



# WebAPIs em Node.js

LuizTools

# SUMÁRIO

Sobre o Autor	4
Antes de Começar	8
Para quem é este livro	9
1. Introdução ao Node.js	10
História do Node.js	11
Características do Node.js	12
Por que Node.js?	14
2. Configurando o Ambiente	18
Instalando o Node.js	19
Visual Studio Code	22
Google Chrome	26
MongoDB	26
Postman	27
3. ExpressJS	28
Routes e Views	35
Event Loop	42
O Problema	42
A solução	43
Task/Event/Message Queue	47
4. Mongodb	49
Introdução ao MongoDB	50
Quando devo usar MongoDB?	52
Quando não devo usar MongoDB?	53
Instalação e Testes	54
Comandos elementares	57
Find avançado	58

Delete	64
5. Criando Web APIs	66
Criando uma Web API	68
MongoDB Driver	69
Listando os clientes na API	75
Atualizando clientes	82
Excluindo clientes	87
Seguindo em frente	90
Curtiu o Livro?	92

# SOBRE O AUTOR

---

Luiz Fernando Duarte Júnior é Bacharel em Ciência da Computação pela Universidade Luterana do Brasil (ULBRA, 2010) e Especialista em Desenvolvimento de Aplicações para Dispositivos Móveis pela Universidade do Vale do Rio dos Sinos (UNISINOS, 2013).

Carrega ainda um diploma de Reparador de Equipamentos Eletrônicos (SENAI, 2005) e duas certificações Scrum para trabalhar com Métodos Ágeis: Professional Scrum Developer e Professional Scrum Master (ambas em 2010).

Atuando na área de TI desde 2006, na maior parte do tempo como desenvolvedor web, é apaixonado por desenvolvimento de software desde que teve o primeiro contato com a linguagem Assembly no curso de eletrônica. De lá para cá teve oportunidade de utilizar diferentes linguagens em diferentes sistemas, mas principalmente com tecnologias web, incluindo ASP.NET, JSP e, nos últimos tempos, Node.js.

*Foi amor à primeira vista e a paixão continua a crescer!*

Trabalhando com Node.js desenvolveu diversos projetos para empresas de todos os tamanhos, desde grandes empresas como Softplan até startups como Busca Acelerada e Só Famosos, além de ministrar palestras e cursos de Node.js para alunos do curso superior de várias universidades e eventos de tecnologia.

Um grande entusiasta da plataforma, espera que com esse livro possa ajudar ainda mais pessoas a aprenderem a desenvolver softwares com Node.js e aumentar a competitividade das empresas brasileiras e a empregabilidade dos profissionais de TI.

Além de viciado em desenvolvimento, atua como Agile Coach no Banco Agiplan, como Developer Evangelist na Umbler e é autor do blog [www.luiztools.com.br](http://www.luiztools.com.br), onde escreve semanalmente sobre empreendedorismo e desenvolvimento de software, bem como mantenedor da página **LuizTools** no Facebook, com o mesmo propósito. Entre em contato, o autor está sempre disposto a ouvir e ajudar seus leitores.



Aproveita e me segue nas redes sociais:



[Facebook.com/luiztools](https://Facebook.com/luiztools)



[Twitter.com/luiztools](https://Twitter.com/luiztools)



[LinkedIn.com/in/luizfduartejr](https://LinkedIn.com/in/luizfduartejr)

Conheça meus outros livros:



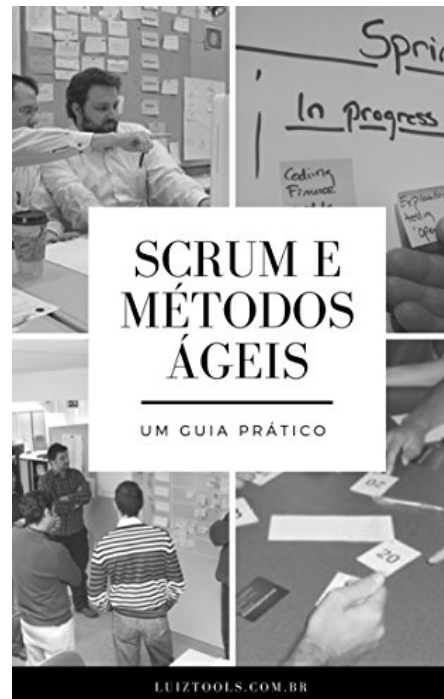
[Programação Web  
com Node.js](#)



[MongoDB para  
Iniciantes](#)

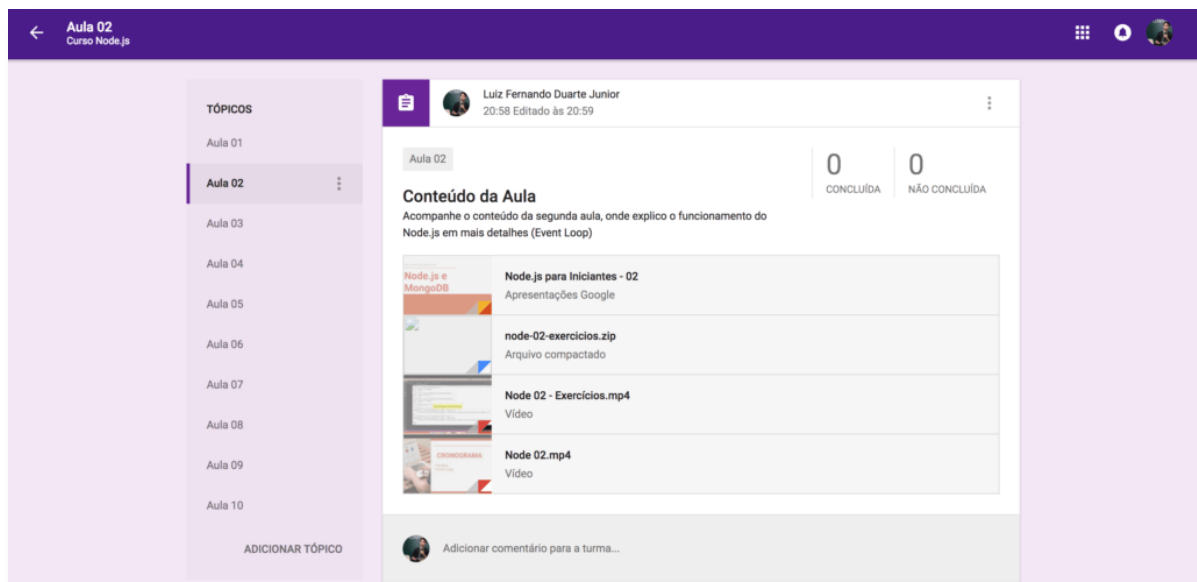


**Criando apps para  
empresas com  
Android**



**Scrum e Métodos  
Ágeis: Um Guia Prático**

Meus Cursos:



**Node.js e MongoDB**

# ANTES DE COMEÇAR

---

“

Without requirements and design, programming is  
the art of adding bugs to an empty text file.

- Louis Srygley

”



Antes de começarmos, é bom você ler esta seção para evitar surpresas e até para saber se este livro é para você.

## Para quem é este livro

Primeiramente, este ebook vai lhe ensinar os primeiros passos de programação de WebAPIs com Node.js, mas não vai lhe ensinar lógica básica e algoritmos, ele exige que você já saiba isso, ao menos em um nível básico (final do primeiro semestre da faculdade de computação, por exemplo).

Parto do pressuposto que você é ou já foi um estudante de Técnico em informática, Ciência da Computação, Sistemas de Informação, Análise e Desenvolvimento de Sistemas ou algum curso semelhante. Usarei diversos termos técnicos ao longo do livro que são comumente aprendidos nestes cursos e que não tenho o intuito de explicar aqui.

O foco deste livro é ensinar alguns poucos aspectos da programação usando Node.js para quem já sabe sobre o básico de web com alguma outra linguagem como PHP e ASP ou está apenas começando nessa plataforma, incluindo algumas “pinceladas” em MongoDB.

Ao término deste livro você estará apto a construir web APIs simples em Node.js e buscar materiais mais avançados para começar a construir softwares profissionais usando estas tecnologias, inclusive o meu livro **Programação Web com Node.js** e o meu curso **Node.js e MongoDB**, que são muito mais completos nesse sentido.

---

# INTRODUÇÃO AO NODE.JS

“

Good software, like wine, takes time.

- Joel Spolsky

”

Node.js é um ambiente de execução de código JavaScript no lado do servidor, open-source e multiplataforma. Historicamente, JavaScript foi criado para ser uma linguagem de scripting no lado do cliente, embutida em páginas HTML que rodavam em navegadores web. No entanto, Node.js permite que possamos usar JavaScript como uma linguagem de scripting server-side também, permitindo criar conteúdo web dinâmico antes da página aparecer no navegador do usuário. Assim, Node.js se tornou um dos elementos fundamentais do paradigma “full-stack” JavaScript, permitindo que todas as camadas de um projeto possa ser desenvolvida usando apenas essa linguagem.

Node.js possui uma arquitetura orientada a eventos capaz de operações de I/O assíncronas. Esta escolha de design tem como objetivo otimizar a vazão e escala de requisições em aplicações web com muitas operações de entrada e saída (request e response, por exemplo), bem como aplicações web real-time (como mensageria e jogos). Basicamente ele aliou o poder da comunicação em rede do Unix com a simplicidade da popular linguagem JavaScript, permitindo que rapidamente milhões de desenvolvedores ao redor do mundo tivessem proficiência em usar Node.js para construir rápidos e escaláveis webserver sem se preocupar com threading.

## História do Node.js

Node.js foi originalmente escrito por Ryan Dahl em 2009 e não foi exatamente a primeira tentativa de rodar JavaScript no lado do servidor, uma vez que 13 anos antes já havia sido criado o Netscape LiveWire Pro Web. Ele inicialmente funcionava apenas em Linux e Mac OS X mas cresceu rapidamente com o apoio da empresa Joyent, onde Dahl trabalhava. Ele conta em diversas entrevistas que foi inspirado a criar o Node.js após ver uma barra de progresso de upload no Flickr. Ele entendeu que o navegador tinha de ficar perguntando para o servidor quanto do arquivo faltava a ser transmitido, pois ele não tinha essa informação, e que isso era um desperdício de tempo e recursos. Ele queria criar um jeito mais fácil de fazer isso.

Suas pesquisas nessa área levaram-no a criticar as possibilidades limitadas do servidor web Apache de lidar (em 2009) com conexões concorrentes e a forma como se criava código web server-side na época que bloqueava os recursos do servidor web a todo momento o que fazia

com que eles tivessem de criar diversas stacks de tarefas em caso de concorrência para não ficarem travados, gerando um grande overhead.

Dahl demonstrou seu projeto no primeiro JSConf europeu em 8 de novembro de 2009 e consistia da engine JavaScript V8 do Google, um evento loop e uma API de I/O de baixo nível (escrita em C++ e que mais tarde se tornaria a libuv), recebendo muitos elogios do público na ocasião. Em janeiro de 2010 foi adicionado ao projeto o NPM, um gerenciador de pacotes que tornou mais fácil aos programadores publicarem e compartilharem códigos e bibliotecas Node.js simplificando a instalação, atualização e desinstalação de módulos, aumentando rapidamente a sua popularidade.

Em 2011 a Microsoft ajudou o projeto criando a versão Windows de Node.js, lançando-a em julho deste ano e daí em diante nunca mais parou de crescer, atualmente sendo mantido pela Node.js Foundation, uma organização independente e sem fins lucrativos que mantém a tecnologia com o apoio da comunidade mundial de desenvolvedores.

## Características do Node.js

### **Node.js é uma tecnologia assíncrona que trabalha em uma única thread de execução**

Por assíncrona entenda que cada requisição ao Node.js não bloqueia o processo do mesmo, atendendo a um volume absurdamente grande de requisições ao mesmo tempo mesmo sendo single thread.

Imagine que existe apenas um fluxo de execução. Quando chega uma requisição, ela entra nesse fluxo, a máquina virtual JavaScript verifica o que tem de ser feito, delega a atividade (consultar dados no banco, por exemplo) e volta a atender novas requisições enquanto este processamento paralelo está acontecendo. Quando a atividade termina (já temos os dados retornados pelo banco), ela volta ao fluxo principal para ser devolvida ao requisitante.

Isso é diferente do funcionamento tradicional da maioria das linguagens de programação, que trabalham com o conceito de multi-threading, onde, para cada requisição recebida, cria-se uma nova thread para atender à mesma. Isso porque a maioria das linguagens tem comportamento bloqueante na thread em que estão, ou seja, se

uma thread faz uma consulta pesada no banco de dados, a thread fica travada até essa consulta terminar.

Esse modelo de trabalho tradicional, com uma thread por requisição é mais fácil de programar, mas mais oneroso para o hardware, consumindo muito mais recursos.

### **Node.js não é uma linguagem de programação.**

Você programa utilizando a linguagem Javascript, a mesma usada há décadas no client-side das aplicações web. Javascript é uma linguagem de scripting interpretada, embora seu uso com Node.js guarde semelhanças com linguagens compiladas, uma vez que máquina virtual V8 (veja mais adiante) faz etapas de pré-compilação e otimização antes do código entrar em operação.

