.json e, se localizado, executará o arquivo associado à propriedade *main*.

```
{
  "name": "my_package",
  "description": "",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
}
```

Você também pode passar o `inspect` sinalizador para o nó através da linha de comando, como faria normalmente:

```
nodemon --inspect ./server.js 80
```

Se você tiver um `package.json` para seu aplicativo, poderá omitir totalmente o script principal e o nodemon lerá o `package.json` para a propriedade `init` e usará esse valor como o aplicativo.

## ROUTS URL

Dando continuidade ao ne especificamente voltando ao assunto sobre o framework ExpressJS, hoje apresentarei mais dicas sobre como desenvolver uma aplicação web com esse incrível módulo. No [post anterior](#) explicamos a essência, que na prática apresentei uma aplicação funcional, um mini-cadastro de clientes explorando as boas práticas de desenvolvimento **orientado à arquitetura REST**, utilizando os principais métodos do **protocolo HTTP (GET, POST, PUT e DELETE)** isso tudo no primeiro post hein!

Otimizando a nossa aplicação neste post incluiremos novos recursos: **inclusão de novas configurações no servidor da aplicação, separação de código HTML do código Server-Side, aplicação de filtros.**

Obs.: Para executar com sucesso o código abaixo, será necessário incluir um novo framework responsável por renderização de HTML dinâmico com Node.js, o [EJS](#).

```
var express = require('express')
var bodyParser = require('body-parser')
var app = express();

app.use(bodyParser.urlencoded({      // to support URL-encoded bodies
  extended: true
}));

var clientes = [];
app.get('/', function(req, res){
  html = '<html><body>';
  html += '<form action="/cliente" method="post">';
  html += '<label>Nome: <input type="text" name="cliente[nome]"></label><br>';
  html += '<label>Idade: <input type="text" name="cliente[idade]"></label><br>';
  html += '<button type="submit">Enviar</button>';
  html += '</form>';
  html += '<br>';
```

```

    html += '<h1>Lista de clientes</h1>';
    html += '<ul>';
    for(var i = 0; i < clientes.length; i++){
        html += '<li>'+clientes[i].nome+ ' | '+clientes[i].idade+'</li>';
    }
    html += '</ul></body></html>';
    res.send(html);
});

app.post('/cliente', function(req, res){
    var cliente = req.body.cliente;
    clientes.push(cliente);
    res.redirect('/');
});
app.listen(3000);

```

Complicando mais a nossa aplicação, que tal adicionarmos novas funcionalidades a esse cadastro, vamos implementar uma rota para excluir um registro, uma para possibilitar a edição de um registro existente e uma para submeter a atualização dos registros editados.

```

var express = require('express')
, app = express();
app.configure(function(){
    app.use(express.bodyParser());
    app.use(express.methodOverride());
});
var clientes = [];
app.get('/', function(req, res){
    var html = '<html><body>';
    html += '<form action="/cliente" method="post">';
    html += '<label>Nome: <input type="text" name="cliente[nome]"></label><br>';
    html += '<label>Idade: <input type="text" name="cliente[idade]"></label><br>';
    html += '<button type="submit">Enviar</button>';
    html += '</form>';
    html += '<br>';
    html += '<h1>Lista de clientes</h1>';
    html += '<ul>';
    for(var i = 0; i < clientes.length; i++){
        html += '<li>'+clientes[i].nome+ ' | '+clientes[i].idade;
        html += '<a href="/cliente/'+i+'/editar">Editar</a> | ';
        html += '<a href="/cliente/'+i+'">Excluir</a></li>';
    }
}

```

```

    html += '</ul></body></html>';
    res.send(html);
  });
  app.post('/cliente', function(req, res){
    var cliente = req.body.cliente;
    clientes.push(cliente);
    res.redirect('/');
  });
  app.get('/cliente/:id/editar', function(req, res){
    var id = req.params.id;
    var html = '<html><body>';
    html += '<h1>Editar dados do cliente: '+clientes[id].nome+'</h1>';
    html += '<form action="/cliente/'+ id +'\" method="post">';
    html += '<input type="hidden" name="_method" value="put">'; //Força o formulário realizar um PUT submit.
    html += '<label>Nome: <input type="text" name="cliente[nome]" value="'+clientes[id].nome+'\"></label>';
    html += '<label>Idade: <input type="text" name="cliente[idade]" value="'+clientes[id].idade+'\"></label>';
    html += '<button type="submit">Enviar</button>';
    html += '</form>';
    html += '</html>';
    res.send(html);
  });
  app.put('/cliente/:id', function(req, res){
    var id = req.params.id;
    clientes[id] = req.body.cliente;
    res.redirect('/');
  });
  app.del('/cliente/:id', function(req, res){
    var id = req.params.id;
    clientes.splice(id, 1);
    res.redirect('/');
  });
  app.listen(3000);

```

Pronto agora sim, criamos um cadastro completo de clientes! Alguns detalhes em destaque para explicar...

**req.params.id:** a função params de uma request, pega valores criados através de uma URL porém apenas valores que estejam citados explicitamente na string da rota, como por exemplo **'/cliente/:id'** que automaticamente criará uma variável com o nome id em que qualquer valor passado por este padrão será o valor dessa variável. Lembresse, **Params** é totalmente diferente de **QueryString**, em que para capturar valores via QueryStrings, basta utilizar a função **request.query.nome\_da\_variavel** veja esse exemplo:

```
// Ao executar a url dessa forma: http://localhost:3000/cliente/1?nome=caio
```



```
// Através dessa rota é possível pegar os seguintes valores dessa url.
app.get('/cliente/:id', function(req, res){
  var id = req.params.id; // valor: 1
  var nome = req.query.nome; // valor: caio
  res.send('Cliente '+id+' : '+nome);
});
```

`app.use(express.methodOverride())`: essa função que esta dentro do `app.configure`, permite que sua aplicação faça sobrescrita de rotas que utilizem métodos HTTP diferentes. Repare no código da nossa aplicação que a maioria das rotas possuem o mesmo nome, por exemplo a rota: `/cliente/:id`, porém são usadas em diferentes funções que vão entre o `get()`, `post()`, `put()` e `del()` que são as funções do **HTTP (GET, POST, PUT e DELETE)** essa é uma boa prática para quem pretende desenvolver aplicações que seguem o padrão REST que é muito utilizado na construção de **WebServices**.

## EJS – Embedded JavaScript (Modelos JavaScript incorporados)

### Instalação

```
npm install ejs
```

```
npm install ejs --save
```

### Criando a dependência do EJS no `package.json`

```
{
  "dependencies": {
    "body-parser": "^1.15.2",
    "ejs": "^2.5.2",
    "express": "^4.14.0",
    "express-ejs-layouts": "^2.2.0",
    "faker": "^3.1.0"
  }
}
```

### Configurando o App para usar o EJS:

```
app.set('view engine', 'ejs');
```

Definindo a pasta de Views do projeto:

```
app.set('views', './app/views');
```

Exemplo de uso do EJS:

```
var express = require("express");
var app = express();
app.set("view engine", "ejs");

app.get("/clientes", function(req, res){
    res.render("cliente ");
});

app.get("/produtos", function(req, res){
    res.render("produto ");
});

app.listen(3000;)
```

**/routes/clientes.js**

```
module.exports = function(app){
    app.get("/clientes", function(req, res){
        res.render("cliente/boas-vindas");
    });
};
```

**/routes/produto.js**

```
module.exports = function(app){
    app.get("/produtos", function(req, res){
        res.render("/produtos");
    });
};
```

## views/head.ejs

```
<meta charset="utf-8">

<link rel="stylesheet" href="/css/bootstrap.css">

<link rel="stylesheet" href="/css/bootstrap-theme.css">
```

## views/produtos/clientes.ejs

```
<html>
  <head>
    <title>Boas Vindas</title>
  </head>
  <body>
    <h1>Seja bem-vindo, cliente!</h1>
  </body>
</html>
```

## views/produtos/produtos.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>Listagem de Livros</title>
    <% include ../head %>
  </head>
  <body>
    <form action="/produtos" method="post">
      <div class="form-group">
        <label for="titulo">Titulo</label>
        <input type="text" name="titulo" placeholder="titulo livro"
              value="<%=livro.titulo%>"/>
      </div>
      <div class="form-group">
        <label for="preco">Preço</label>
        <input type="text" name="preco" placeholder="valor do livro"
              value="<%=livro.preco%>"/>
      </div>
      <div class="form-group">
        <label for="descricao">Descricao</label>
        <textarea name="descricao" cols="50" rows="5">
          <%=livro.descricao%>
        </textarea>
      </div>
    </form>
  </body>
</html>
```

```
        </textarea>

    </div>

    <button type="submit" class="btn btn-primary">Gravar</button>

</form>

</body>
</html>
```

## Criando uma visão dinâmica com ejs

Construir páginas estáticas é ótimo, mas às vezes precisamos alimentar conteúdo dinâmico em nossos modelos. Como o Render é construído sobre o Express, ele suporta todos os mecanismos de templates e vem com ejs . No exemplo abaixo abordaremos como passar dados de uma rota para um modelo ejs.

No arquivo app.js

