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).

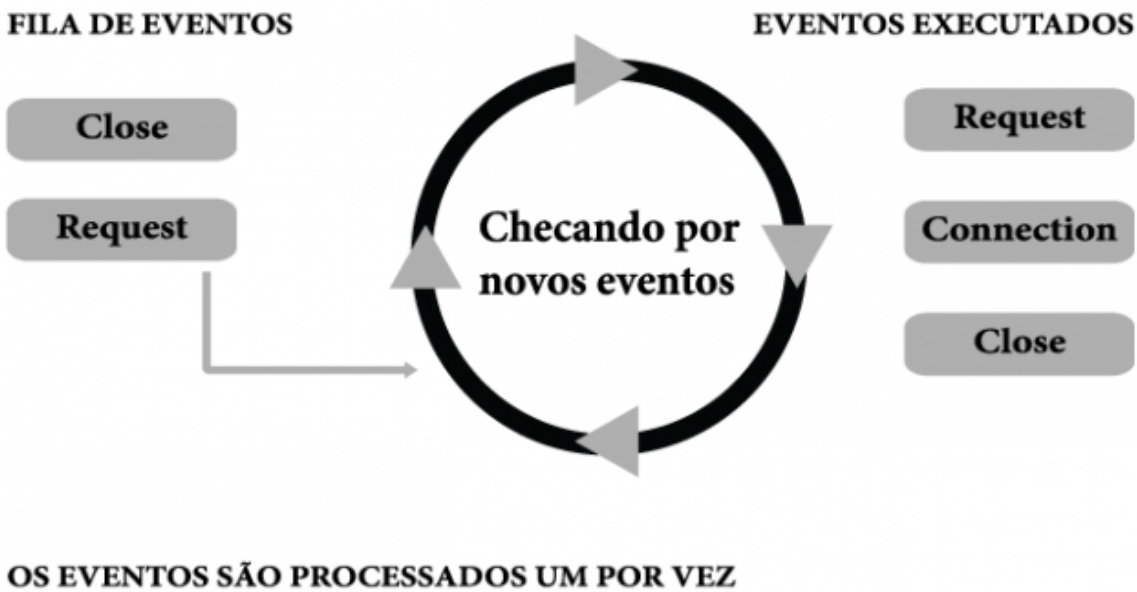
É por isso que o Node.js trabalha com assincronismo. Ele permite que você desenvolva um sistema totalmente orientado a eventos, tudo isso graças ao Event-loop.

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?



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
};
```

```
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>"
}
```

Importante Bibliotecas e componentes externos

O **require** existe só em CommonJS (a maneira que o Node.js criou para importar e exportar modulos 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';

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

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 metódo';
      }
```

```
}

var objeto = new Exemplo(); //Instância do construtor "Exemplo"

//Alerta os respectivos textos na tela
alert(objeto.propriedade),
alert(objeto.metodo());
```

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.ServerResponde
- 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 raíz "/"** 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.writeHeader(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 retona uma string referente ao endereço raíz 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 perfomáticas** e normalmente **não bloqueiam** a execução de outras **rotinas síncronas** do seu sistema, mesmo quando ocorrem problemas durante suas execuções.

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',{enconding:'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 existenstes 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);
        });
});
```

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 síntaxe 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 apartir 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_usuário**: 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.