### **Node.js não é um framework Javascript.**

Ele está mais para uma plataforma de aplicação (como um Nginx?), na qual você escreve seus programas com Javascript que serão compilados, otimizados e interpretados pela máquina virtual V8. Essa VM é a mesma que o Google utiliza para executar Javascript no browser Chrome, e foi a partir dela que o criador do Node.js, Ryan Dahl, criou o projeto. O resultado desse processo híbrido é entregue como código de máquina server-side, tornando o Node.js muito eficiente na sua execução e consumo de recursos.

Devido a essas características, podemos traçar alguns cenários de uso comuns, onde podemos explorar o real potencial de Node.js:

### **Node.js serve para fazer APIs.**

Esse talvez seja o principal uso da tecnologia, uma vez que por default ela apenas sabe processar requisições. Não apenas por essa limitação, mas também porque seu modelo não bloqueante de tratar as requisições o torna excelente para essa tarefa consumindo pouquíssimo hardware.

### **Node.js serve para fazer backend de jogos, IoT e apps de mensagens.**

Sim eu sei, isso é praticamente a mesma coisa do item anterior, mas realmente é uma boa ideia usar o Node.js para APIs nestas circunstâncias (backend-as-a-service) devido ao alto volume de requisições que esse tipo de aplicações efetuam.

## **Node.js serve para fazer aplicações de tempo real.**

Usando algumas extensões de web socket com Socket.io, Comet.io, etc é possível criar aplicações de tempo real facilmente sem onerar demais o seu servidor como acontecia antigamente com Java RMI, Microsoft WCF, etc.

## **Por que Node.js?**

Algumas características do JavaScript e conseqüentemente do Node.js têm levado a uma adoção sem precedentes desta plataforma no mercado mundial de desenvolvimento de software. Algumas delas são:

### **Node.js utiliza a linguagem JavaScript**

JavaScript tem algumas décadas de existência e milhões de programadores ao redor do mundo. Qualquer pessoa sai programando em JS em minutos (não necessariamente bem) e você contrata programadores facilmente para esta tecnologia. O mesmo não pode ser dito das plataformas concorrentes.

### **Node.js permite Javascript full-stack**

Uma grande reclamação de muitos programadores web é ter de trabalhar com linguagens diferentes no front-end e no back-end. Node.js resolve isso ao permitir que você trabalhe com JS em ambos e a melhor parte: nunca mais se preocupe em ficar traduzindo dados para fazer o front-end se comunicar com o backend e vice-versa. Você pode usar JSON para tudo.

Claro, isso exige uma arquitetura clara e um tempo de adaptação, uma vez que não haverá a troca de contexto habitual entre o client-side e o server-side. Mas quem consegue, diz que vale muito a pena.

### **Node.js é muito leve e é multiplataforma**

Isso permite que você consiga rodar seus projetos em servidores abertos e com o SO que quiser, diminuindo bastante seu custo de hardware (principalmente se estava usando Java antes) e software (se pagava licenças de Windows). Só a questão de licença de Windows que você economiza em players de datacenter como Amazon chega a 50% de economia, fora a economia de hardware que em alguns projetos meus chegou a 80%.

## **Ecosistema gigantesco**

Desde a sua criação, as pessoas têm construído milhares de bibliotecas de código-aberto para Node.js, a maioria delas hospedadas no site do NPM (que já possui mais de 475 mil extensões) e outras tantas no GitHub. Além de bibliotecas, tem-se desenvolvido muitos frameworks de servidores para acelerar o desenvolvimento de aplicações como Connect, Express.js, Socket.IO, Koa.js, Hapi.js, Sails.js, Meteor, Derby e muitos outros, o que aumentou a popularidade de Node.js dentro do uso corporativo.

## **Aceitação da tecnologia no ambiente corporativo**

Hoje Node.js é adotado por muitas empresas, incluindo grandes nomes como GoDaddy, Groupon, IBM, LinkedIn, Microsoft, Netflix, PayPal, Rakuten, SAP, Tuenti, Voxer, Walmart, Yahoo!, e Cisco Systems.

## **IDEs e ferramentas**

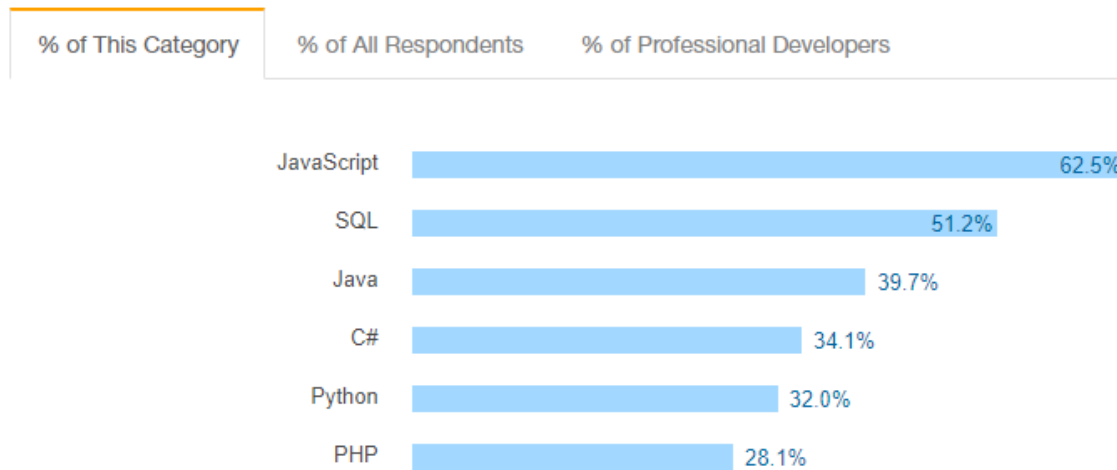
Ambientes de desenvolvimento modernos fornecem recursos de edição e debugging específicas para aplicações Node.js como Atom, Brackets, JetBrains WebStorm, Microsoft Visual Studio, NetBeans, Nodeclipse e Visual Studio Code. Também existem diversas ferramentas web como Codeanywhere, Codenvy, Cloud9 IDE, Koding e o editor de fluxos visuais Node-RED.

Não existem números oficiais a respeito da adoção de Node.js ao redor do mundo, mas existem algumas pesquisas que podem dar uma luz à essa questão. O gráfico abaixo mostra a StackOverflow Developer Survey 2017, onde milhares de usuários do site responderam qual(is) linguagem(ns) de programação ele(s) usa(m) no dia-a-dia, a resposta não pode ser menos óbvia, com o JavaScript figurando na primeira posição.



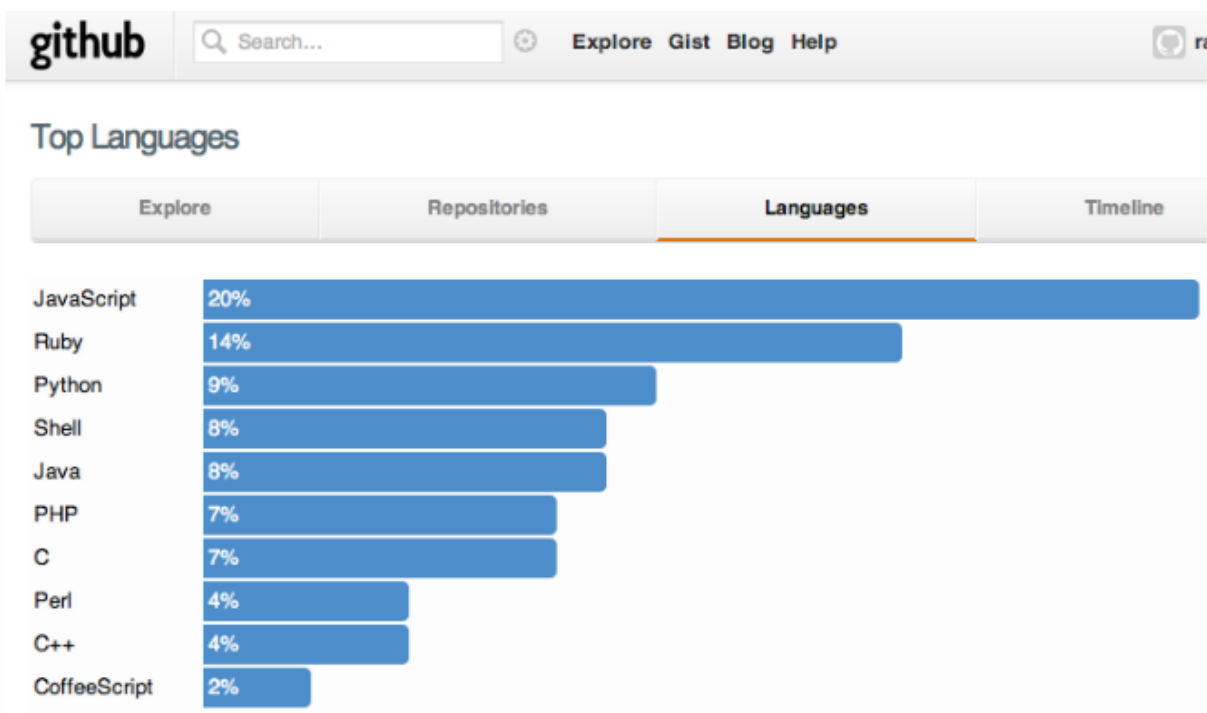
## Most Popular Technologies

### Programming Languages



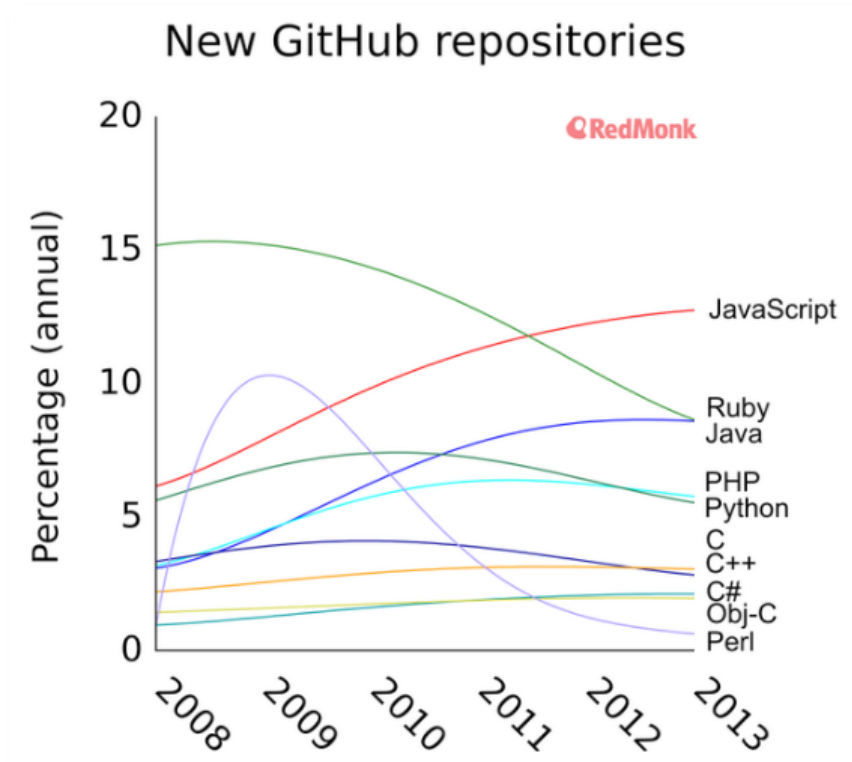
Fonte: <https://insights.stackoverflow.com/survey/2017#technology>

Outro gráfico interessante, esse mais antigo, de 2012, mostra a porcentagem de repositórios, por linguagem, hospedados no GitHub. Ou seja, à época (este número tem crescido cada vez mais), um a cada cinco projetos hospedados no GitHub eram projetos JavaScript.





Outro gráfico, este de 2013, mostra o crescimento do JavaScript como linguagem principal de novos projetos no GitHub, o que é consoante ao gráfico anterior, que mostrava o total de projetos na plataforma. Ou seja, em 2013, um em cada sete novos projetos adicionados ao GitHub era com a linguagem JavaScript.



Muitas, mas muitas empresas e desenvolvedores ao redor do mundo usam JavaScript e Node.js.

Eu realmente acredito que você deveria usar também.

---

# CONFIGURANDO O AMBIENTE

2

“

Talk is cheap. Show me the code .

- *Linus Torvalds*

”

Para que seja possível programar com a plataforma Node.js é necessária a instalação de alguns softwares visando não apenas o funcionamento, mas também a produtividade no aprendizado e desenvolvimento dos programas.

Nós vamos começar simples, para entender os fundamentos, e avançaremos rapidamente para programas e configurações cada vez mais complexas visando que você se torne um profissional nessa tecnologia o mais rápido possível.

Portanto, vamos começar do princípio.