```
var express = require('express');
var app = express();
app.set('view engine', 'ejs');

app.get('/', function(req, res) {
  var drinks = [
    { bebida: 'Bloody Mary', valor: 3 },
    { bebida: 'Martini', valor: 5 },
    { bebida: 'Scotch', valor: 10 }
  ];

  let blogPosts = [
    {
      title: 'Perk is for real!',
      body: '...',
      author: 'Aaron Lerner',
      publishedAt: new Date('2016-03-19'),
      createdAt: new Date('2016-03-19')
    }
  ];

  var titulo = 'Nome do Bar';

  res.render('index', {
    drinks: drinks,
    titulo: titulo,
    posts: blogPosts
  });
});

app.listen(3000);
```

No arquivo views/index.ejs

```

<h2><%= titulo %></h2>

<ul>
  <% drinks.forEach(function(drink) { %>
    <li><%= drink.bebida %> - <%= drink.valor %></li>
  <% }); %>
</ul>

<h1>Recent Blog Posts</h1>
<% for(let i = 0; i < posts.length; i++) { %>
  <article>
    <h2><%= posts[i].title %></h2>
    <p><%= posts[i].body %></p>
  </article>
<% } %>

```

## Retornando no projeto desenvolvido com o Express

Para otimizar nossa aplicação incluiremos novos recursos: **inclusão de novas configurações no servidor da aplicação, separação de código HTML do código Server-Side, aplicação de filtros.**

Mãos a obra! Confira abaixo a versão do código antigo já com algumas otimizações pra explicar:

```

var express = require('express')
  , app = express();
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  app.set('view options', {layout: false});
  app.use(express.bodyParser());
  app.use(express.methodOverride());
});
var clientes = [];
app.get('/', function(req, res){
  res.render('index', {clientes: clientes});
});
app.post('/cliente', function(req, res){
  var cliente = req.body.cliente;
  clientes.push(cliente);
  res.redirect('/');
});
app.get('/cliente/:id/editar', function(req, res){
  var id = req.params.id;
  res.render('edit', {cliente: clientes[id], id: id});
});
app.put('/cliente/:id', function(req, res){

```

```

    var id = req.params.id;
    clientes[id] = req.body.cliente;
    res.redirect('/');
  });
  app.del('/cliente/:id', function(req, res){
    var id = req.params.id;
    clientes.splice(id, 1);
    res.redirect('/');
  });
  app.listen(3000);

```

Reparem no quão **clean** ficou o código do server-side da aplicação, simplesmente separando código **html** do **javascript**, e melhorando o dinamismo de manipulações html, utilizamos o EJS como framework template engine, com isso incluiremos as páginas com extensões .ejs (index.ejs e edit.ejs) no **subdiretório chamado views**. Tudo isso com incluindo 3 novos parâmetros no **callback da função app.configure()**:

```

// Informa o subdiretório das views.
app.set('views', __dirname + '/views');
// Informa o template engine de renderização das views (Framework EJS)
app.set('view engine', 'ejs');
// Manipula layouts, o EJS não possui esse recurso então deixe false.
app.set('view options', {layout: false});

```

Abaixo mostrarei como ficarão as **views index.ejs e edit.ejs** que são referenciadas em algumas rotas da aplicação, através do método: `res.render('nome_da_view', {variaveis para renderização no cliente})`

```

<!-- index.ejs -->
<html>
  <body>
    <h1>Cadastro de cliente</h1>
    <form action="/cliente" method="post">
      <label>Nome:
        <input type="text" name="cliente[nome]">
      </label>
      <label>Idade:
        <input type="text" name="cliente[idade]">
      </label>
      <button type="submit">Enviar</button>
    </form>
    <h1>Lista de clientes</h1>
    <ul>
      <% for(var i = 0; i < clientes.length; i++){ %>

```

```
<li><%= clientes[i].nome %> - <%= clientes[i].idade %>
    <a href="/cliente/<%= i %>/editar">Editar</a>
    &nbsp;|&nbsp;
    <a href="/cliente/<%= i %>">Excluir</a>
</li>
<% } %>
</ul>
</body>
</html>
<!-- edit.ejs -->
<html>
  <body>
    <h1>Editar dados do cliente: <%= cliente.nome %></h1>
    <form action="/cliente/<%= id %>" method="post">
      <input type="hidden" name="_method" value="put">
      <label>Nome:
        <input type="text" name="cliente[nome]" value="<%= cliente.nome %>">
      </label>
      <label>Idade:
        <input type="text" name="cliente[idade]" value="<%= cliente.idade
%>">
      </label>
      <button type="submit">Enviar</button>
    </form>
  </body>
</html>
```

Pronto, agora temos em nossa aplicação o html separado dos arquivos js, e isso além de organizar seus códigos facilita na manutenção do mesmo. Agora pense comigo, o que acontecerá se eu cadastrar um cliente em branco? Seria legal adicionarmos um filtro para tratar os dados de forma que lance um erro caso tente cadastrar com dados em branco, e será isso que faremos! Antes das rotas da aplicação, adicionaremos a seguinte rotina para renderizar uma tela de erros:

```
app.use(function(req, res, next){
  res.render('404', {status: 404});
});

app.error(function(err, req, res, next){
  res.render('500', {status: 500, error: err});
});
```

Com isso teremos que criar duas páginas uma com o nome 404.ejs e a outra 500.ejs, em que elas irão renderizar mensagens de erros do sistema.

```
<!-- 404.ejs -->
```

```

<html>
  <head>
    <title>Página não encontrada.</title>
  </head>
  <body>
    <h1>Página não encontrada.</h1>
  </body>
</html>
<!-- 500.ejs -->
<html>
  <head>
    <title>Erro na aplicação.</title>
  </head>
  <body>
    <h1>Erro na aplicação.</h1>
    <h3>Detalhes: <%= error.message %></h3>
  </body>
</html>

```

Agora com toda estrutura para tratamento de erros pronto, precisamos criar uma regra de tratamento sobre os campos do cliente, que na prática serão incluídos funções de filtros para serem realizadas antes da execução de uma determinada rota. Para se fazer isso no Express, qualquer rota que incluir a declaração de uma função como callback antes da função principal da rota será considerado como filtro que executa antes da rota principal, veja o exemplo abaixo:

```

// Funções que recebem três parâmetros são consideradas filtros.
var validarCampos = function(req, res, next){
  if(req.body.cliente.nome){
    return next(new Error('Informe o nome do cliente.'));
  }
  if(req.body.cliente.idade){
    return next(new Error('Informe a idade do cliente.'));
  }
  return next();
}
// Inclua a função validarCampos no callback de execução de uma rota.
app.post('/cliente', validarCampos, function(req, res){
  var cliente = req.body.cliente;
  clientes.push(cliente);
  res.redirect('/');
});
app.put('/cliente/:id', validarCampos, function(req, res){
  var id = req.params.id;
  clientes[id] = req.body.cliente;

```



```
res.redirect('/');  
});
```

Esse foi um exemplo simples de tratamento de erros, na prática recomendo que utilize-o para tratamentos complexos, visto que esse exemplo de validação de campos é apenas um exemplo hipotético de como explorar os redirecionamentos de páginas de erros. Para facilitar veja abaixo como ficou o nosso código back-end da aplicação:

```
var express = require('express')  
  , app = express();  
app.configure(function(){  
  app.set('views', __dirname + '/views');  
  app.set('view engine', 'ejs');  
  app.set('view options', {layout: false});  
  app.use(express.bodyParser());  
  app.use(express.methodOverride());  
});  
var clientes = [];  
var validarCampos = function(req, res, next){  
  if(req.body.cliente.nome){  
    return next(new Error('Informe o nome do cliente.'));  
  }  
  if(req.body.cliente.idade){  
    return next(new Error('Informe a idade do cliente.'));  
  }  
  return next();  
}  
app.use(function(req, res, next){  
  res.render('404', {status: 404});  
});  
app.error(function(err, req, res, next){  
  res.render('500', {status: 500, error: err});  
});  
app.get('/', function(req, res){  
  res.render('index', {clientes: clientes});  
});  
app.post('/cliente', validarCampos, function(req, res){  
  var cliente = req.body.cliente;  
  clientes.push(cliente);  
  res.redirect('/');  
});  
app.get('/cliente/:id/editar', function(req, res){  
  var id = req.params.id;
```

```

    res.render('edit', {cliente: clientes[id], id: id});
  });
  app.put('/cliente/:id', validarCampos, function(req, res){
    var id = req.params.id;
    clientes[id] = req.body.cliente;
    res.redirect('/');
  });
  app.del('/cliente/:id', function(req, res){
    var id = req.params.id;
    delete clientes[id];
    res.redirect('/');
  });
  app.listen(3000);

```

Para entender melhor vamos analisar o arquivo app.js, perceba que temos nesse arquivo as instruções:

```

var routes = require('./routes/index');
app.get('/', routes);

```

Primeiro armazenamos na variável routes o código que contém em /routes/index.js.

A segunda instrução nos diz que, quando for feito um GET na url '/' executaremos o que está presente em routes.

Nesse arquivo, /routes/index.js temos o código:

```

res.render('index', { title: 'Express' });

```

Que vai renderizar o arquivo index.ejs da pasta views, passando title com a string Express.

No arquivo views/index.ejs temos o código:

```

<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>

```

Que vai resultar em:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Express</title>
  </head>
  <body>
    <h1>Express</h1>
    <p>Welcome to Express</p>
  </body>
</html>
```

O ejs é uma engine com sintaxe semelhante ao PHP. Nos permite escrever códigos como:

```
<body>
  <% for(var i=0; i<10; i++) { %>
    <p>Welcome to <%= title %> <%= i %></p>
  <% } %>
</body>
```

Resultando em:

```
<body>
  <p>Welcome to Express 0</p>
  <p>Welcome to Express 1</p>
  <p>Welcome to Express 2</p>
  <p>Welcome to Express 3</p>
  <p>Welcome to Express 4</p>
  <p>Welcome to Express 5</p>
  <p>Welcome to Express 6</p>
  <p>Welcome to Express 7</p>
  <p>Welcome to Express 8</p>
  <p>Welcome to Express 9</p>
</body>
```

É basicamente assim que o Express funciona. Agora você já pode criar suas próprias rotas, por exemplo, uma REST API para eventos.

```
var express = require('express');
var app = express();
var router = express.Router();

router.route('/api/events')
```

```

    .get(function(req, res, next) {
        // Código para listar eventos
    })

    .post(function(req, res, next) {
        // Código para criar eventos
    });

router.route('/api/events/:event_id')
    .get(function(req, res, next) {
        // Código para mostrar detalhe do evento
        // baseado no parâmetro id
    })

    .put(function(req, res, next) {
        // Código para atualizar evento
        // baseado no parâmetro id
    })

    .delete(function(req, res, next) {
        // Código para mostrar deletar evento
        // baseado no parâmetro id
    });

```

## ExpressJS e Middlewares

Seguindo na série de NodeJS. Já vimos como instalar o Node, criamos uma aplicação com o Express e já entendemos como funcionam as rotas e o view engine. Vamos agora entender um dos conceitos mais poderosos no Express: Middlewares.