## Instalando o Node.js

Você já tem o Node.js instalado na sua máquina?

A plataforma Node.js é distribuída gratuitamente pelo seu mantenedor, Node.js Foundation, para diversos sistemas operacionais em seu website oficial [Nodejs.org](https://nodejs.org):

**<https://nodejs.org>**

Na tela inicial você deve encontrar dois botões grandes e verdes para fazer download e embora a versão recomendada seja sempre a mais antiga e estável (6 na data que escrevo este livro), eu gostaria que você baixasse a versão mais recente (8, atualmente) para que consiga avançar completamente por este livro, fazendo todos os exercícios e acompanhando todos os códigos sem nenhum percalço.

A instalação não requer nenhuma instrução especial, apenas avance cada uma das etapas e aceite o contrato de uso da plataforma.

O Node.js é composto basicamente por:

- » Um runtime JavaScript (Google V8, o mesmo do Chrome);
- » Uma biblioteca para I/O de baixo nível (libuv);
- » Bibliotecas de desenvolvimento básico (os core modules);
- » Um gerenciador de pacotes via linha de comando (NPM);
- » Um gerenciador de versões via linha de comando (NVM);
- » Utilitário REPL via linha de comando;

Deve ser observado que o Node.js não é um ambiente visual ou uma ferramenta integrada de desenvolvimento, embora mesmo assim seja possível o desenvolvimento de aplicações complexas apenas com o uso do mesmo, sem nenhuma ferramenta externa.

Após a instalação do Node, para verificar se ele está funcionando, abra seu terminal de linha de comando (DOS, Terminal, Shell, bash, etc) e digite o comando abaixo:

Código 1: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 node -v
```

O resultado deve ser a versão do Node.js atualmente instalada na sua máquina, no meu caso, “v8.0.0”. Isso mostra que o Node está instalado na sua máquina e funcionando corretamente.

Inclusive este comando ‘node’ no terminal pode ser usado para invocar o utilitário REPL do Node.js (Read-Eval-Print-Loop) permitindo programação e execução via terminal, linha-a-linha. Apenas para brincar (e futuramente caso queira provar conceitos), digite o comando abaixo no seu terminal:

Código 2: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 node
```

Note que o terminal irá ficar esperando pelo próximo comando, entrando em um modo interativo de execução de código JavaScript em cada linha, a cada Enter que você pressionar.

Apenas para fins ilustrativos, pois veremos JavaScript em mais detalhes nos próximos capítulos, inclua os seguintes códigos no terminal, pressionando Enter após digitar cada um:

Código 3: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 var x=0
2 console.log(x)
```

O que aparecerá após o primeiro comando?  
E após o segundo?

E se você escrever e executar (com Enter) o comando abaixo?

Código 4: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | console.log(x+1)
```

Essa ferramenta REPL pode não parecer muito útil agora, mas conforme você for criando programas mais e mais complexos, fazer provas de conceito rapidamente via terminal de linha de comando vai lhe economizar muito tempo e muitas dores de cabeça.

Avançando nossos testes iniciais (apenas para nos certificarmos de que tudo está funcionando como deveria), vamos criar nosso primeiro programa JavaScript para rodar no Node.js com apenas um arquivo.

Abra o editor de texto mais básico que você tiver no seu computador (Bloco de Notas, vim, nano, etc) e escreva dentro dele o seguinte trecho de código:

Código 5: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | console.log('Olá mundo!')
```

Agora salve este arquivo com o nome de index.js (certifique-se que a extensão do arquivo seja “.js”, não deixe que seu editor coloque “.txt” por padrão) em qualquer lugar do seu computador, mas apenas memorize esse lugar, por favor. :)

Para rodar esse programa JavaScript, abra novamente o terminal de linha de comando e execute o comando abaixo:

Código 6: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | node /documents/index.js
```

Isto irá executar o programa contigo no arquivo `/documents/index.js` usando o runtime do Node. Note que aqui eu salvei meu arquivo `.js` na pasta `documents`, logo no seu caso, esse comando pode variar (no Windows inclusive usa-se `\` ao invés de `/`, por exemplo). Uma dica é quando abrir o terminal, usar o comando `cd` para navegar até a pasta onde seus arquivos JavaScript são salvos. Eu inclusive recomendo que você crie uma pasta `NodeProjects` ou simplesmente `Projects` na sua pasta de usuário para guardar todos os exemplos desse livro de maneira organizada. Assim, sempre que abrir um terminal, use o comando `cd` para ir até a pasta apropriada.

Se o seu terminal já estiver apontando para a pasta onde salvou o seu arquivo `.js`, basta chamar o comando `node` seguido do respectivo nome do arquivo (sem pasta) que vai funcionar também.

Ah, o resultado da execução anterior? Apenas um `'Olá mundo!'` (sem aspas) escrito no seu console, certo?!

Note que tudo que você precisa de ferramentas para começar a programar Node é exatamente isso: o runtime instalado e funcionando, um terminal de linha de comando e um editor de texto simples. Obviamente podemos adicionar mais ferramentas ao nosso arsenal, e é disso que trata a próxima sessão.

## Visual Studio Code

Ao longo deste livro iremos desenvolver uma série de exemplos de softwares escritos em JavaScript usando o editor de código Visual Studio Code, da Microsoft.

Esta não é a única opção disponível, mas é uma opção bem interessante e é a que uso, uma vez que reduz consideravelmente a curva de aprendizado, os erros cometidos durante o aprendizado e possui ferramentas de depuração muito boas, além de suporte a Git e linha de comando integrada. Apesar de ser desenvolvido pela Microsoft, é um projeto gratuito, de código-aberto, multi-plataforma e com extensões para diversas linguagens e plataformas, como Node.js. E diferente da sua contraparte mais “parruda”, o Visual Studio original, ele é bem leve e pequeno.

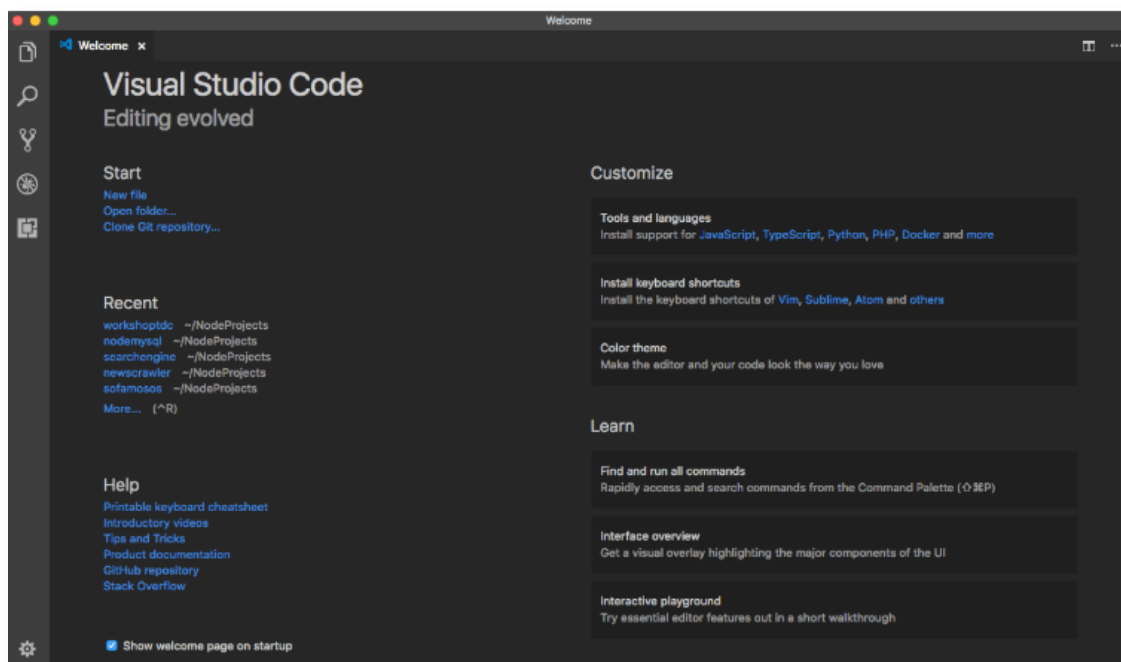
Outras excelentes ferramentas incluem o Visual Studio Community, Atom e o Sublime, sendo que o primeiro eu já utilizava quando era programador .NET e usei durante o início dos meus aprendizados com Node. No entanto, não faz sentido usar uma ferramenta tão pesada para uma plataforma tão leve, mesmo ela sendo gratuita na versão Community. O segundo (Atom) eu nunca usei, mas tive boas recomendações, já o terceiro (Sublime) eu já usei e sinceramente não gosto, especialmente na versão free que fica o tempo todo te pedindo para fazer upgrade pra versão paga. Minha opinião.

Para baixar e instalar o Visual Studio Code, acesse o seguinte link, no site oficial da ferramenta:

<https://code.visualstudio.com/>

Você notará um botão grande e verde para baixar a ferramenta para o seu sistema operacional. Apenas baixe e instale, não há qualquer preocupação adicional.

Após a instalação, mande executar a ferramenta Visual Studio Code e você verá a tela de boas vindas, que deve se parecer com essa abaixo, dependendo da versão mais atual da ferramenta. Chamamos esta tela de Boas Vindas (Welcome Screen).



No menu do topo você deve encontrar a opção File > New File, que abre um arquivo em branco para edição. Apenas adicione o seguinte código nele:

Código 7: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 console.log('Hello World')
```

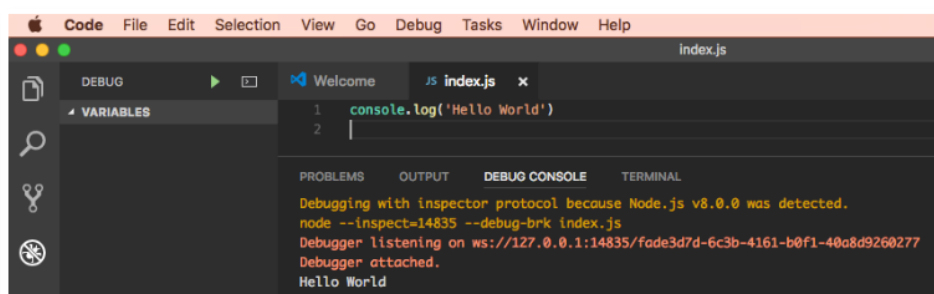
JavaScript é uma linguagem bem direta e sem muitos rodeios, permitindo que com poucas linhas de código façamos coisas incríveis. Ok, um olá mundo não é algo incrível, mas este mesmo exemplo em linguagens de programação como C e Java ocuparia muito mais linhas.

Basicamente o que temos aqui é o uso do objeto 'console', que nos permite ter acesso ao terminal onde estamos executando o Node, e dentro dele estamos invocando a função 'log' que permite escrever no console passando um texto entre aspas (simples ou duplas, tanto faz, mas recomendo simples).

Uma outra característica incomum no JavaScript é que caso tenhamos apenas uma instrução no escopo em questão (neste caso a linha do comando), não precisamos usar ';' (ponto-e-vírgula). Se você colocar, vai notar que funciona também, mas não há necessidade neste caso.

Salve o arquivo escolhendo File > Save ou usando o atalho Ctrl + S. Minha sugestão é que salve dentro de uma pasta NodeProjects/HelloWorld para manter tudo organizado e com o nome de index.js. Geralmente o arquivo inicial de um programa Node.js se chama index, enquanto que a extensão '.js' é obrigatória para arquivos JavaScript.

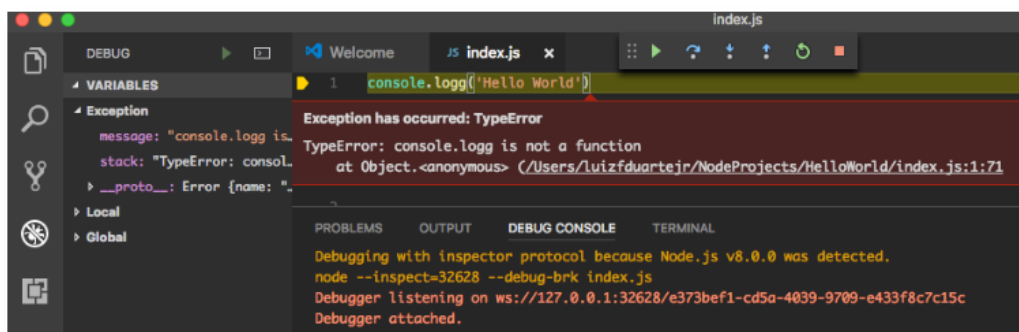
Para executar o programa escolha Debug > Start Debugging (F5). O Visual Studio Code vai lhe perguntar em qual ambiente deve executar esta aplicação (Node.js neste caso) e após a seleção irá executar seu programa com o resultado abaixo como esperado.





Parabéns! Seu programa funciona!

Se houver erros de execução, estes são avisados com uma mensagem vermelha indicando qual erro, em qual arquivo e em qual linha. Se mais de um arquivo for listado, procure o que estiver mais ao topo e que tenha sido programado por você. Os erros estarão em Inglês, idioma obrigatório para programadores, e geralmente possuem solução se você souber procurar no Google em sites como StackOverflow entre outros.



No exemplo acima eu digitei erroneamente a função 'log' com dois 'g's (TypeError = erro de digitação). Conceitos mais aprofundados sobre JavaScript serão vistos posteriormente.

Caso note que sempre que manda executar o projeto (F5) ele pergunta qual é o ambiente, é porque você ainda não configurou um projeto corretamente no Visual Studio Code. Para fazer isso é bem simples, vá no menu File > Open e selecione a pasta do seu projeto, HelloWorld neste caso. O VS Code vai entender que esta pasta é o seu projeto completo.

Agora, para criarmos um arquivo de configuração do VS Code para este projeto, basta ir no menu Debug > Add Configuration, selecionar a opção Node.js e salvar o arquivo, sem necessidade de configurações adicionais. Isso irá criar um arquivo launch.json, de uso exclusivo do VS Code, evitando que ele sempre pergunte qual o environment que você quer usar.

E com isso finalizamos a construção do nosso Olá Mundo em Node.js usando Visual Studio Code, o primeiro programa que qualquer programador deve criar quando está aprendendo uma nova linguagem de programação!

## Google Chrome

O Google Chrome é um navegador de internet, desenvolvido pela companhia Google com visual minimalista e compilado com base em componentes de código licenciado e sua estrutura de desenvolvimento de aplicações (framework).

Em 2 de setembro de 2008 foi lançado a primeira versão ao mercado, sendo uma versão beta e em 11 de dezembro de 2008 foi lançada a primeira versão estável ao público em geral. O navegador atualmente está disponível, em mais de 51 idiomas, para as plataformas Windows, Mac OS X, Android, iOS, Ubuntu, Debian, Fedora e openSUSE.

Atualmente, o Chrome é o navegador mais usado no mundo, com 49,18% dos usuários de Desktop, contra 22,62% do Internet Explorer e 19,25% do Mozilla Firefox, segundo a StatCounter. Além de desenvolver o Google Chrome, o Google ainda patrocina o Mozilla Firefox, um navegador desenvolvido pela Fundação Mozilla.

Durante muitos exemplos deste livro será necessária a utilização de um navegador de Internet. Todos os exemplos foram criados e testados usando o navegador Google Chrome, na versão mais recente disponível à época que era a versão 59. Caso não possuam o Google Chrome na sua máquina, baixe a versão mais recente no site oficial antes de avançar no livro: <https://www.google.com.br/chrome/browser/desktop/index.html>

Além de um excelente navegador, o Google Chrome ainda possui uma série de ferramentas para desenvolvedor que são muito úteis como um inspetor de código HTML da página, um depurador de JavaScript online, métricas de performance da página e muito mais.

## MongoDB

MongoDB é um banco da categoria dos não-relacionais ou NoSQL, por não usar o modelo tradicional de tabelas com colunas que se relacionam entre si. Em MongoDB trabalhamos com documentos BSON (JSON binário) em um modelo de objetos quase idêntico ao do JavaScript.

Falaremos melhor de MongoDB quando chegar a hora, pois ele será o banco de dados que utilizaremos em nossas lições que exijam persistência de informações. Por ora, apenas baixe e extraia o conteúdo do pacote compactado de MongoDB para o seu sistema operacional, sendo a pasta de arquivos de programas ou similar a mais recomendada. Você encontra a distribuição gratuita de MongoDB para o seu sistema operacional no site oficial:

<http://mongodb.org>

Apenas extraia os arquivos, não há necessidade de qualquer configuração e entraremos nos detalhes quando chegar a hora certa.

## Postman

O Postman é uma suíte de ferramentas que auxiliam a vida do desenvolvedor de APIs. Usaremos esta fantástica e gratuita ferramenta mais tarde, quando estivermos desenvolvendo nossas APIs com Node.js. Ela é usada, na época de escrita desse livro, por mais de 3 milhões de desenvolvedores ao redor do mundo.

<https://www.getpostman.com/>

Baixe e instale a versão correta para seu sistema operacional e não se preocupe em entendê-la agora, ensinarei você a usá-lo mais tarde, quando chegar a hora.

---

# EXPRESSJS

3

“

The most important property of a program  
is whether it accomplishes the intention of  
its user.

- C.A.R. Hoare

”

Considerando o que foi falado sobre Node.js no capítulo inicial, existem diversos tipos de aplicações que podemos criar com esta tecnologia. Uma categoria particularmente em alta atualmente são o das aplicações web ou sistemas web. No entanto, criar uma aplicação web do zero com Node.js não é uma tarefa muito simples, motivo pelo qual geralmente utiliza-se algum framework em conjunto com ele.

O ExpressJS é o web framework mais famoso da atualidade para Node.js. Com ele você consegue criar aplicações e WebAPIs muito rápida e facilmente.

Uma vez com o NPM instalado (capítulo 2), vamos instalar um módulo que nos será muito útil em diversos momentos deste livro. Rode o seguinte comando com permissão de administrador no terminal ('sudo' em sistemas Unix-like):

Código 8: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 C:\node> npm install -g express-generator
```

O express-generator é um módulo que permite rapidamente criar a estrutura básica de um projeto Express via linha de comando, da seguinte maneira (aqui considero que você salva seus projetos na pasta C:\node):

Código 9: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 C:\node> express -e --git workshop
```

O “-e” é para usar a view-engine (motor de renderização) EJS, ao invés do tradicional Jade/Pug (falaremos dele mais pra frente). Já o “--git” deixa seu projeto preparado para versionamento com Git. Aperte Enter e o projeto será criado (talvez ele peça uma confirmação, apenas digite ‘y’ e confirme).

Depois entre na pasta e mande instalar as dependências com npm install:

Código 10: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

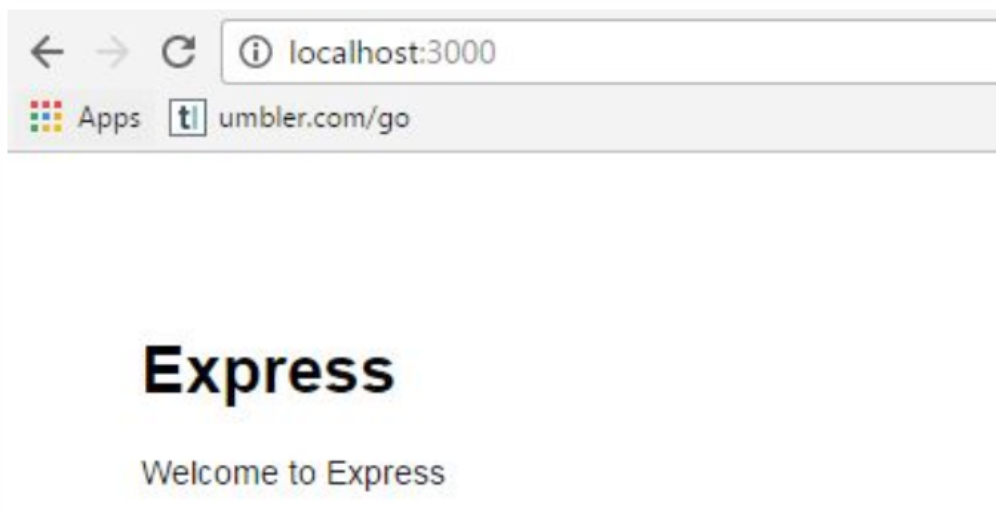
```
1 | cd workshop
2 | npm install
```

Ainda no terminal de linha de comando e, dentro da pasta do projeto, digite:

Código 11: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | > npm start
```

Isso vai fazer com que a aplicação default inicie sua execução em localhost:3000, que você pode acessar pelo seu navegador.



Vamos entender o framework Express agora.

Entre na pasta bin e depois abra o arquivo www que fica dentro dela. Esse é um arquivo sem extensão que pode ser aberto com qualquer editor de texto.

Dentro do `www` você deve ver o código JS que inicializa o servidor web do Express e que é chamado quando digitamos o comando `'npm start'` no terminal. Ignorando os comentários e blocos de funções, temos:

Código 12: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 var app = require('../app');
2 var debug = require('debug')('workshop:server');
3 var http = require('http');
4
5 var port = normalizePort(process.env.PORT ||
6   '3000');
7 app.set('port', port);
8
9 var server = http.createServer(app);
10
11 server.listen(port);
12 server.on('error', onError);
13 server.on('listening', onListening);
```

Na primeira linha é carregado um módulo local chamado `app`, que estudaremos na sequência. Depois, um módulo de debug usado para imprimir informações úteis no terminal durante a execução do servidor. Na última linha do primeiro bloco carregamos o módulo `http`, elementar para a construção do nosso webserver.

No bloco seguinte, apenas definimos a porta que vai ser utilizada para escutar requisições. Essa porta pode ser definida em uma variável de ambiente (`process.env.PORT`) ou caso essa variável seja omitida, será usada a porta 3000.

O servidor `http` é criado usando a função apropriada (`createServer`) passando o `app` por parâmetro e depois definindo que o server escute (`listen`) a porta pré-definida. Os dois últimos comandos definem

manipuladores para os eventos de error e listening, que apenas ajudam na depuração dos comportamentos do servidor.

Note que não temos muita coisa aqui e que com pouquíssimas linhas é possível criar um webserver em Node.js. Esse arquivo `www` é a estrutura mínima para iniciar uma aplicação web com Node.js e toda a complexidade da aplicação em si cabe ao módulo `app.js` gerenciar. Ao ser carregado com o comando `require`, toda a configuração da aplicação é executada, conforme veremos a seguir.

Abra agora o arquivo `app.js`, que fica dentro do diretório da sua aplicação Node.js (workshop no meu caso). Este arquivo é o coração da sua aplicação, embora não exista nada muito surpreendente dentro. Você deve ver algo parecido com isso logo no início:

Código 13: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 var express = require('express');
2 var path = require('path');
3 var favicon = require('serve-favicon');
4 var logger = require('morgan');
5 var cookieParser = require('cookie-parser');
6 var bodyParser = require('body-parser');
7
8 var index = require('./routes/index');
9 var users = require('./routes/users');
```

Isto define um monte de variáveis JavaScript e referencia elas a alguns pacotes, dependências, funcionalidades do Node e rotas. Rotas direcionam o tráfego e contém também alguma lógica de programação (embora você consiga, se quiser, usar padrões mais “puros” como MVC se desejar). Quando criamos o projeto Express, ele criou estes códigos JS pra gente e vamos ignorar a rota ‘users’ por enquanto e nos focar no index, controlado pelo arquivo `c:\node\workshop\routes\index.js` (falaremos dele mais tarde).



Na sequência você deve ver:

Código 14: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | var app = express();
```

Este é bem importante. Ele instancia o Express e associa nossa variável `app` à ele. A próxima seção usa esta variável para configurar coisas do Express.

Código 15: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1  // view engine setup
2  app.engine('html', require('ejs').renderFile);
3  app.set('views', __dirname + '/views');
4  app.set('view engine', 'ejs');
5
6  // uncomment after placing your favicon in /
7  public
8  //app.use(favicon(path.join(__dirname,
9  'public', 'favicon.ico')));
10 app.use(logger('dev'));
11 app.use(bodyParser.json());
12 app.use(bodyParser.urlencoded({ extended: false
13 }));
14 app.use(cookieParser());
15 app.use(express.static(path.join(__dirname,
16 'public')));
17
18 app.use('/', index);
19 app.use('/users', users);
```

Isto diz ao app onde ele encontra suas views, qual engine usar para renderizar as views (EJS) e chama alguns métodos para fazer com que as coisas funcionem. Note também que esta linha final diz ao Express para acessar os objetos estáticos a partir de uma pasta /public/, mas no navegador elas aparecerão como se estivessem na raiz do projeto. Por exemplo, a pasta images fica em c:\node\workshop\public\images mas é acessada em <http://localhost:3000/images>

Os próximos três blocos são manipuladores de erros para desenvolvimento e produção (além dos 404). Não vamos nos preocupar com eles agora, mas resumidamente você tem mais detalhes dos erros quando está operando em desenvolvimento.