Middleware, em poucas palavras e de maneira bem simples, é um código que roda no pipeline de requisição http. O request chega, passa pelo middleware, e em algum dos componentes desse middleware ele é tratado e a resposta é enviada. Normalmente configuramos os middlewares no arquivo app.js, que configura toda a aplicação. Toda vez que encontrar uma chamada do método "use" da aplicação do express (normalmente "app.use") você verá um middleware. Vejamos os middlewares presentes na aplicação básica do Express, assim que é criada. Aqui está o seu app.js:

```

var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

```

```

var app = express();

app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));

if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});

```

Vemos o uso de 6 middlewares, juntos, e mais um somente em ambiente de desenvolvimento (`express.errorHandler`). De onde eles vêm? Vocês podem ver que todos aparecem como extensões do objeto “`express`”, de forma indireta ou direta. No entanto, o projeto do Express não é o responsável por esses middlewares, eles ficam em outro projeto (com uma exceção), em outro componente npm chamado “`Connect`”. O `Connect` é a base dos middlewares do Express.

Vá à página do `Connect` e você verá diversos outros middlewares não listados no `app.js` que o Express cria. Dos que temos disponíveis nesse arquivo, via objeto “`express`”, todos (menos um) vem do `Connect`. O que o Express faz na prática é expor os middlewares do `Connect` como se fossem dele, para que você não tenha que referenciar ou instalar o `Connect` diretamente, nem requerer ele no `app.js`. Você pode utilizar o `Connect` diretamente, sem o Express, e aí vai ter que instalar ele via NPM, e vai usar o objeto do `Connect` para referenciar os middlewares. Com Express isso não é necessário.

Vejamos então o que faz cada um dos middlewares que estamos utilizando:

**Favicon.** Entrega um favicon, que por padrão é o do Express, mas você pode trocar informado o caminho de um arquivo.

Logger. Mostra na console onde o node está rodando as chamadas recebidas pelo server. Costumo deixar desligado em produção. Muito útil durante desenvolvimento. Fica assim:

```
~/proj/src/rabiscos/web$ node app
Express server listening on port 3000
GET / 200 419ms - 170b
GET / 200 14ms - 170b
GET /stylesheets/style.css 200 11ms - 110b
GET /stylesheets/style.css 304 4ms
GET / 200 20ms - 170b
GET /stylesheets/style.css 200 3ms - 110b
```

BodyParser. Fundamental, faz o parse durante um post, independente se é um form sem file upload, com file upload, ou um post ajax. Na verdade é a reunião de 3 outros middlewares, json, urlencoded e multipart.

MethodOverride. Permite usar métodos http em browsers ou situações em que esses métodos não seriam suportados. Por exemplo, browsers em geral não enviam de formulários o método PUT, mas se você colocar um elemento hidden com nome "\_method" e valor "PUT", o express vai entender que é um PUT, por causa desse middleware.

app.router. Esse é um middleware do Express diretamente, não do Connect. Ele coloca as suas rotas (app.get, app.post, etc), no pipeline dos middlewares. Sua chamada é opcional, e já vou explicar porque.

Static. Serve arquivos estáticos, podendo mapear um diretório físico a um diretório virtual. O padrão é entregar qualquer arquivo que esteja no diretório "public". É por esse motivo que você consegue carregar o arquivo de folha de estilos que está em no diretório "public/stylesheets/style.css" chamando <http://localhost:3000/stylesheets/style.css>.

ErrorHandler. Quando acontece uma exception ele mostra ela. Muito útil durante desenvolvimento e perigoso em produção (por expor partes internas do server), você o vê quando tem um problema na aplicação e ele mostra o stack trace todo no navegador.

É importante entender que os middlewares são executados em ordem. Cada request que chega passa por cada um dos middlewares, até que um deles lide com o request, ou passe o request para o middleware seguinte. Assim, se você colocar 50 middlewares até chegar no middleware que recebe os posts da sua aplicação você terá um overhead considerável.

Com Express você é quem decide o peso da sua aplicação. Quanto mais middlewares, mais pesada. Isso permite ter uma app extremamente leve que faz somente o que precisa. E porque a ordem importa? É fácil explicar. Imagine que você tem o arquivo public/carrinho/index.html, mas também tem uma rota que aponta para um comportamento dinâmico em "/carrinho", via "app.get('/carrinho', algumafuncao)". Quem vai responder para "/carrinho", a sua rota, ou o arquivo estático? Depende da ordem dos middlewares. Se você colocou o middleware Static antes do app.router, o arquivo estático será servido, se colocou depois será a sua rota. É simples assim. Lembre-se que eu disse que o cadastro do middleware de rotas com "app.use(app.router)" era opcional? Pois é, isso porque o express vai utilizar suas rotas mesmo se você não o chamar, mas vai colocá-las no final do pipeline de middlewares. Se você quiser ela em outro lugar, você tem que configurá-la com app.use. Por convenção recomendo sempre colocar o app.use de rotas pra ficar explícito. E evite colocar arquivos diretamente no diretório "public" que poderiam conflitar com suas rotas, no caso, seria melhor colocar o arquivo em "public/html/carrinho/index.html".

Diversos outros middlewares estão disponíveis. Há middlewares para sessão, gestão de cookies, cache, autenticação, todos já disponíveis no Connect e portanto no Express, além de inúmeros outros feitos pela comunidade. As possibilidades são infinitas. O interessante é que com uma linha de configuração você pode incluir um novo middleware e alterar completamente o comportamento da sua aplicação. Por exemplo, pra integrar autenticação

via usuário/senha, Facebook, Twitter, Github, etc, eu utilizo o middleware Everyauth, que já faz tudo pra mim e tenho apenas que fazer uma configuração mínima (exemplo aqui). É simples assim.

E o que é um middleware? Você deve estar imaginando que é algo super complexo, certo? Mas não é. Uma linguagem funcional como JavaScript permite tornar algumas coisas bem simples e esse é um caso: um middleware é uma função que recebe um request, um response, e uma outra função que chamará o próximo middleware. Veja, por exemplo, o código do middleware favicon aqui. O método `express.favicon()` vai retornar uma função que recebe (req, res, next) e serve o arquivo. Simples assim. Assim, se você notar que tem um comportamento que precisa que aconteça no meio do pipeline, você pode simplesmente escrever uma função com essa assinatura e chamar `app.use` nela no arquivo `app.js` e pronto: já terá o seu middleware funcionando.

A ideia de middlewares não é nova, apesar de ser implementada de formas diferentes. O Rails usa, o OWIN usa, e diversos outros frameworks web usam. É importante entender o que eles fazem e como funcionam para tirar o melhor proveito do ambiente que está disponível. Dê uma boa lida na

## Sua primeira aplicação com o Express - Hello Word

Criando o primeiro arquivo JS com Express (`program.js`)

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

app.get('/', function (req, res) {
  res.send('Hello World!')
})

var server = app.listen(3000, function () {
  var host = server.address().address
  var port = server.address().port

  console.log('Executando seu app em http://%s:%s', host, port)
})
```

Salve e volte para o terminal, execute o arquivo.

```
node program.js
```

Você poderá ver o resultado abrindo a url <http://localhost:3000> em seu navegador.

## Explicando o código

Incluimos o módulo **Express.js** através de `require()` e atribuímos a variável `express`.

```
var express = require('express')
```

Inicializamos o módulo **Express.js** chamando a função `express()`.

```
var app = express()
```

Vamos executar a função `get()`, ela aceita 2 parâmetros: o primeiro é o caminho, url. O segundo é a função de **callback** que será executada após identificado o caminho. Através do método `res.send()` elaboramos uma resposta para o navegador.

```
app.get('/', function (req, res) {  
  res.send('Hello World!')  
})
```

Agora dizemos ao **Express.js** para escutar na porta 3000

```
var server = app.listen(3000)
```

Isso já seria o suficiente para funcionar, bastaria agora executar no terminal `node program.js` e abrir a url <http://0.0.0.0:3000> no navegador.

A função `app.listen()` aceita como segundo parâmetro uma função de **callback** e é isso o que vamos fazer.