Código 16: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | module.exports = app;
```

Uma parte importantíssima do Node é que basicamente todos os arquivos .js são módulos e basicamente todos os módulos exportam um objeto que pode ser facilmente chamado em qualquer lugar no código. Nosso objeto app é exportado no módulo acima para que possa ser usado no arquivo www, como vimos anteriormente.

Uma vez que entendemos como o www e o app.js funcionam, é hora de partirmos pra diversão!

O Express na verdade é um middleware web. Uma camada que fica entre o HTTP server criado usando o módulo http do Node.js e a sua aplicação web, interceptando cada uma das requisições, aplicando regras, carregando telas, servindo arquivos estáticos, etc. Resumindo: simplificando e muito a nossa vida como desenvolvedor web.

Existem duas partes básicas e essenciais que temos de entender do Express para que consigamos programar minimamente usando ele: routes e views ou “rotas e visões”. Falaremos delas agora.

## Routes e Views

Quando estudamos o app.js demos uma rápida olhada em como o Express lida com routes e views, mas você não deve se lembrar disso.

Routes são regras para manipulação de requisições HTTP. Você diz que, por exemplo, quando chegar uma requisição no caminho '/teste', o fluxo dessa requisição deve passar pela função 'X'. No app.js, registramos duas rotas nas linhas abaixo:

Código 17: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 // códigos...
2 var index = require('./routes/index');
3 var users = require('./routes/users');
4
5 // mais códigos...
6
7 app.use('/', index);
8 app.use('/users', users);
```

Carregamos primeiro os módulos que vão lidar com as rotas da nossa aplicação. Cada módulo é um arquivo .js na pasta especificada (routes). Depois, dizemos ao app que para requisições no caminho raiz da aplicação ('/'), o módulo index.js irá tratar. Já para as requisições no caminho '/users', o módulo users.js irá lidar. Ou seja, o app.js apenas repassa as requisições conforme regras básicas, como um middleware.

Abra o arquivo routes/index.js para entendermos o que acontece após redirecionarmos requisições na raiz da aplicação para ele.

Código 18: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```

1 var express = require('express');
2 var router = express.Router();
3
4 router.get('/', function(req, res, next) {
5   res.render('index', { title: 'Express' });
6 });
7
8 module.exports = router;

```

Primeiro carregamos o módulo `express` e com ele o objeto `router`, que serve para manipular as requisições recebidas por esse módulo. O bloco central de código é o que mais nos interessa. Nele especificamos que quando o `router` receber uma requisição GET na raiz da requisição, que essa requisição será tratada pela função passada como segundo parâmetro. E é aqui que a magia acontece.

**Nota:** você pode usar `router.get`, `router.post`, `router.delete`, etc. O objeto `router` consegue rotear qualquer requisição HTTP que você precisar. Veremos isso na prática mais pra frente.

**Atenção:** o trecho `"router.get('/')"` (no `routes/index.js`) não quer dizer a mesma coisa que `"app.use('/')"` (no `app.js`). Na verdade eles são cumulativos. Por exemplo, eu posso dizer que toda requisição na raiz (`'/'`) vai pro `index.js` e dentro dele eu definir que `"router.get('/teste')"` faz uma coisa e `"router.get('/new')"` faz outra, pois ambos começam na raiz (`'/'`). Agora se eu disser `"app.use('/teste', teste)"` (no `app.js`), essas requisições irão ser processadas pelo módulo `routes/teste.js` que pode ter um `"router.get('/')"` (que lida com requisições em `'/teste/'`) e tantos outros manipuladores que eu quiser, como `"router.get('/novo')"` (que lida com requisições `'/teste/novo'`). Mais pra frente veremos como ter parâmetros no caminho da requisição, variáveis, etc.

A função anônima passada como segundo parâmetro do `router.get` será disparada toda vez que chegar um GET na raiz da aplicação. A esse comportamento chamamos de `callback`. Para cada requisição que chegar (`call`), nós disparamos a `function` (`callback` ou ‘retorno da chamada’). Esse modelo de `callbacks` é o coração do comportamento assíncrono baseado em eventos do Node.js e falaremos bastante dele ao longo desse livro.

Essa função ‘mágica’ possui três parâmetros: `req`, `res` e `next`.

`req`: contém informações da requisição HTTP que disparou esta `function`. A partir dele podemos saber informações do cabeçalho (`header`) e do corpo (`body`) da requisição livremente, o que nos será muito útil.

`res`: é o objeto para enviar uma resposta ao requisitante (`response`). Essa resposta geralmente é uma página HTML, mas pode ser um arquivo, um objeto JSON, um erro HTTP ou o que você quiser devolver ao requisitante.

`next`: é um objeto que permite repassar a requisição para outra função manipular. É uma técnica mais avançada que exploraremos quando surgir a necessidade.

Vamos focar nos parâmetros `req` e `res` aqui. O ‘`req`’ é a requisição em si, já o ‘`res`’ é a resposta.

Dentro da função de `callback` do `router.get`, temos o seguinte código (que já foi mostrado antes):

Código 19: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

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

Aqui dizemos que deve ser `renderizado` na resposta (`res.render`) a `view` ‘`index`’ com o `model` entre `chaves` (`{}`). Se você já estudou o padrão MVC antes, deve estar se sentindo em casa e entendendo que o `router` é o `controller` que liga o `model` com a `view`.

As views são referenciadas no `res.render` sem a extensão, e todas encontram-se na pasta `views`. Falaremos delas mais tarde. Já o `model` é um objeto JSON com informações que você queira enviar para a view usar. Nesse exemplo, estamos enviando um título (`title`) para view usar.

Experimente mudar a string `'Welcome to Express'` para outra coisa que você quiser, salve o arquivo `index.js`, derrube sua aplicação no terminal (`Ctrl+C`), execute-a com `'npm start'` e acesse novamente `localhost:3000` para ver o texto alterado conforme sua vontade.

Para entender essa 'bruxaria' toda, temos de entender como as views funcionam no Express.

Lembra lá no início da criação da nossa aplicação Express usando o `express-generator` que eu disse para usar a opção `'-e'` no terminal?

*`"express -e --git workshop"`*

Pois é, ela influencia como que nossas views serão interpretadas e renderizadas nos navegadores. Neste caso, usando `-e`, nossa aplicação será configurada com a view-engine EJS (Embedded JavaScript) que permite misturar HTML com JavaScript server-side para criar os layouts.

**Nota:** a view-engine padrão do Express (sem a opção `-e`) é a Pug (antiga Jade). Ela não é ruim, muito pelo contrário, mas como não usa a linguagem HTML padrão optei por usar a EJS. Existem outras alternativas no mercado, como HandleBars (`hbs`), mas nesse livro usarei EJS do início ao fim para não confundir ninguém.

Voltando ao `app.js`, o bloco abaixo configura como que o nosso 'view engine' irá funcionar:

Código 20: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```

1 // view engine setup
2 app.engine('html', require('ejs').renderFile);
3 app.set('views', __dirname + '/views');
4 app.set('view engine', 'ejs');

```

Aqui dissemos que vamos renderizar HTML (a linguagem padrão para criação de páginas web) usando o objeto `renderFile` do módulo `'ejs'`. Depois, dizemos que todas as views ficarão na pasta `'views'` da raiz do projeto e por fim dizemos que o motor de renderização (view engine) será o `'ejs'`.

Esta é toda a configuração necessária para que arquivos HTML sejam renderizados usando EJS no Express. Cada view conterá a sua própria lógica de renderização e será armazenada na pasta `views`, em arquivos com a extensão `.ejs`.

Abra o arquivo `/views/index.ejs` para entender melhor como essa lógica funciona:

Código 21: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title><%= title %></title>
5     <link rel='stylesheet' href='/stylesheets/
6 style.css' />
7   </head>
8   <body>
9     <h1><%= title %></h1>
10    <p>Welcome to <%= title %></p>
11  </body>
12 </html>

```

Neste livro não aprendemos HTML, coisa que você facilmente pode aprender em outros livros ou mesmo na Internet em sites como W3Schools, w3c.org e MDN. Caso não conheça HTML, apenas entenda que palavras entre `<>` são chamadas de ‘tags’ e que cada uma representa um elemento de layout. Já as tags `<% %>` são server-tags, tags especiais que são processadas pelo Node.js e que podem conter códigos JavaScript dentro. Esses códigos serão acionados quando o navegador estiver renderizando este arquivo.

No nosso caso, apenas estamos usando `<%= title %>` que é o mesmo que dizer ao navegador ‘renderiza o conteúdo da variável title’. Hmmm, onde que vimos essa variável title antes?

Dentro do routes/index.js!!!

Código 22: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

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

O title é a informação passada junto ao parâmetro de model do res.render. Exploraremos mais esse conceito de model futuramente, mas por ora basta entender que tudo que você passar como model no res.render pode ser usado pela view que está sendo renderizada.

Para finalizar nosso estudo de Express e para ver se você entendeu direitinho como as routes e views funcionam, lhe proponho um desafio: crie uma nova view que deve ser exibida quando o usuário acessar ‘/new’ no navegador. Como ainda não vimos HTML, use o mesmo código HTML da view index, mas mude a mensagem conforme sua vontade.

Se não consegue pensar no passo-a-passo necessário sozinho, use a lista de tarefas abaixo:

crie o arquivo da nova rota em ‘routes’;

para programar a rota, use como exemplo o routes/index.js;

- » crie o arquivo da nova view em ‘views’;
- » para criar o layout da view, use como exemplo a view/index.ejs;
- » no app.js, carregue a sua rota em uma variável local e diga para o app usar a sua variável quando chegar requisições em ‘/new’;



Se nem mesmo com esse passo-a-passo você conseguir fazer sozinho, abaixo você encontra o código da nova rota em routes/new.js (ele espera que você já tenha um arquivo views/new.ejs idêntico ao views/index.ejs):

Código 23: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 var express = require('express');
2 var router = express.Router();
3
4 router.get('/', function(req, res, next) {
5   res.render('new', { title: 'Novo Cadastro' });
6 });
7
8 module.exports = router;
```

E abaixo o código que deve ser colocado no app.js para registrar sua nova rota:

Código 24: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 var new = require('./routes/new');
2 app.use('/new', new);
```

Outra alternativa, que talvez você tenha ‘sacado’ é adicionar uma nova rota dentro do routes/index.js tratando GET /new o que te poupa de ter de adicionar código novo no app.js. A explicação do porque isso funciona encontra-se no último bloco Atenção, dá uma lida lá se você ainda não leu.

**Nota:** você deve ter reparado que estes códigos gerados pelo `express-generator` não usam os padrões de variáveis e `;` descritos no capítulo anterior sobre JavaScript. Aqui você viu `'var'` sendo usada o tempo todo e sempre as linhas foram terminadas com `;`. Infelizmente o módulo `express-generator` não segue as boas práticas mais modernas de código JavaScript e você vai ter de aprender a lidar com essas diferenças, pois muitos exemplos na Internet não seguem esses padrões.

## Event Loop

Existe um elemento chave que não temos como ignorar quando o assunto é Node.js, independente se você usar o Express ou não. Estou falando do Event Loop.

Grande parte das características e principalmente das vantagens do Node.js se devem ao funcionamento do seu loop single-thread principal e como ele se relaciona com as demais partes do Node, como a biblioteca C++ libuv.

Assim, a ideia deste tópico é ajudar você a entender como o Event Loop do Node.js funciona, o que deve lhe ajudar a entender como tirar o máximo de proveito dele.

## O Problema

Antes de entrar no Event Loop em si, vamos primeiro entender porque o Node.js possui um e qual o problema que ele propõe resolver.

A maioria dos backends por trás dos websites mais famosos não fazem computações complicadas. Nossos programas passam a maior parte do tempo lendo ou escrevendo no disco, ou melhor, esperando a sua vez de ler e escrever, uma vez que é um recurso lento e concorrido. Quando não estamos nesse processo de ir ao disco, estamos enviando ou recebendo bytes da rede, que é outro processo igualmente demorado (que nem fizemos com as requisições do Express). Ambos processos podemos resumir como operações de I/O (Input & Output) ou E/S (Entrada & Saída).

Processar dados, ou seja executar algoritmos, é estupidamente mais rápido do que qualquer operação de IO que você possa querer fazer. Mesmo se tivermos um SSD em nossa máquina com velocidades de leitura de 200-730 MB/s fará com que a leitura de 1KB de dados leve 1.4 microssegundos. Parece rápido? Saiba que nesse tempo uma CPU de 2GHz consegue executar 28.000 instruções.

Isso mesmo. Ler um arquivo de 1KB demora tanto tempo quanto executar 28.000 instruções no processador. É muito lento.

Quando falamos de IO de rede é ainda pior. Faça um teste, abra o CMD e execute um ping no site do google.com, um dos mais rápidos do planeta:

```
1 $ ping google.com
2 64 bytes from 172.217.16.174: icmp_seq=0 ttl=52
3 time=33.017 ms
4 64 bytes from 172.217.16.174: icmp_seq=1 ttl=52
5 time=83.376 ms
6 64 bytes from 172.217.16.174: icmp_seq=2 ttl=52
7 time=26.552 ms
```

A latência média nesse teste é de 44 milisegundos. Ou seja, enviar um ping para o Google demora o mesmo tempo que uma CPU necessita para executar 88 milhões de operações.

Ou seja, quando estamos fazendo uma chamada a um recurso na Internet, poderíamos estar fazendo cerca de 88 milhões de coisas diferentes na CPU.

É muita diferença!

## A solução

A maioria dos sistemas operacionais lhe fornece mecanismos de programação assíncrona, o que permite que você mande executar tarefas concorrentes que não ficam esperando uma pela outra, desde que uma não precise do resultado da outra, é claro.

Esse tipo de comportamento pode ser alcançado de diversas maneiras. Atualmente a forma mais comum de fazer isso é através do uso de threads o que geralmente torna nosso código muito mais complexo. Por exemplo, ler um arquivo em Java é uma operação bloqueante, ou seja, seu programa não pode fazer mais exceto esperar a comunicação com a rede ou disco terminar. O que você pode fazer é iniciar uma thread diferente para fazer essa leitura e mandar ela avisar sua thread principal quando a leitura terminar.

Novas formas e programação assíncrona tem surgido com o uso de interfaces async como em Java e C#, mas isso ainda está evoluindo. Por ora isso é entediante, complicado, mas funciona. Mas e o Node? A característica de single-thread dele obviamente deveria representar um problema uma vez que ele só consegue executar uma tarefa de um usuário por vez, certo? Quase.

O Node usa um princípio semelhante ao da função `setTimeout(func, x)` do Javascript, onde a função passada como primeiro parâmetro é delegada para outra thread executar após x milissegundos, liberando a thread principal para continuar seu fluxo de execução. Mesmo que você defina x como 0, o que pode parecer algo inútil, isso é extremamente útil pois força a função a ser realizada em outra thread imediatamente.

No Node.js, sempre que você chama uma função síncrona (i.e. “normal”) ela vai para uma “call stack” ou pilha de chamadas de funções com o seu endereço em memória, parâmetros e variáveis locais. Se a partir dessa função você chamar outra, esta nova função é empilhada em cima da anterior (não literalmente, mas a ideia é essa). Quando essa nova função termina, ela é removida da call stack e voltamos o fluxo da função anterior. Caso a nova função tenha retornado um valor, o mesmo é adicionado à função anterior na call stack.

Mas o que acontece quando chamamos algo como `setTimeout`, `http.get`, `process.nextTick`, ou `fs.readFile` (estes últimos que veremos mais adiante)? Estes não são recursos nativos do V8, mas estão disponíveis no Chrome WebApi e na C++ API no caso do Node.js.

Vamos dar uma olhada em uma aplicação Node.js comum - um servidor escutando em `localhost:3000`. Após receber a requisição, o servidor

vai ler um arquivo para obter um trecho de texto e imprimir algumas mensagens no console e depois retorna a resposta HTTP.

Código 25: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 //coloque todo o conteúdo abaixo dentro de um
2 arquivo index.js
3 //rode o comando "npm init" na mesma pasta do
4 index.js e apenas aperte Enter para tudo
5 //rode os comandos "npm install -S express fs"
6 para instalar as dependências
7 //use o comando "node index" na pasta do index.
8 js para iniciar esse programa
9 const express = require('express')
10 const fs = require('fs') //fs é o módulo file-
11 system, para ler arquivos
12 const app = express()
13
14 app.get('/', processRequest)
15
16 function processRequest (request, response) {
17   readText(request, response)
18   console.log('requisição terminou')
19 }
20
21 function readText (request, response) {
22   //salve um arquivo teste.txt junto a esse
23   arquivo com qualquer coisa dentro
24   fs.readFile('teste.txt', function(err, data) {
25     if (err)
```

```

26         console.log('erro na leitura')
27         return response.status(500).send('Erro
28 ao ler o arquivo.')
29     }
30     response.write(data)
31     response.end();
32     console.log('leu arquivo')
33 });
34
35 console.log('Lendo o arquivo, aguarde.')
36 }
37
38 app.listen(3000)

```

Não esqueça de seguir as instruções dos comentários ao longo do código para fazê-lo funcionar corretamente e depois acesse localhost:3000 no seu navegador.

O que será impresso quando uma requisição é enviada para localhost:3000?

Se você já mexeu um pouco com Node antes, não ficará surpreso com o resultado, pois mesmo que console.log('Lendo o arquivo, aguarde.') tenha sido chamado depois de console.log('leu arquivo') no código, o resultado da requisição será como abaixo (caso não tenha criado o arquivo txt):

---

**Lendo o arquivo, aguarde.  
requisição terminou  
erro na leitura**

---

Ou então, caso tenha criado o arquivo teste.txt:

---

**Lendo o arquivo, aguarde.  
requisição terminou  
leu arquivo**

---

O que aconteceu?

Mesmo o V8 sendo single-thread, a API C++ do Node não é. Isso significa que toda vez que o Node for solicitado para fazer uma operação bloqueante, Node irá chamar a libuv que executará concorrentemente com o Javascript em background. Uma vez que esta thread concorrente terminar ou jogar um erro, o callback fornecido será chamado com os parâmetros necessários.

A libuv é uma biblioteca C++ open-source usada pelo Node em conjunto com o V8 para gerenciar o pool de threads que executa as operações concorrentes ao Event Loop single-thread do Node. Ela cuida da criação e destruição de threads, semáforos e outras “magias” que são necessárias para que as tarefas assíncronas funcionem corretamente. Essa biblioteca foi originalmente escrita para o Node, mas atualmente outros projetos a usam também.

## **Task/Event/Message Queue**

Javascript é uma linguagem single-thread orientada a eventos. Isto significa que você pode anexar gatilhos ou listeners aos eventos e quando o respectivo evento acontece, o listener executa o callback que foi fornecido.

Toda vez que você chama `setTimeout`, `http.get` ou `fs.readFile`, Node.js envia estas operações para a libuv executá-las em uma thread separada do pool, permitindo que o V8 continue executando o código na thread principal. Quando a tarefa termina e a libuv avisa o Node disso, o Node dispara o callback da referida operação.

No entanto, considerando que só temos uma thread principal e uma call stack principal, onde que os callbacks ficam guardados para serem executados? Na Event/Task/Message Queue, ou o nome que você preferir. O nome ‘event loop’ se dá à esse ciclo de eventos que acontece

infinitamente enquanto há callbacks e eventos a serem processados na aplicação.

Em nosso exemplo anterior, de leitura de arquivo, nosso event loop ficou assim:

1. Express registrou um handler para o evento 'request' que será chamado quando uma requisição chegar em '/'
2. ele começa a escutar na porta 3000
3. a stack está vazia, esperando pelo evento 'request' disparar
4. quando a requisição chega, o evento dispara e o Express chama o handler configurado: `processRequest`
5. `processRequest` é empilhado na call stack
6. `readText` é chamado dentro da função anterior e é também empilhado na call stack
7. `fs.readFile` é chamado com o parâmetro 'teste.txt' e definimos o handler/callback para o evento de término (end) da requisição.
8. a leitura do arquivo 'teste.txt' no disco é enviada para uma thread em background e a execução continua
9. 'Lendo o arquivo, aguarde.' é impresso no console e `readText` retorna
10. `olaMundo()` é chamada, 'olá mundo' é impresso no console
11. `processRequest` retorna, é retirado da call stack, deixando-a vazia
12. ficamos esperando pela chamada de `fs.readFile` nos responder
13. uma vez que a resposta chegue, o callback é disparado (em resposta ao evento 'end' da leitura de arquivo)
14. o callback anônimo que passamos é chamado, é colocado na call stack com todas as variáveis locais, o que significa que ele pode ver e modificar os valores de `express`, `fs`, `app`, `request`, `response` e todas as funções que definimos
15. `response.write()` é chamado, mas isso também é executado em uma thread do pool para a stream de respostas não fique bloqueada e o handler anônimo é retirado da pilha.

E é assim que tudo funciona!

Vale salientar que por padrão o pool de threads da libuv inicia com 4 threads concorrentes e que isso pode ser configurado conforme a sua necessidade.



---

# MONGODB

4

“

Truth can only be found in one place: the code.

- *Robert C. Martin*

”

Há uns 10 anos, mais ou menos, eu estava fazendo as cadeiras de Banco de Dados I e Banco de Dados II na faculdade de Ciência da Computação. Eu via como modelar um banco de dados relacional, como criar consultas e executar comandos SQL, além de álgebra relacional e um pouco de administração de banco de dados Oracle.

Isso tudo me permitiu passar a construir sistemas de verdade, com persistência de dados. A base em Oracle me permitiu aprender o simplíssimo MS Access rapidamente e, mais tarde, migrar facilmente para o concorrente, SQL Server. Posteriormente cheguei ainda a trabalhar com MySQL, SQL Server Compact, Firebird (apenas estudos) e SQLite (para apps Android).

Todos relacionais. Todos usando uma sintaxe SQL quase idêntica. Isso foi o bastante para mim durante alguns anos. Mas essa época já passou faz tempo. Hoje em dia, cada vez mais os projetos dos quais participo têm exigido de mim conhecimentos cada vez mais políglotas de persistência de dados, ou seja, diferentes linguagens e mecanismos para lidar com os dados das aplicações, dentre eles, o MongoDB.

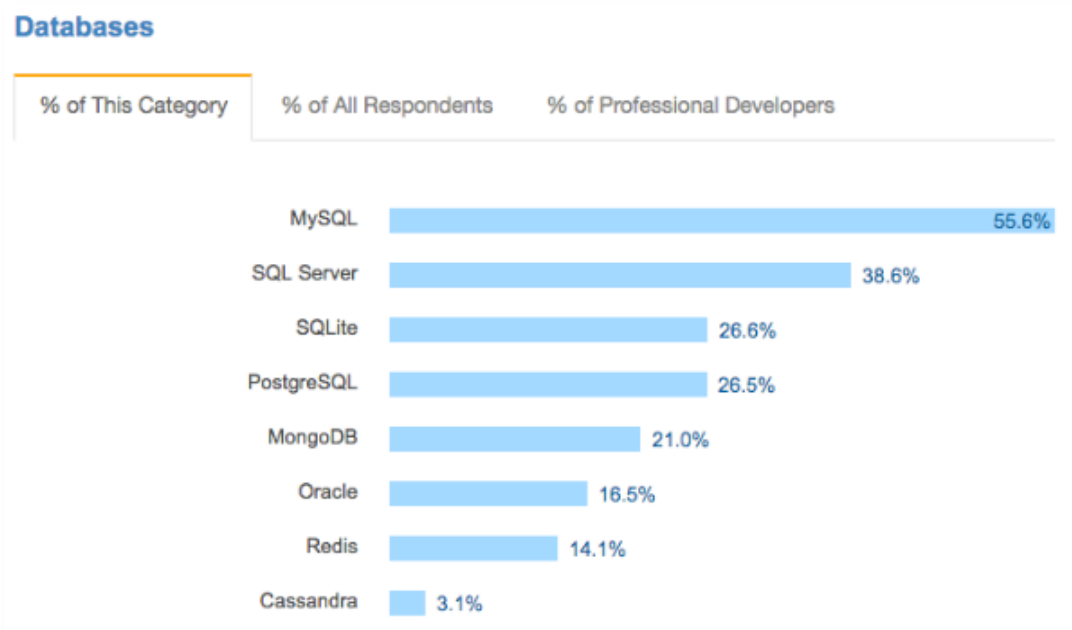
## Introdução ao MongoDB

MongoDB é um banco de dados de código aberto, gratuito, de alta performance, sem esquemas e orientado a documentos lançado em fevereiro de 2009 pela empresa 10gen. Foi escrito na linguagem de programação C++ (o que o torna portátil para diferentes sistemas operacionais) e seu desenvolvimento durou quase 2 anos, tendo iniciado em 2007.

Por ser orientado a documentos JSON (armazenados em modo binário, nomeado de BSON), muitas aplicações podem modelar informações de modo muito mais natural, pois os dados podem ser aninhados em hierarquias complexas e continuar a ser indexáveis e fáceis de buscar, igual ao que já é feito em JavaScript.

Existem dezenas de bancos NoSQL no mercado, não porque cada um inventa o seu, como nos fabricantes tradicionais de banco SQL (não existem diferenças tão gritantes assim entre um MariaDB e um MySQL atuais que justifique a existência dos dois, por exemplo). É apenas uma questão ideológica, para dizer o mínimo, e MongoDB é um deles.

Existem dezenas de bancos NOSQL porque existem dezenas de problemas de persistência de dados que o SQL tradicional não resolve. Bancos não-relacionais document-based (que armazenam seus dados em documentos) são os mais comuns e mais proeminentes de todos, sendo o seu maior expoente o banco MongoDB como o gráfico abaixo da pesquisa mais recente de **bancos de dados utilizados pela audiência do StackOverflow em 2017** mostra.



Dentre todos os bancos não relacionais o MongoDB é o mais utilizado com  $\frac{1}{5}$  de todos os respondentes alegarem utilizar ele em seus projetos, o que é mais do que até mesmo o Oracle, um banco muito mais tradicional.

Basicamente neste tipo de banco (document-based ou document-oriented) temos coleções de documentos, nas quais cada documento é autossuficiente, contém todos os dados que possa precisar, ao invés do conceito de não repetição + chaves estrangeiras do modelo relacional.

A ideia é que você não tenha de fazer JOINS pois eles prejudicam muito a performance em suas queries (são um mal necessário no modelo relacional, infelizmente). Você modela a sua base de forma que a cada query você vai uma vez no banco e com apenas uma chave primária pega tudo que precisa.

Obviamente isto tem um custo: armazenamento em disco. Não é raro bancos MongoDB consumirem muitas vezes mais disco do que suas contrapartes relacionais.