```
var server = app.listen(3000, function () {  
  ...  
  ...  
  ...  
})
```

Neste callback, vamos recuperar o endereço de request (url) e a porta utiliza.

```
var server = app.listen(3000, function () {  
  
  var host = server.address().address  
  var port = server.address().port  
  
})
```

Com esses dados, em mãos podemos imprimi-los no console.

```
var server = app.listen(3000, function () {  
  
  var host = server.address().address
```



```
var port = server.address().port
```

```
console.log('Executando seu app em http://%s:%s', host, port)
```

```
})
```

Para testar abra o navegador com o endereço <http://localhost:3000>

```
http://localhost:3000
```

## Montando rotas no framework Express.js

Este é o segundo artigo sobre o framework [Express.js](#). Estou assumindo que você já leu, praticou e entendeu o artigo de [introdução ao Express](#).

Abaixo vemos um exemplo para algumas rotas.

```
var express = require('express')
```

```
var app = express()
```

```
// Rota para a home, seu domínio.
```

```
app.get('/', function (req, res) {
```

```
  res.send('Sua home');
```

```
})
```

```
// Rota para /produtos
```

```
app.get('/produtos', function (req, res) {
```

```
  res.send('exibindo produtos!');
```

```
})
```

```
// Rota para /usuarios
```

```
app.get('/usuarios', function (req, res) {
```

```
  res.send('exibindo usuários');
```

```
})
```

```
app.listen(3000);
```

Crie um arquivo qualquer, por exemplo, `exemplo.js`, insira o conteúdo acima e execute no terminal:

```
node exemplo.js
```

Abra seu navegador e experimente as seguintes url's:

- <http://localhost:3000/>
- <http://localhost:3000/produtos/>
- <http://localhost:3000/usuarios/>

O **Express.js** vem equipado com funções de roteamento para beneficiar-se do conceito **REST**.

As seguintes rotas apontam todas para a url `/user` porém, são distintas pelo método (verbo) utilizado.

```
app.post('/user', function (req, res) {  
  res.send('Got a POST request at /user');  
})  
  
app.put('/user', function (req, res) {  
  res.send('Got a PUT request at /user');  
})  
  
app.delete('/user', function (req, res) {  
  res.send('Got a DELETE request at /user');  
})
```

Para você compreender de fato os exemplos acima, não tem jeito, deve saber o básico sobre **REST**.

Nós podemos acessar, por exemplo, usuários (users) sobre o protocolo **HTTP** usando o CRUD (Create, Read, Update e Delete) mapeada para uma API **REST**.

- **GET /users** – Método index que retorna a lista de usuários
- **POST /users/** – Cria um novo usuário
- **GET /users/:id** – Retorna um único usuário através de seu id
- **PUT /users/:id** – Atualiza um único usuário através de seu id
- **DELETE /users/:id** – Deleta um único usuário através de seu id

## API

É cada dia mais comum termos aplicações que funcionem única e exclusivamente pela Internet, sendo consumidas por navegadores em desktops, notebooks ou dispositivos móveis, isto é, independente de plataforma. Por outro lado, grandes empresas precisam alimentar seus softwares de gestão (estoque, contabilidade, ERP e redes sociais) com dados, a todo momento.

Com essas duas problemáticas em mente (softwares sendo acessados pela Web e empresas precisando alimentar seus sistemas) começou-se a pensar em uma solução de software que permitisse a conversa entre sistemas e usuários.

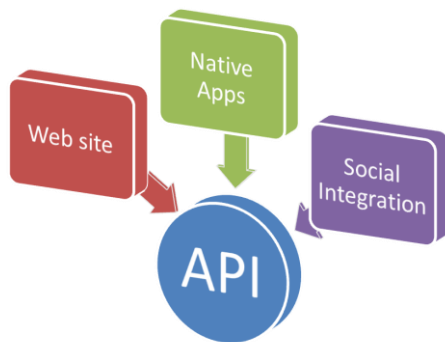
Durante anos, diversas alternativas surgiram e, de uma forma geral, essas aplicações ficaram conhecidas como **APIs**. Basicamente, o funcionamento dessas aplicações baseava-se em fornecer um ponto de acesso entre a aplicação e seu cliente, seja ele um usuário ou uma outra aplicação.

O que é API?

O acrônimo **API** que provém do inglês **Application Programming Interface** (Em português, significa Interface de Programação de Aplicações), trata-se de um conjunto de rotinas e padrões estabelecidos e documentados por uma aplicação A, para que outras aplicações consigam utilizar as funcionalidades desta aplicação A, sem precisar conhecer detalhes da implementação do software.

Desta forma, entendemos que as APIs permitem uma **interoperabilidade entre aplicações**. Em outras palavras, a comunicação entre aplicações e entre os usuários.

Exemplo de API: Twitter Developers



Agora que já sabemos que uma API permite a interoperabilidade entre usuários e aplicações, isso reforça ainda mais a importância de pensarmos em algo padronizado e, de preferência, de fácil representação e compreensão por humanos e máquinas. Isso pode soar um pouco estranho, mas veja esses três exemplos:

Representação XML

```
<endereco>
  <rua>
    Rua Recife
  </rua>
  <cidade>
    Paulo Afonso
  </cidade>
</endereco>
```

Representação JSON

```
{endereco:
```

```
{  
  rua: Rua Recife,  
  cidade: Paulo Afonso  
}  
}
```

#### Representação YAML