## Quando devo usar MongoDB?

MongoDB foi criada com Big Data em mente. Ele suporta tanto escalonamento horizontal quanto vertical usando replica sets (instâncias espelhadas) e sharding (dados distribuídos), tornando-o uma opção muito interessante para grandes volumes de dados, especialmente os desestruturados.

Dados desestruturados são um problema para a imensa maioria dos bancos de dados relacionais, mas não tanto para o MongoDB. Quando o seu schema é variável, é livre, usar MongoDB vem muito bem a calhar. Os documentos BSON (JSON binário) do Mongo são schemaless e aceitam quase qualquer coisa que você quiser armazenar, sendo um mecanismo de persistência perfeito para uso com tecnologias que trabalham com JSON nativamente, como JavaScript (e consequentemente Node.js).

```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```



Cenários altamente recomendados e utilizados atualmente são em catálogos de produtos de e-commerces. Telas de detalhes de produto em ecommerces são extremamente complicadas devido à diversidade de informações aliada às milhares de variações de características entre os produtos que acabam resultando em dezenas de tabelas se aplicado sobre o modelo relacional. Em MongoDB essa problemática é tratada de uma maneira muito mais simples, que explicarei mais adiante.

Além do formato de documentos utilizado pelo MongoDB ser perfeitamente intercambiável com o JSON serializado do JS, MongoDB opera basicamente de maneira assíncrona em suas operações, assim como o próprio Node.js, o que nos permite ter uma persistência extremamente veloz aliado a uma plataforma de programação igualmente rápida.

Embora o uso de Node.js com bancos de dados relacionais não seja incomum, é com os bancos não-relacionais como MongoDB e Redis que ele mostra todo o seu poder de tecnologia para aplicações real-time e volumes absurdos de requisições na casa de 500 mil/s, com as configurações de servidor adequadas.

Além disso, do ponto de vista do desenvolvedor, usar MongoDB permite criar uma stack completa apenas usando JS uma vez que temos JS no lado do cliente, do servidor (com Node) e do banco de dados (com Mongo), pois todas as queries são criadas usando JS também, como você verá mais à frente.

## Quando não devo usar MongoDB?

Nem tudo são flores e o MongoDB não é uma “bala de prata”, ele não resolve todos os tipos de problemas de persistência existentes.

Você não deve utilizar MongoDB quando relacionamentos entre diversas entidades são importantes para o seu sistema. Se for ter de usar muitas “chaves estrangeiras” e “JOINS”, você está usando do jeito errado, ou, ao menos, não do jeito mais indicado.

Além disso, diversas entidades de pagamento (como bandeiras de cartão de crédito) não homologam sistemas cujos dados financeiros dos clientes não estejam em bancos de dados relacionais tradicionais. Obviamente isso não impede completamente o uso de MongoDB em sistemas financeiros, mas o restringe apenas a certas partes (como dados públicos).

Além disso, o MongoDB possui alguns concorrentes que possuem variações interessantes que de repente se encaixam melhor como solução ao seu problema. Outro mecanismo bem interessante desta categoria de banco de dados é o RethinkDB, que foca em consultas-push

real-time de dados do banco, ao invés de polling como geralmente se faz para atualizar a tela.

Para os amantes de .NET tem o RavenDB, que permite usa a sintaxe do LINQ e das expressões Lambda do C# direto na caixa, com curva de aprendizagem mínima.

Mais uma adição para seu conhecimento: Elasticsearch. Um mecanismo de pesquisa orientado a documentos poderosíssimo quando o assunto é pesquisa textual, foco da ferramenta. Ele é uma implementação do Apache Lucene, assim como Solr e Sphinx, mas muito superior à esses dois e bem mais popular atualmente também.

## Instalação e Testes

Diversos players de cloud computing fornecem versões de Mongo hospedadas e prontas para uso como **Umbler** e **mLab**, no entanto é muito importante um conhecimento básico de administração local de MongoDB para entender melhor como tudo funciona. Não focaremos aqui em nenhum aspecto de segurança, de alta disponibilidade, de escala ou sequer de administração avançada de MongoDB. Deixo todas estas questões para você ver junto à documentação oficial no site oficial, onde inclusive você pode estudar e tirar as certificações.

Caso ainda não tenha feito isso, acesse o site oficial do MongoDB e baixe gratuitamente a versão mais recente para o seu sistema operacional, que é a versão 3.4 na data em que escrevo este livro.

<http://www.mongodb.org>

Baixe o arquivo compactado e, no caso do Windows, rode o executável que extrairá os arquivos na sua pasta de Arquivos de Programas (não há uma instalação de verdade, apenas extração de arquivos), seguido de uma pasta server/versão, o que é está ok para a maioria dos casos, mas que eu prefiro colocar em C:\Mongo ou dentro de Applications no caso do Mac.

Dentro dessa pasta do Mongo podem existir outras pastas, mas a que nos interessa é a pasta bin. Nessa pasta estão uma coleção de utilitários de linha de comando que são o coração do MongoDB (no caso do Windows, todos terminam com .exe):

- » **mongod**: inicializa o servidor de banco de dados;
- » **mongo**: inicializa o cliente de banco de dados;
- » **mongodump**: realiza dump do banco (backup binário);
- » **mongorestore**: restaura dumps do banco (restore binário);
- » **mongoimport**: importa documentos JSON ou CSV pro seu banco;
- » **mongoexport**: exporta documentos JSON ou CSV do seu banco;
- » entre outros.

Para subir um servidor de MongoDB na sua máquina é muito fácil: execute o utilitário mongod via linha de comando como abaixo, onde dbpath é o caminho onde seus dados serão salvos (esta pasta já deve estar criada).

Código 26: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 C:\mongo\bin> mongod --dbpath C:\mongo\data
```

Isso irá iniciar o servidor do Mongo. Uma vez que apareça no prompt “[initandlisten] waiting for connections on port 27017”, está pronto, o servidor está executando corretamente e você já pode utilizá-lo, sem segurança alguma e na porta padrão 27017.

**Nota:** se já existir dados de um banco MongoDB na pasta data, o mesmo banco que está salvo lá ficará ativo novamente, o que é muito útil para os nossos testes.

Agora abra outro prompt de comando (o outro ficará executando o servidor) e novamente dentro da pasta bin do Mongo, digite:

Código 27: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 c:\mongo\bin> mongo
```

Após a conexão funcionar, se você olhar no prompt onde o servidor do Mongo está rodando, verá que uma conexão foi estabelecida e um sinal de “>” indicará que você já pode digitar os seus comandos e queries para enviar à essa conexão.

Ao contrário dos bancos relacionais, no MongoDB você não precisa construir a estrutura do seu banco previamente antes de sair utilizando ele. Tudo é criado conforme você for usando, o que não impede, é claro, que você planeje um pouco o que pretende fazer com o Mongo.

O comando abaixo no terminal cliente mostra os bancos existentes nesse servidor:

Código 28: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | show databases
```

Se é sua primeira execução ele deve listar as bases admin e local. Não usaremos nenhuma delas. Agora digite o seguinte comando para “usar” o banco de dados “workshop” (um banco que você sabe que não existe ainda):

Código 29: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | > use workshop
```

O terminal vai lhe avisar que o contexto da variável “db” mudou para o banco workshop, que nem mesmo existe ainda (mas não se preocupe com isso!). Essa variável “db” representa agora o banco workshop e podemos verificar quais coleções existem atualmente neste banco usando o comando abaixo:

Código 30: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | > show collections
```

Isso também não deve listar nada, mas não se importe com isso também. Assim como fazemos com objetos JS que queremos chamar funções, usaremos o db para listar os documentos de uma coleção de customers (clientes) da seguinte forma:

Código 31: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>



```
1 > db.customers.find()
```

find é a função para fazer consultas no MongoDB e, quando usada sem parâmetros, retorna todos os documentos da coleção. Obviamente não listará nada pois não inserimos nenhum documento ainda, o que vamos fazer agora com a função insert:

Código 32: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.insert({ nome: "Luiz", idade: 29})
```

A função insert espera um documento JSON por parâmetro com as informações que queremos inserir, sendo que além dessas informações o MongoDB vai inserir um campo \_id automático como chave primária desta coleção.

Como sabemos se funcionou? Além da resposta ao comando insert (nInserted indica quantos documentos foram inseridos com o comando), você pode executar o find novamente para ver que agora sim temos customers no nosso banco de dados. Além disso se executar o “show collections” e o “show databases” verá que agora sim possuímos uma coleção customers e uma base workshop nesse servidor.

Tudo foi criado a partir do primeiro insert e isso mostra que está tudo funcionando bem no seu servidor MongoDB!

## Comandos elementares

Este não é um livro focado em MongoDB, mas alguns comandos elementares são importantes que você conheça antes de voltarmos a codificar aplicações em Node.js, que a partir de agora terão persistência nesta fantástica tecnologia.

### Array Insert

Na seção anterior aprendemos a fazer um find() que retorna todos os documentos de uma coleção e um insert que insere um novo documento em uma coleção, além de outros comandos menores. Agora vamos adicionar mais alguns registros no seu terminal cliente mongo:

Código 33: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > custArray = [{ nome : "Fernando", idade : 29 },  
2 { nome : "Teste", "uf" : "RS" }] db.customers.  
3 insert(custArray)
```

**Atenção:** para o nome dos campos dos seus documentos e até mesmo para o nome das coleções do seu banco, use o padrão de nomes de variáveis JS (camel-case, sem acentos, sem espaços, etc).

**Nota:** no exemplo acima a variável `custArray` passa a existir durante toda a sessão do terminal a partir do comando seguinte.

Nesse exemplo passei um array com vários documentos para nossa função `insert` inserir na coleção `customers` e isso nos trás algumas coisas interessantes para serem debatidas. Primeiro, sim, você pode passar um array de documentos por parâmetro para o `insert`. Segundo, você notou que o segundo documento não possui “idade”? E que ele possui um campo “uf”?

## Find avançado

Para se certificar que todos documentos foram realmente inseridos na coleção, use o seguinte comando:

Código 34: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.find().pretty()
```

É o mesmo comando `find()` que usamos anteriormente, mas com a função `pretty()` no final para identar o resultado da função no terminal, ficando mais fácil de ler. Use e você vai notar a diferença, principalmente em consultas com vários resultados.

Mas voltando à questão do “uf”, ao contrário dos bancos relacionais, o MongoDB possui schema variável, ou seja, se somente um customer tiver “uf”, somente ele terá esse campo, não existe um schema pré-definido compartilhado entre todos os documentos, cada um é independente. Obviamente considerando que eles compartilham a

mesma coleção, é interessante que eles possuam coisas em comum, caso contrário não faz sentido guardar eles em uma mesma coleção.

Mas como fica isso nas consultas? E se eu quiser filtrar por “uf”? Não tem problema!

Essa é uma boa deixa para eu mostrar como filtrar um find() por um campo do documento:

Código 35: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.find({uf: "RS"})
```

Note que a função find pode receber um documento por parâmetro representando o filtro a ser aplicado sobre a coleção para retornar documentos. Nesse caso, disse ao find que retornasse todos os documentos que possuam o campo uf definido como “RS”. O resultado no seu terminal deve ser somente o customer de nome “Teste” (não vou falar do \_id dele aqui pois o valor muda completamente de um servidor MongoDB para outro).

**Atenção:** MongoDB é case-sensitive ao contrário dos bancos relacionais, então cuidado!

Experimente digitar outros valores ao invés de “RS” e verá que eles não retornam nada, afinal, não basta ter o campo uf, ele deve ser exatamente igual a “RS”.

Além de campos com valores específicos, esse parâmetro do find permite usar uma infinidade de operadores como por exemplo, trazer todos documentos que possuam a letra ‘a’ no nome:

Código 36: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.find({nome: { $regex: /a/ }})
```

Se você já mexeu com expressões regulares (regex) em JS antes, sabe exatamente como usar e o poder desse recurso junto a um banco de dados, sendo um equivalente muito mais poderoso ao LIKE dos bancos relacionais.

Mas e se eu quiser trazer todos os customers maiores de idade?

Código 37: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.find({idade: {$gte: 18}})
```

O operador \$gte (Greater Than or Equal) retorna todos os documentos que possuam o campo idade e que o valor do mesmo seja igual ou superior à 18. E podemos facilmente combinar filtros usando vírgulas dentro do documento passado por parâmetro, assim como fazemos quando queremos inserir campos em um documento:

Código 38: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.find({nome: "Luiz", idade: {$gte: 18}})
```

O que a expressão acima irá retornar?

Se você disse customers cujo nome sejam Luiz e que sejam maiores de idade, você acertou!

E a expressão abaixo:

Código 39: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.find({nome: { $regex: /a/ },  
2 idade: {$gte: 18}})
```

Customers cujo nome contenham a letra ‘a’ e que sejam maiores de idade, é claro!