```
endereco:  
  rua: rua Recife  
  cidade: Paulo Afonso
```

Qual deles você escolheria para informar o endereço em uma carta? Provavelmente o último, por ser de fácil entendimento para humanos, não é mesmo? Contudo, as 3 representações são válidas, pois nosso entendimento final é o mesmo, ou seja, a semântica é a mesma.

Por outro lado, você deve concordar comigo que a primeira representação (**XML**) é mais verbosa, exigindo um esforço extra por parte de quem está escrevendo. No segundo exemplo (**JSON**) já é algo mais leve de se escrever. Já o último (**YAML**), é praticamente como escrevemos no dia a dia.

Sendo assim, esse é o primeiro passo que precisamos dar para permitir a comunicação interoperável. E o mais legal é que essas 3 representações são válidas atualmente, ou seja, homens e máquinas podem ler, escrever e entender esses formatos.

## Origem do REST

O HTTP é o principal protocolo de comunicação para sistemas Web, existente há mais de 20 anos, e em todo esse tempo sofreu algumas atualizações. Nos anos 2000, um dos principais autores do protocolo HTTP, Roy Fielding, sugeriu, dentre outras coisas, o uso de novos métodos HTTP. Estes métodos visavam resolver problemas relacionados a semântica quando requisições HTTP eram feitas.

Estas sugestões permitiram o uso do HTTP de uma forma muito mais próxima da nossa realidade, dando sentido às requisições HTTP. Para melhor compreensão, veja os exemplos abaixo (requisições em formatos fictícios):

GET <http://www.meusite.com/usuarios>

DELETE <http://www.meusite.com/usuarios/jackson>

POST <http://www.meusite.com/usuarios> -data {nome: joaquim}

Pela simples leitura (mesmo o método GET e DELETE sendo em inglês) é possível inferir que no primeiro caso estamos pegando (GET) todos os usuários do site, ou seja, teremos uma lista de todos os usuários que estão cadastrados no sistema/site. Já, no segundo caso, estamos apagando (DELETE) o usuário Jackson. No último exemplo, estamos usando o método POST, em que percebemos o envio de dados extras para cadastrar um novo usuário.

Veja o quão simples ficou expressar o que desejamos realizar ao acessar um determinado endereço, usando verbos específicos para URLs específicas e usando dados padronizados, quando necessário.

Estes princípios apresentados fazem parte do REST! Em outras palavras, nesses exemplos, temos: uma representação padronizada, verbos e métodos usados, bem como, URLs.

## O que é REST?

**REST** significa *Representational State Transfer*. Em português, **Transferência de Estado Representacional**. Trata-se de uma abstração da arquitetura da Web. Resumidamente, o REST consiste em princípios/regras/constraints que, quando seguidas, permitem a criação de um projeto com interfaces bem definidas. Desta forma, permitindo, por exemplo, que aplicações se comuniquem.

Existe uma certa confusão quanto aos termos **REST** e **RESTful**. Entretanto, ambos representam os mesmos princípios. A diferença é apenas gramatical. Em outras palavras, sistemas que utilizam os princípios REST são chamados de RESTful.

**REST:** conjunto de princípios de arquitetura

**RESTful:** capacidade de determinado sistema aplicar os princípios de REST.

O REST é um estilo de arquitetura que define um conjunto de restrições e propriedades baseados em HTTP. Web Services que obedecem ao estilo arquitetural REST, ou web services RESTful, fornecem interoperabilidade entre sistemas de computadores na Internet.

Webservices compatíveis com REST permitem que os sistemas solicitantes acessem e manipulem representações textuais de recursos da Web usando um conjunto uniforme e predefinido de operações sem estado. Outros tipos de web services, como web services SOAP, expõem seus próprios conjuntos arbitrários de operações.

"Recursos web" foram primeiramente definidos na World Wide Web como documentos ou arquivos identificados por seus URLs. Entretanto, hoje eles possuem uma definição muito mais genérica e abstrata que abrangem todas as coisas ou entidades que podem ser identificadas, nomeadas, endereçadas ou manipuladas qualquer que seja a maneira, na web. Em um web service RESTful, requisições feitas a um URI de recurso extrairá uma resposta que pode estar em XML, HTML, JSON ou algum outro formato. A resposta pode confirmar que alguma alteração foi realizada para o recurso armazenado e a resposta pode fornecer ligações de hipertexto para outros recursos ou coleções de recursos relacionados. Quando o HTTP é usado, como é mais comum, as operações disponíveis são GET, POST, PUT, DELETE e outros métodos HTTP CRUD pré-definidos.

Por meio da utilização de um protocolo sem estado e operações padrões, sistemas REST destinam-se para desempenho rápido, confiabilidade e habilidade de crescimento, por meio da reutilização de componentes que podem ser gerenciados e atualizados sem afetar o sistema como um todo, mesmo que esteja em execução.

O REST ignora os detalhes da implementação de componente e a sintaxe de protocolo com o objetivo de focar nos papéis dos componentes, nas restrições sobre sua interação com outros componentes e na sua interpretação de elementos de dados significantes.

O termo transferência de estado representacional foi apresentado e definido no ano de 2000 por Roy Fielding, um dos principais autores da especificação do protocolo HTTP que é utilizado por sites da Internet, em uma tese de doutorado (PHD) na UC Irvine. Sua dissertação explicou os princípios da REST que ficou conhecida como "modelo de objeto HTTP" no início de 1994, e foi usada nos padrões HTTP 1.1 e Identificadores de Recursos Uniformes.

O termo destina-se a evocar uma imagem de como uma aplicação web bem projetada se comporta: ela é uma rede de recursos web (uma máquina de estado virtual) onde o usuário progride ao longo da aplicação por meio de seleção de ligações, como /user/tom, e operações como GET ou DELETE (transições de estado), resultando no próximo recurso (representando o próximo estado da aplicação) sendo transferida ao usuário para sua utilização.

REST se referia, originalmente, a um conjunto de princípios de arquitetura (descritos mais abaixo), na atualidade se usa no sentido mais amplo para descrever qualquer interface web simples que utiliza XML (ou YAML, JSON, ou texto puro) e HTTP, sem as abstrações adicionais dos protocolos baseados em padrões de trocas de mensagem como o protocolo de serviços Web SOAP. É possível projetar sistemas de serviços Web de acordo com o estilo arquitetural REST descrito por Fielding, e também é possível projetar interfaces XMLHTTP de acordo com o estilo de RPC mas sem utilizar SOAP. Estes usos diferentes do termo REST causam certa confusão em discussões técnicas, onde RPC não é um exemplo de REST.

## **Princípios**

Um protocolo cliente/servidor sem estado: cada mensagem HTTP contém toda a informação necessária para compreender o pedido. Como resultado, nem o cliente e nem o servidor necessitam gravar nenhum estado das comunicações entre mensagens. Na prática, muitas aplicações baseadas em HTTP utilizam cookies e outros mecanismos para manter o estado da sessão (algumas destas práticas, como a reescrita de URLs, não são permitidas pela regra do REST).

Um conjunto de operações bem definidas que se aplicam a todos os recursos de informação: HTTP em si define um pequeno conjunto de operações, as mais importantes são POST, GET, PUT e DELETE. Com frequência estas operações são combinadas com operações CRUD para a persistência de dados, onde POST não se encaixa exatamente neste esquema.

Uma sintaxe universal para identificar os recursos. No sistema REST, cada recurso é unicamente direcionado através da sua URI.

O uso de hipermídia, tanto para a informação da aplicação como para as transições de estado da aplicação: a representação deste estado em um sistema REST são tipicamente HTML ou XML. Como resultado disto, é possível navegar com um recurso REST a muitos outros, simplesmente seguindo ligações sem requerer o uso de registros ou outra infraestrutura adicional.

## **Recursos**

Um conceito importante em REST é a existência de recursos (elementos de informação), que podem ser usados utilizando um identificador global (um Identificador Uniforme de Recurso) para manipular estes recursos, os componentes da rede (clientes e servidores) se comunicam através de uma interface padrão (HTTP) e trocam representações de recursos (os arquivos ou ficheiros são recebidos e enviados) – é uma questão polêmica e gera grande discussão, sem a distinção entre recursos e suas representações é demasiado utópico o seu uso prático na rede, onde é popular na comunidade RDF.

O pedido pode ser transmitido por qualquer número de conectores (por exemplo clientes, servidores, caches, etc) mas não poderá ver mais nada do seu próprio pedido (conhecido com separação de camadas, outra restrição do REST, que é um princípio comum com muitas outras partes da arquitetura de redes e da informação). Assim, uma aplicação pode interagir com um recurso conhecendo o identificador do recurso e a ação requerida, não necessitando conhecer se existem

caches, proxys, ou outra, entre ela e o servidor que guarda a informação. A aplicação deve compreender o formato da informação de volta (a representação), que é geralmente um documento em formato HTML ou XML, onde também pode ser uma imagem ou qualquer outro conteúdo.

Uma aplicação REST pode definir os seguintes tipos de recursos:

Usuario {}

Localizacao {}

Cada recurso tem seu próprio identificador. Os clientes trabalhariam com estes recursos através das operações padrão de HTTP, como o GET para chamar uma cópia do recurso. Observa-se como cada objeto tem sua própria URL e pode ser facilmente "cacheado", copiado e guardado como marcador. POST utiliza-se geralmente para ações com efeitos colaterais, como enviar uma ordem de compra.

Por exemplo, o registo para um Utilizador poderia ter o seguinte aspecto:

```
<usuario>
  <nome>Maria Juana</nome>
  <genero>feminino</genero>
  <localizacao href=http://www.example.org/locations/us/ny/new\_york>
    Nova York, NY, US
  </localizacao>
</usuario>
```

A Web como a conhecemos hoje, funciona seguindo práticas REST, e para entendermos melhor, vamos considerar o exemplo abaixo:

1. Você entra com um endereço em seu navegador (Chrome, Firefox, Edge, etc).
2. Seu navegador estabelece uma conexão TCP/IP com o servidor de destino e envia uma requisição GET HTTP com o endereço que você digitou.
3. O servidor interpreta sua requisição e de acordo com o que foi solicitado, uma resposta HTTP é retornada ao seu navegador.
4. A resposta retornada pode ser de sucesso, contendo alguma representação em formato HTML, ou pode ser algum erro, como por exemplo o famoso 404 Not Found, que indica que o endereço/recurso que você solicitou não pôde ser encontrado.
5. Em caso de sucesso, o seu navegador interpreta o HTML e você consegue navegar pela página renderizada.

Todo esse processo é repetido enquanto você está navegando em alguma página Web. Cada link que você clica ou formulário que submete, efetua os passos que discutimos acima.

Se analisarmos detalhadamente o que discutimos, podemos extrair alguns elementos principais; esses elementos são responsáveis por permitirmos criar aplicações Web da forma que conhecemos hoje, e são esses elementos que vamos detalhar melhor abaixo.

## Recursos

Um recurso é um elemento abstrato e que nos permite mapear qualquer coisa do mundo real como um elemento para acesso via Web.

Se analisarmos o primeiro passo da lista anterior, podemos relembrar que falamos de um endereço. Esse endereço possui basicamente uma parte que nos permite endereçar algo, e é o que chamamos de URL (*Universal Resource Locator*), um exemplo seria <http://www.algaworks.com>.

A partir desse endereço, estamos aptos a acessar algum recurso, que poderia ser por exemplo, cursos ou alunos.

Dito isso, podemos verificar que ao acessarmos por exemplo, o endereço <http://www.algaworks.com/cursos>, é nos retornado uma representação HTML do recurso “cursos” (faça um teste no seu próprio navegador).

Da mesma forma, se acessarmos por exemplo, <http://www.algaworks.com/alunos>, podemos verificar que esse recurso não existe e que uma página indicando a resposta 404 nos será mostrada.

## **Interfaces Uniforme**

Para interagirmos com os recursos que aprendemos no tópico anterior, o HTTP nos fornece uma interface de operações padronizadas, permitindo que possamos criar, atualizar, pesquisar, remover e executar operações sob um determinado recurso.

Além de operações padronizadas, o HTTP fornece um conjunto de respostas para que os clientes (navegadores, APIs, etc) possam saber, de forma adequada, como agir perante uma determinada resposta.

Abaixo, veja uma breve descrição dos principais métodos e respostas.

### **GET**

O método GET é utilizado quando existe a necessidade de se obter um recurso. Ao executar o método GET sob um recurso, uma representação será devolvida pelo servidor.

Em aplicações Web, normalmente é retornado uma representação HTML.

### **POST**

Utilizamos o método POST quando desejamos criar algum recurso no servidor a partir de uma determinada representação. Exemplo disso é quando fazemos a submissão de algum formulário em uma página Web.

### **PUT**

Semelhante ao método POST, a ideia básica do método PUT é permitir a atualização de um recurso no servidor.

### **DELETE**

Como você já deve estar imaginando, o método DELETE é utilizado com o intuito de remover um recurso em um determinado servidor.

## **Respostas**

Baseado nos métodos que discutimos, o servidor deve processar cada uma das requisições e retornar uma resposta adequada. Veja um resumo de cada uma dessas respostas.



1XX – Informações Gerais  
2XX – Sucesso  
3XX – Redirecionamento  
4XX – Erro no cliente  
5XX – Erro no servidor

Para cada tipo que você pode ver, existe uma série de respostas relacionadas. Por exemplo, se o servidor retornar um “200 OK”, significa que o recurso pedido foi retornado com sucesso.

Por outro lado, se o servidor retornar um “404 Not Found”, significa que o recurso que estamos pedindo não foi encontrado.

Apesar de muitas aplicações Web não seguirem o uso adequado de métodos e respostas, devemos sempre que possível, construir nossas aplicações utilizando-as da maneira mais adequada possível.

## Representações

Em aplicações Web, a representação mais utilizada é o HTML. Essa representação é utilizada como forma de encapsular as informações relacionadas a um determinado recurso.

Além do HTML, podemos utilizar XML, JSON, ou algum outro formato que melhor atenda o cenário que estamos desenvolvendo.

É importante citar que um recurso pode ter mais de uma representação, ou seja, podemos construir aplicações que para determinados cenários, retornem representações diferentes, baseado nas necessidades de cada cliente.

## Hypermedia

*Hypermedia* é um dos conceitos mais importantes em aplicações que seguem o modelo REST. É essa característica que permite criarmos aplicações que possam evoluir de formas tão surpreendentes.

Uma representação *hypermedia* trabalha basicamente como um motor de estado, permitindo que clientes naveguem nos mesmos. Cada estado é um documento (uma página HTML) e possui referências para futuros estados (a partir de links).

```
<a href="produtos.html">Produtos</a>  
<a href="clientes.html">Clientes</a>  
<a href="contato.html">Contato</a>
```

Se repararmos no código acima, podemos verificar que temos uma representação HTML que nos permite ir para as próximas páginas a partir do uso de links.

O seu navegador sabe interpretar o significado da tag <a> e criar uma requisição correta para o servidor.

Abaixo vemos um exemplo simplista para algumas rotas definidas.

```
var express = require('express')  
  
var app = express()  
  
// Rota para a home, seu domínio.  
app.get('/', function (req, res) {  
  res.send('Sua home');  
})
```

```
// Rota para /produtos

app.get('/produtos', function (req, res) {
  res.send('exibindo produtos!');
})
```

```
// Rota para /usuarios

app.get('/usuarios', function (req, res) {
  res.send('exibindo usuários');
})
```

```
app.listen(3000)
```

## ENVIANDO EMAIL NO NODE.JS UTILIZANDO O NODemailer

O nodemailer como o próprio site diz é um módulo para aplicações Node.js que permite enviar um e-mail facilmente.

Instalação:

```
npm install nodemailer --save
```

Após o módulo instalado, crie um arquivo app.js com o seguinte código.

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

var $usuario = 'seu email aqui @gmail.com';
var $senha = 'sua senha aqui';

var transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: $usuario,
    pass: $senha
  }
});

var $destinatario = 'Qualquer email';

var mailOptions = {
  from: $usuario,
  to: $destinatario,
  subject: 'Enviando um email pelo Node.js',
  text: 'Muito fácil enviar um email pelo node, tente você também!'
};

transporter.sendMail(mailOptions, function(error, info){
  if (error) {
    console.log(error);
  } else {
    console.log('Email enviado: ' + info.response);
  }
});
```

Para enviar o e-mail eu utilizei um email do gmail, criei uma variável para meu usuário de email e uma para minha senha e depois criei o transporter, que é o responsável por fazer a autenticação no servidor de email.

```
var $usuario = 'seu email aqui @gmail.com';
var $senha = 'sua senha aqui';

var transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: $usuario,
    pass: $senha
  }
});
```

Com o transporter criado temos que criar um objeto que terá as informações do destinatário e a mensagem, este objeto será interpretado pelo nodemailer na hora do envio da mensagem.

```
var $destinatario = 'Qualquer email';

var mailOptions = {
  from: $usuario,
  to: $destinatario,
  subject: 'Enviando um email pelo Node.js',
  text: 'Muito fácil enviar um email pelo node, tente você também!'
};
```

E por fim mandamos o transporter entregar a mensagem, se der algum erro informa o erro no console, senão apresenta uma mensagem de sucesso.

```
transporter.sendMail(mailOptions, function(error, info){
  if (error) {
    console.log(error);
  } else {
    console.log('Email enviado: ' + info.response);
  }
});
```

## Integração de Banco de dados com NodeJS e Express

A inclusão da capacidade de se conectar à banco de dados em aplicativos do Express é apenas uma questão de se carregar um driver Node.js apropriado para o banco de dados no seu aplicativo. Este documento explica brevemente como incluir e utilizar alguns dos mais populares módulos do Node.js para sistemas de bancos de dados no seu aplicativo do Express:

- [Cassandra](#)
- [CouchDB](#)
- [LevelDB](#)
- [MySQL](#)
- [MongoDB](#)
- [Neo4j](#)
- [PostgreSQL](#)
- [Redis](#)
- [SQLite](#)
- [ElasticSearch](#)

# NodeJS + MySQL

## Parte 1: Criando o banco de dados

baixar e instalar o MySQL (gratuito) na sua máquina;

Baixe, instale e crie um novo banco de dados para uso nesse tutorial. Durante a instalação, você irá precisar definir uma senha para o usuário 'root', não esqueça dela.

A segunda opção é bem simples também, mais vai exigir que você tenha

## Parte 2: Criando e populando a tabela

Agora que você já tem o banco pronto, vamos criar uma tabela nele e colocar alguns dados de exemplo. Não pule esta etapa pois vamos fazer tudo isso usando Node.js!

Crie uma pasta para guardar os arquivos do seu projeto Node.js, você pode fazer isso pelo console se quiser, usaremos ele algumas vezes nesse tutorial. No exemplo abaixo, criei a pasta e depois entrei dentro dela.

```
1 > mkdir nodemysql
2 > cd nodemysql
```

Agora execute no console o comando "npm init" que o próprio NPM (gerenciador de pacotes do Node) vai te guiar para a construção do arquivo packages.json, que é o arquivo de configuração do projeto. Se ficar em dúvida ou com preguiça, segue o meu package.json abaixo:

```
1 {
2   "name": "nodemysql",
3   "version": "1.0.0",
4   "description": "tutorial de node com mysql",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \\\"Error: no test specified\\\" && exit 1"
8   },
9   "author": "luiztools",
10  "license": "ISC"
11 }
```

Com o arquivo de configurações criado, vá no console novamente, na pasta do projeto e digite o seguinte comando para instalar a extensão mysql, que permite usar Node com MySQL:

```
nodemysql> npm install -S mysql
```

A flag “-S” diz que é pra salvar essa dependência no arquivo packages.json. Se você abrir o arquivo vai ver que tem coisa nova por lá.

Agora, crie um arquivo create-table.js dentro dessa pasta, que será o arquivo que vai criar e popular nossa tabela que usaremos neste exercício. Também usaremos ele para entender o básico de comandos SQL, conexão com o banco, etc. Vamos começar nosso create-table.js definindo uma constante para a String de conexão com o banco e uma constante para o objeto que vai carregar a extensão mysql (e que mais tarde usaremos para conectar, executar SQL, etc). O código abaixo é auto explicativo:

```
const mysql = require('mysql');  
  
const connection = mysql.createConnection({  
  host : 'XXX',  
  port : XXX,  
  user : 'XXX',  
  password : 'XXX',  
  database : 'XXX'  
});
```

Agora, usaremos esse objeto connection para fazer uma conexão e, em caso de sucesso, imprimir uma mensagem de sucesso. Caso contrário, se der erro, uma mensagem de falha:

```
connection.connect(function(err){  
  if(err) return console.log(err);  
  console.log('conectou!');  
})
```

Se esse código lhe parece muito bizarro, calma, é fácil de entender. O objeto connection permite que façamos coisas no banco de dados, uma delas é a conexão (connect). No entanto, o Node.js trabalha de maneira assíncrona, o que quer dizer que ele não espera pela conexão ser estabelecida. Quando ele terminar de estabelecer a conexão, ele vai executar a função de callback passada por parâmetro, contendo ou não um objeto de erro.

Para executar esse arquivo, abra o console (se estiver usando VS Code, apenas aperta F5 com este arquivo aberto no editor) e na pasta do projeto digite:

```
nodemysql> node create-table.js
```

Agora que sabemos como conectar no MySQL através de Node.js, é hora de executarmos o comando que vai criar a tabela e popular ela, ao invés de simplesmente imprimir “conectou”. Sendo assim, vamos criar uma função JS nesse arquivo pra fazer a criação da tabela:

```
function createTable(conn){

    const sql = "CREATE TABLE IF NOT EXISTS Clientes (\n"+
        "ID int NOT NULL AUTO_INCREMENT,\n"+
        "Nome varchar(150) NOT NULL,\n"+
        "CPF char(11) NOT NULL,\n"+
        "PRIMARY KEY (ID)\n"+
        ");";

    conn.query(sql, function (error, results, fields){
        if(error) return console.log(error);
        console.log('criou a tabela!');
    });
}
```

Coloque a chamada desta função no callback após a conexão no banco, passando o objeto conn por parâmetro, como abaixo:

```
connection.connect(function(err){
    if(err) return console.log(err);
    console.log('conectou!');
    createTable(connection);
})
```

Mande rodar esse arquivo novamente e verá que ele criará a sua tabela com sucesso. Para adicionar algumas linhas de exemplo, vamos criar outra função que vai fazer um bulk insert no MySQL via Node.js (inserção de várias linhas de uma vez):

```
function addRows(conn){

    const sql = "INSERT INTO Clientes(Nome,CPF) VALUES ?";

    const values = [
```

```

    ['teste1', '12345678901'],
    ['teste1', '09876543210'],
    ['teste3', '12312312399']
  ];
  conn.query(sql, [values], function (error, results, fields){
    if(error) return console.log(error);
    console.log('adicionou registros!');
    conn.end();//fecha a conexão
  });
}

```

Essa função deve ser chamada dentro do callback da query que criou a tabela, logo abaixo de onde diz “console.log(‘criou a tabela!’);”. Se não quiser fazer isso dessa maneira, você pode fazer pela sua ferramenta de gerenciamento do MySQL (como o [MySQL Workbench](#)) ou pelo [Visual Studio](#), se estiver usando ele para programar Node.js.

### Parte 3: Criando a API

Agora que já temos nosso banco de dados MySQL pronto, com dados de exemplo e aprendemos como fazer a conexão nele, vamos criar uma API básica usando Express para conseguir criar um CRUD com Node.js + MySQL no próximo passo. Se já sabe como montar uma API básica com Node + Express, pule esta parte.

Vamos começar adicionando a dependência do Express (framework web) e do Body-Parser (parser para os POSTs futuros) no projeto via linha de comando na pasta do mesmo:

```
1 nodesqlserver> npm install -S express body-parser
```

Na sequência, vamos criar um arquivo index.js na pasta do projeto onde vamos criar o nosso servidor da API para tratar as requisições que chegarão em breve. Vamos começar bem simples, apenas definindo as constantes locais que serão usadas mais pra frente:

```

1 const express = require('express');
2 const app = express();
3 const bodyParser = require('body-parser');
4 const port = 3000; //porta padrão
5 const mysql = require('mysql');

```

Agora, logo abaixo, vamos configurar nossa aplicação (app) Express para usar o body parser que carregamos da biblioteca body-parser, permitindo que recebamos mais tarde POSTs nos formatos URLEncoded e JSON:

```
1 //configurando o body parser para pegar POSTS mais tarde
```

```
2 app.use(bodyParser.urlencoded({ extended: true }));
3 app.use(bodyParser.json());
```

Na sequência, vamos criar um roteador e dentro dele definir uma regra inicial que apenas exibe uma mensagem de sucesso quando o usuário requisitar um GET na raiz da API (/) para ver se está funcionando.

```
1 //definindo as rotas
2 const router = express.Router();
3 router.get('/', (req, res) => res.json({ message: 'Funcionando!' }));
4 app.use('/', router);
```

Note que na última linha eu digo que requisições que chegarem na raiz devem ser mandadas para o router. Por fim, adicionamos as linhas abaixo no final do arquivo que dão o start no servidor da API:

```
1 //inicia o servidor
2 app.listen(port);
3 console.log('API funcionando!');
```

Teste sua API executando via console o seu index.js com o comando 'node index.js'. Você deve ver a mensagem de 'API funcionando!' no console, e se acessar no navegador localhost:3000 deve ver o JSON default que deixamos na rota raiz!



API Funcionando

#### Parte 4: Criando a listagem de clientes

Agora que temos uma API funcionando, vamos adicionar uma rota /clientes que listará todos os clientes do banco de dados. Para fazer isso, primeiro vamos criar uma função que executará consultas SQL no banco usando uma conexão que será criada a cada uso (existem técnicas mais avançadas para maior performance, mas por ora, isso resolve satisfatoriamente), como abaixo:

```
1 function execSQLQuery(sqlQry, res){
2   const connection = mysql.createConnection({
3     host   : 'XXX',
4     port   : XXX,
5     user    : 'XXX',
6     password : 'XXX',
7     database : 'XXX'
8   });
9
10  connection.query(sqlQry, function(error, results, fields){
11    if(error)
12      res.json(error);
13    else
14      res.json(results);
15    connection.end();
16    console.log('executou!');
17  });
18 }
```



Esta função pode ficar no final do seu arquivo index.js e nós a usaremos para fazer todas as operações de banco de dados da nossa API, começando com consultar todos os clientes. Para isso, começaremos criando a rota /clientes logo abaixo da rota / (raiz):

```
1 router.get('/clientes', (req, res) =>{
2   execSQLQuery('SELECT * FROM Clientes', res);
3 })
```

Agora, ao executarmos novamente nosso projeto e ao acessarmos a URL localhost:3000/clientes, veremos todos os clientes cadastrados no banco de dados (no passo 2, lembra?):



E com isso finalizamos a listagem de todos clientes na nossa API!

## Parte 5: Criando a pesquisa de um cliente

Agora, se o usuário quiser ver apenas um cliente, ele deverá passar o ID do mesmo na URL, logo após o /clientes. Para fazer isso, vamos modificar nossa rota criada no passo anterior, /clientes, para aceitar um parâmetro opcional ID. Além disso, dentro do processamento da rota, se vier o ID, devemos fazer uma consulta diferente da anterior, como mostra o código abaixo, com os ajustes na mesma rota do passo anterior:

```
1 router.get('/clientes/:id?', (req, res) =>{
2   let filter = '';
3   if(req.params.id) filter = ' WHERE ID=' + parseInt(req.params.id);
4   execSQLQuery('SELECT * FROM Clientes' + filter, res);
5 })
```

O `parseInt` que coloquei é apenas uma proteção contra SQL Injection uma vez que neste caso o ID deve ser um inteiro válido. Não é a melhor forma de resolver isso, mas vai nos atender por enquanto sem ter de entrar em conceitos mais avançados. Manda rodar e teste no navegador, verá que está funcionando perfeitamente!



um

E com isso terminamos a pesquisa por cliente.

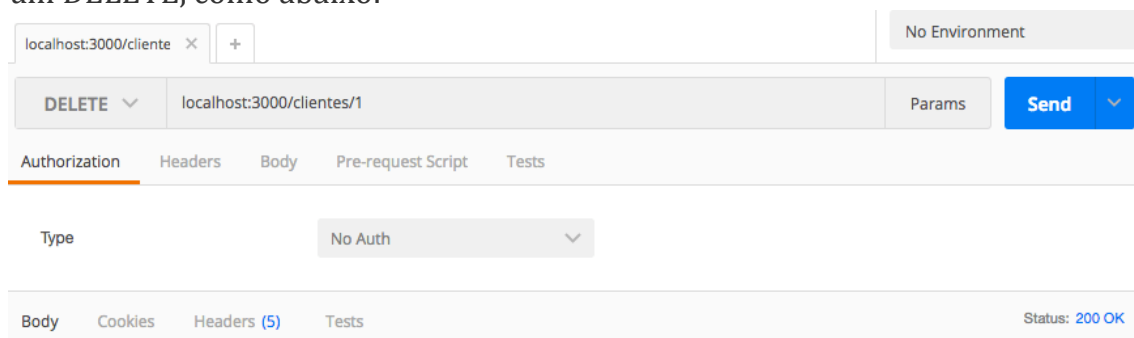
## Parte 6: Excluindo um cliente

Para excluir um cliente vamos fazer um processo parecido com o de pesquisar um cliente, no entanto, mudaremos o verbo HTTP de GET para DELETE, como manda o protocolo. Adicione a nova rota logo após as demais:

```
1 router.delete('/clientes/:id', (req, res) =>{  
2   execSQLQuery('DELETE FROM Clientes WHERE ID=' + parseInt(req.params.id), res);  
3 })
```

Note que desta vez o parâmetro id na URL não é opcional (não usei ? após :id). E dentro do processamento da requisição delete do router eu mando um SQL de DELETE passando o ID numérico.

Para testar essa rota você tem duas alternativas, ou usa o POSTMAN para forjar um DELETE, como abaixo:



### DELETE com POSTMAN

Ou fazer via console usando cURL (se tiver ele instalado na sua máquina):

```
1 > curl -X DELETE http://localhost:3000/clientes/1
```

Em ambos os casos você deve obter uma resposta 200 OK (caso não tenha dado erro) e se mandar listar todos clientes novamente, verá que o número 1 sumiu.

## Parte 7: Adicionando um cliente

Agora vamos adicionar um novo cliente com um POST na rota /clientes. Adicione esta nova rota logo abaixo das anteriores.

```
1 router.post('/clientes', (req, res) =>{  
2   const nome = req.body.nome.substring(0,150);  
3   const cpf = req.body.cpf.substring(0,11);  
4   execSQLQuery('INSERT INTO Clientes(Nome, CPF) VALUES('${nome}','${cpf}')', res);  
5 });
```

Nela, eu pego as variáveis que devem vir junto ao POST, faço algumas validações de tamanho e tipo de dado e depois junto elas a um comando de INSERT que vai ser executado no banco de dados.

Para testar esse POST, você usar o POSTMAN, como mostrado anteriormente:

The screenshot shows the Postman interface for a POST request to `localhost:3000/clientes/`. The 'Body' tab is selected, and the 'x-www-form-urlencoded' format is chosen. The body contains two key-value pairs: `nome` with value `luiz` and `cpf` with value `12345678901`. The status bar at the bottom indicates a successful response with `Status: 200 OK`.

Key	Value	Description
<input checked="" type="checkbox"/> nome	luiz	
<input checked="" type="checkbox"/> cpf	12345678901	
New key	Value	Description

## POST no Postman

Se quiser fazer via cURL:

```
1 > curl -X POST -d "nome=luiz&cpf=12345678901" http://localhost:3000/clientes
```

Também podemos permitir outras maneiras de passar os dados com a requisição POST, como através de JSON ou através da URL, mas isso foge do escopo deste artigo que deve focar mais no CRUD com MySQL + Node.

Se testar agora vai ver que é possível inserir novos registros no banco de dados através de requisições POST.

## Parte 8: Atualizando um cliente

E para finalizar o CRUD, vamos ver como podemos atualizar um cliente no banco de dados SQL Server através da nossa API Node.js. Para fazer updates podemos usar os verbos PUT ou PATCH. O protocolo diz que devemos usar PUT se pretendemos passar todos os parâmetros da entidade que está sendo atualizada, mas não vamos alterar jamais o ID, então usaremos PATCH nesta API.

Crie uma rota PATCH em `/clientes` esperando o ID do cliente a ser alterado.

```
1 router.patch('/clientes/:id', (req, res) =>{
2   const id = parseInt(req.params.id);
3   const nome = req.body.nome.substring(0,150);
4   const cpf = req.body.cpf.substring(0,11);
5   execSQLQuery(`UPDATE Clientes SET Nome='${nome}', CPF='${cpf}' WHERE ID=${id}`, res);
6 })
```

No código acima, pegamos o ID que veio na URL e as demais informações que vieram no corpo da requisição, fazendo as mesmas validações que já havia feito antes (podemos melhorar a segurança aqui). Depois monto o UPDATE com as variáveis locais e mando para nossa função de executar SQL.

Para testar uma requisição PATCH, você pode usar o POSTMAN:

The screenshot shows a Postman interface for a PATCH request to `localhost:3000/clientes/4`. The 'Body' tab is selected, showing a table with two rows: 'nome' with value 'fernando' and 'cpf' with value '12345678901'. The status bar at the bottom indicates 'Status: 200 OK'.

Key	Value	Description
<input checked="" type="checkbox"/> nome	fernando	
<input checked="" type="checkbox"/> cpf	12345678901	
New key	value	description

## PATCH no POSTMAN

Ou o cURL:

```
1 > curl -X PATCH -d "nome=fernando&cpf=12345678901" http://localhost:3000/clientes/4
```

O resultado é o mesmo: o cliente cujo ID=4 vai ter o seu nome alterado para 'fernando'. Note que se ele não existir, ocasionará um erro que será apresentado no corpo da resposta.

E com isso finalizamos o CRUD da nossa API Node.js que usa MySQL como persistência de dados.

## Bônus 1: ORM

Se você não curte muito a ideia de ficar usando SQL no meio dos seus códigos JS, experimente usar alguma biblioteca ORM (Object-Relational Mapping) como o Sequelize. Nunca usei, mas muita gente recomenda por possuir suporte a MySQL, SQL Server, PostgreSQL e SQLite.

## Bônus 2: Boas práticas

Não quis deixar o tutorial muito extenso e por isso deixei de passar algumas boas práticas. Como foi levantada esta questão no grupo do Facebook de Node.js pelo usuário Ícaro Tavares, achei válido trazer as sugestões para cá. Se você está lendo esse bônus é porque se interessa em saber mais do que o básico e isso é ótimo, então lá vai.

### SQL Injection

O código que forneci nos exemplos concatenando strings para formar uma query SQL deixa uma brecha potencialmente nociva no código da sua API chamada SQL Injection. Para evitar isso, você pode usar '?' no lugar das variáveis que devem ser substituídas e passar um array de variáveis para substituição como segundo parâmetro da função query.

*Como assim?*

Para um dado comando SQL como:

```
1 const consulta = "SELECT * FROM tabela WHERE id = ?";
```

note que coloquei o '?' no lugar do valor do id, e poderia fazer isso para quantos filtros minha consulta tiver. Agora, quando chamar a função query do módulo mysql, farei da seguinte forma, passando um array de variáveis/valores para preencher esses '?', na mesma ordem que foram descritos na consulta:

```
1 conn.query(consulta, [id], callback);
```

Dessa forma, meu código não fica mais suscetível à SQL Injection!

### **Conexões auto-gerenciadas**

No código que forneci, nós temos total controle sobre a abertura e fechamento de conexões. Como boa prática, você pode deixar para a própria função query se encarregar de abrir e fechar essas conexões, se livrando desse “microgerenciamento”.

Em meus testes pessoais notei que isso pode ser melhor ou pior que o exemplo que mostrei no tutorial, considerando peculiaridades da sua infraestrutura de MySQL. Na dúvida, o meu código não tem a melhor performance, mas a garantia de funcionamento mesmo em hospedagens compartilhadas, onde não temos tanto controle das configurações do MySQL.

### **SELECT \***

A menos que você sempre vá usar todas as informações da sua tabela, jamais use SELECT \*, mas sim a versão extensão da consulta SQL em que você passa todas as colunas da tabela que deseja retornar. Pense em um SELECT como sendo um download do seu banco de dados, quanto menos colunas você incluir no SELECT, mais rápido será esse download pois menos bytes serão retornados.

## **NODEJS – CARREGANDO HTML EM ARQUIVO SEPARADO.**

Neste tutorial vou mostrar como carregar um arquivo html externo, também usaremos rotas e teremos um mini log de acesso as páginas, por isso recomendo que antes leia:

[Primeiros passos com NodeJS.](#)

[Rotas com NodeJS.](#)

[NodeJS – File System – Trabalhando com arquivos.](#)

## Início.

Para começar criei uma pasta para organizar o projeto e dentro dela mais duas pasta, uma com o nome de html e outra com o nome de log, que conterão respectivamente nossos arquivos html e nosso arquivo de log.

## Arquivos html.

Primeiro vamos criar os nossos arquivos html, serão 4 arquivos, index.html, contato.html, blog.html e erro.html.

### ***index.html***

```
<html>
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title> Html carregado por file system</title>
</head>
<body>
  <h1>Olá mundo</h1>
  <h4>Primeira página</h4>
</body>
</html>
```

As próximas páginas terão o mesmo formato, o que mudará é o corpo.

### ***contato.html***

```
<body>
  <h1>Contato</h1>
  <h4>Bem vindo a página de contato!</h4>
</body>
```

### ***blog.html***

```
<body>
  <h1>Blog</h1>
  <h4>Bem vindo ao meu blog!</h4>
</body>
```

### ***erro.html***

```
<body>
  <h1>Erro 404</h1>
  <h4>Página não encontrada!</h4>
</body>
```

## Servidor.js

Após os arquivos html criados criaremos o arquivo js.

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (request, response) {
  fs.readFile(__dirname + "/html/index.html", function(err, data) {
    response.end(data);
  });
});

server.listen(3000, function () {
  console.log('Servidor rodando na porta 3000');
```

```
});
```

Este será nosso primeiro teste. Abriremos a página index.html que está dentro do diretório.

A constante `__dirname` retorna o diretório raiz da aplicação, então estou passando o caminho fixo do arquivo e como sabemos a função de callback recebe dois parâmetros, o erro e o próprio arquivo.

Como passei um caminho fixo não fiz o teste para saber se daria erro, somente mandei escrever o html na tela e pronto, já estamos carregando nossa primeira página externa.

Entretanto o que queremos é que ele carregue as páginas que estão na nossa url, para isso modificaremos nosso código.

```
var http = require('http');
var fs = require('fs');
var url = require('url');

var pathPage = function(page) {
    return __dirname + "/html/" + page + ".html";
};

var router = function(pathname) {
    if(pathname && pathname !== "/") {
        var exist = fileExists( pathPage(pathname) );
        return exist ? pathPage(pathname) : pathPage("erro");
    }
    return pathPage("index");
};

var fileExists = function(filePath) {
    try{
        return fs.statSync(filePath).isFile();
    } catch (err) {
        return false;
    }
};

var server = http.createServer(function (request, response) {
    var page = router( url.parse(request.url).pathname );
    fs.readFile(page, function(err, data) {
        response.end(data);
    });
});

server.listen(3000, function () {
    console.log('Servidor rodando na porta 3000');
});
```

## Explicando o código acima.

A primeira modificação foi a inclusão do modulo 'url'.

Após isso criei uma função pathPage() que é responsável por concatenar o caminho da página, para que não seja preciso escrever várias vezes.

A segunda função criada foi a router(), esta função testará nossas rotas para ver se existem ou não e retornará o caminho completo.

O primeiro teste dentro da router() é saber se recebeu alguma rota e se essa rota é diferente de "/", pois se for igual ou não receber rota nenhuma chamará a página principal(index.html).

Se a rota for diferente de "/" atribuo a uma variável exist o retorno da função fileExists();

A função `fileExists` testa se o caminho que passamos é um arquivo, se for retorna `true` se não for dá um erro e fazemos retornar `false`, por isso o `try catch`.

Para mais informações [consulte a documentação do node.js sobre o stats](#).

Antigamente poderíamos ter feito a validação desta forma.

```
var exist = fs.existsSync( pathPage(pathname) );
return exist ? pathPage(pathname) : pathPage("erro");
```

Sem precisar criar a função `fileExists`, porém o [existsSync foi depreciado](#).

E por último mudamos nossa função `createServer()`;

Ela terá uma variável `page` que recebera qual caminho tem que abrir da rota.

## Log da aplicação.

E para finalizar iremos criar o log da aplicação.

É nem simples, dentro da função `createServer()`, adicionaremos a linha:

```
var msg = page + " acessada em " + new Date() + "\n";
```

E depois:

```
fs.writeFile(pathLog, msg, {encoding : 'utf-8', flag: 'a'}, function (err)
{
    if (err) throw err;
});
```

Caso tenha alguma dúvida no código acima leia:

[NodeJS – File System – Trabalhando com arquivos](#).

Pronto agora temos nossa aplicação, renderizando arquivos `html` externos, mapeando as rotas e gravando um log.