Outros operadores que você pode usar junto ao filtro do find são:

- » **\$e**: exatamente igual (=)
- » **\$ne**: diferente (<> ou !=)
- » **\$gt**: maior do que (>)
- » **\$lt**: menor do que (<)
- » **\$lte**: menor ou igual a (<=)
- » **\$in**: o valor está contido em um array de possibilidades, como em um OU. Ex: {idade: {\$in: [10,12] }}
- » **\$all**: MongoDB permite campos com arrays. Ex: { tags: ["NodeJS", "MongoDB"] }. Com esse operador, você compara se seu campo multivalorado possui todos os valores de um array específico. Ex: {tags: {\$all: ["NodeJS", "Android"]}}
- » entre outros!

Você também pode usar `findOne` ao invés de `find` para retornar apenas o primeiro documento, ou ainda as funções `limit` e `skip` para limitar o número de documentos retornados e para ignorar alguns documentos, especificamente, da seguinte maneira:

Código 40: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.find().skip(1).limit(10)
```

No exemplo acima retornaremos 10 customers ignorando o primeiro existente na coleção.

E para ordenar? Usamos a função `sort` no final de todas as outras, com um documento indicando quais campos e se a ordenação por aquele campo é crescente (1) ou decrescente (-1), como abaixo em que retorno todos os customers ordenados pela idade:

Código 41: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.find().sort({idade: 1})
```

**Nota:** assim como nos bancos relacionais, os métodos de consulta retornam em ordem de chave primária por padrão, o que neste caso é o `_id`.

Ok, vimos como usar o find de maneiras bem interessantes e úteis, mas e os demais comandos de manipulação do banco?

### Update

Além do insert que vimos antes, também podemos atualizar documentos já existentes, por exemplo usando o comando update e derivados. O jeito mais simples (e mais burro) de atualizar um documento é chamando a função update na coleção com 2 parâmetros:

documento de filtro para saber qual(is) documento(s) será(ão) atualizado(s);  
novo documento que substituirá o antigo;

Como em:

Código 42: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.update({nome: "Luiz"}, {nome:  
2 "Luiz", idade: 29, uf: "RS"})
```

Como resultado você deve ter um nModified maior do que 1, mostrando quantos documentos foram atualizados.

Por que essa é a maneira mais burra de fazer um update? Porque além de perigosa ela exige que você passe o documento completo a ser atualizado no segundo parâmetro, pois ele substituirá o original!

Primeira regra do update inteligente: se você quer atualizar um documento apenas, comece usando updateOne ao invés de update. O updateOne vai te obrigar a usar operadores ao invés de um documento inteiro para a atualização, o que é muito mais seguro.

Segunda regra do update inteligente: sempre que possível, use a chave primária (\_id) como filtro da atualização, pois ela é sempre única dentro da coleção.

Terceira regra do update inteligente: sempre use operadores ao invés de documentos inteiros no segundo parâmetro, independente do número de documentos que serão atualizados.

Mas que operadores são esses?

Assim como o find possui operadores de filtro, o update possui operadores de atualização. Se eu quero, por exemplo, mudar apenas o nome de um customer, eu não preciso enviar todo o documento do respectivo customer com o nome alterado, mas sim apenas a expressão de alteração do nome, como abaixo (já usando o `_id` como filtro, que é mais seguro):

Código 43: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.updateOne({_id: ObjectId
2 ("59ab46e433959e2724be2cbd")}, {$set: {idade: 28}})
```

**Nota:** para saber o `_id` correto do seu update, faça um find primeiro e não tente copiar o meu pois não vai repetir.

Esta função vai alterar (operador `$set`) a idade para o valor 28 do documento cujo `_id` seja “59ab46e433959e2724be2cbd” (note que usei uma função `ObjectId` para converter, pois esse valor não é uma string).

**Nota:** você pode usar null se quiser “limpar” um campo.

O operador `$set` recebe um documento contendo todos os campos que devem ser alterados e seus respectivos novos valores. Qualquer campo do documento original que não seja indicado no set continuará com os valores originais.

**Atenção:** o operador `$set` não adiciona campos novos em um documento, somente altera valores de campos já existentes.

Não importa o valor que ela tenha antes, o operador `$set` vai sobrescrevê-lo. Agora, se o valor anterior importa, como quando queremos incrementar o valor de um campo, não se usa o operador `$set`, mas sim outros operadores. A lista dos mais úteis operadores de update estão abaixo:

- » **\$unset**: remove o respectivo campo do documento;
- » **\$inc**: incrementa o valor original do campo com o valor especificado;
- » **\$mul**: multiplica o valor original do campo com o valor especificado;
- » **\$rename**: muda o nome do campo para o nome especificado;

Além disso, existe um terceiro parâmetro oculto no update que são as opções de update. Dentre elas, existe uma muito interessante do MongoDB: **upsert**, como abaixo:

Código 44: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 > db.customers.updateOne({nome: "LuizTools"},  
2 {nome: "LuizTools", uf: "RS"}, {upsert: true})
```

Um **upsert** é um **update** como qualquer outro, ou seja, vai atualizar o documento que atender ao filtro passado como primeiro parâmetro, porém, se não existir nenhum documento com o respectivo filtro, ele será inserido, como se fosse um **insert**.

**upsert** = **update** + **insert**

Eu já falei como amo esse banco de dados? :D

## Delete

Pra encerrar o nosso conjunto de comandos mais elementares do MongoDB falta o **delete**.

Existe uma função **delete** e uma **deleteOne**, o que a essa altura do campeonato você já deve saber a diferença. Além disso, assim como o **find** e o **update**, o primeiro parâmetro do **delete** é o filtro que vai definir quais documentos serão deletados e todos os operadores normais do **find** são aplicáveis.

Sendo assim, de maneira bem direta:

Código 45: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>



```
1 | > db.customers.delete({nome: "Luiz"})
```

Vai excluir todos os clientes cujo nome seja igual a “Luiz”.

Simples, não?!

Obviamente existem coisas muito mais avançadas do que esse rápido tópico de MongoDB. Lhe encorajo a dar uma olhada no site oficial do banco de dados onde há a seção de documentação, vários tutoriais e até mesmo a possibilidade de tirar certificações online para garantir que você realmente entendeu a tecnologia.

---

# CRIANDO WEB APIS

5

“

Some of the best programming is done on paper, really. Putting it into the computer is just a minor detail.

- *Max Kanat-Alexander*

”

Uma Web API é uma interface de programação de aplicações (API) tanto para um servidor quanto um navegador. É utilizada para se conseguir recuperar somente o valor necessitado num banco de dados de um site, por exemplo.

Uma web API server-side, que é o que nos interessa aqui, é uma interface programática consistente de um ou mais endpoints publicamente expostos para um sistema definido de mensagens pedido-resposta (request-response), tipicamente expressado em JSON ou XML, que é exposto via a internet—mais comumente por meio de um servidor web baseado em HTTP. Resumidamente podemos dizer que são versões mais “leves” e menos “burocráticas” que o seu antecessor, os web services.

Falando de tecnologias, qualquer linguagem server-side pode ser usada para programar web APIs. Visando tornar o serviço mais organizado é geralmente utilizado algum padrão de mercado para o formato de transmissão de dados, como XML, JSON ou CSV e visando estabelecer um padrão de comunicação, é usado algum protocolo, geralmente REST.

Com esses três itens você tem o suficiente para escrever uma web API, embora a adição de uma camada de dados seja quase 100% presente em todos web services.

Todo o núcleo de seu serviço fica longe do usuário, em um servidor web, que tratará as requisições de todos clientes em um único lugar, com um único código fonte. É como se fosse um software que não tem interface, mas que apenas recebe e responde requisições. Esse software pode ser escrito em praticamente qualquer linguagem de programação, desde que o servidor web esteja configurado para tal. Algumas tecnologias populares atualmente para desenvolvimento de web APIs:

- Ruby
- PHP
- ASP.NET
- Node.js
- Python

Como dito anteriormente, muito provavelmente a sua web API terá uma base de dados, o que exigirá o conhecimento de alguma linguagem de consulta/manipulação de dados, como o SQL por exemplo, no caso dos bancos relacionais, ou JavaScript, no caso do MongoDB.

Durante o desenvolvimento dos exercícios anteriores fizemos uso do protocolo HTTP para comunicação pela Internet, mas apenas de uma parte dele e sem um entendimento maior do seu comportamento. Tópico que elucidaremos melhor agora.

## Criando uma Web API

Vamos começar criando um novo projeto Express, mas desta vez não usaremos o express-generator, pois não precisamos em uma Web API de muitas coisas que ele gera automaticamente.

Preparando o projeto

Comece criando uma pasta webapi para guardarmos nossos fontes dentro. Dentro dessa pasta, crie um arquivo app.js vazio, onde codificaremos o coração da nossa web API mais tarde.

Agora acesse o terminal e navegue até essa pasta, usando o comando abaixo para criar o nosso arquivo package.json:

Código 46: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | npm init
```

O NPM init é um recurso interativo para criação do package.json. Ele vai lhe fazer diversas perguntas que você pode apenas apertar Enter para ficar a resposta default (indicada entre parênteses) ou escrever suas respostas personalizadas.

Uma das perguntas é o entry-point, ou arquivo de “start” da sua aplicação (seja ela uma web API ou não). Se você criou o app.js conforme mencionou, o npm init irá lhe sugerir este arquivo como sendo o entry-point.

O próximo passo é instalar algumas dependências que vamos precisar, usando o npm install:

```
1 | npm i express body-parser
```

Código 47: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

Note que desta vez usei ‘i’ ao invés de ‘install’ e que informei o nome de mais de um pacote de uma vez só, o que é perfeitamente possível com o NPM. Não sei se reparou, mas também não usei a opção “-S” pois nas versões mais recentes do NPM ele já salva a dependência no package.json automaticamente.

Com essas dependências instaladas, vamos começar a programar!

**Atenção:** usaremos o mesmo banco de dados MongoDB que criamos anteriormente (banco workshop, com coleção customers). Então não se esqueça de garantir que ele esteja executando corretamente com o mongod.

## MongoDB Driver

Existem diferentes formas de se conectar a bancos de dados usando Node.js, usarei aqui o driver oficial dos criadores do MongoDB que se chama apenas mongodb.

Para instalar essa dependência na sua aplicação Node.js (que chamamos de webapi no último tópico, lembra?), rode o seguinte comando que além de baixar a dependência também salva a mesma no seu package.json:

Código 48: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | npm install mongodb
```

Para deixar nossa aplicação minimamente organizada não vamos sair por aí escrevendo lógica de acesso à dados. Vamos centralizar tudo

o que for responsabilidade do MongoDB dentro de um novo módulo chamado `db.js`, que nada mais é do que um arquivo `db.js` na raiz do nosso projeto.

Esse arquivo será o responsável pela conexão e manipulação do nosso banco de dados, usando o driver nativo do MongoDB. Adicione estas linhas nele:

Código 49: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 const MongoClient = require("mongodb").MongoClient
2 MongoClient.connect("mongodb://localhost:27017/
3 workshop")
4   .then(conn => global.conn = conn.db("workshop"))
5   .catch(err => console.log(err))
6
7 module.exports = { }
```

Uau, muitas coisas novas para eu explicar aqui!

Vamos lá!

Na primeira linha, carregamos o módulo `mongodb` usando o comando `require` e de tudo que este módulo exporta vamos usar apenas o objeto `MongoClient`, que armazenamos em uma variável local.

Com essa variável carregada, usamos a função `connect` passando a connection string. Caso não tenha experiência com programação, uma connection string é uma linha de texto informando os dados de conexão com o banco. No caso do MongoDB ela deve ser nesse formato:

```
1 mongodb://usuario:senha@servidor:porta/banco
```

Como não temos usuário e senha no nosso banco de dados, omitiremos essas duas informações.

**Nota:** existem formas mais elegantes de armazenar essas informações, mas deixarei para que você descubra em materiais mais avançados ou na prática, trabalhando profissionalmente com estas tecnologias.

A terceira e quarta linhas podem parecer muito estranhas mas são um formato mais recente de programação JavaScript chamada Promises. Promises são um jeito mais elegante do que callbacks para lidar com funções assíncronas. A conexão com o banco de dados pode demorar um pouco dependendo de onde ele esteja hospedado e sabemos que o Node.js não permite bloqueio da thread principal, por isso usamos a função ‘then’ para dizer qual função será executada (callback) após a conexão ser estabelecida com sucesso.

Nesse caso usamos outro conceito mais recente que são as arrow functions, uma notação especial para funções anônimas onde declaramos os parâmetros (entre parênteses se houver mais de um) seguido de um ‘=>’ e depois os comandos da função (entre chaves se houver mais de um).

```
1 | (parametros) => { comandos }
```

No caso da function connection do MongoClient, ela retorna um objeto de conexão em caso de sucesso, que vamos armazenar globalmente usando a função abaixo ao mesmo tempo que selecionamos o banco que iremos manipular (workshop):

Código 50: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 | conn => global.conn = conn.db("workshop")
```

Essa é uma sugestão bem comum, fazer uma única conexão com o MongoDB e compartilhá-la globalmente com toda sua aplicação.

Mas e se a conexão não for estabelecida com sucesso?

Nesse caso usamos a função ‘catch’ passando outra função de callback, mas essa para o caso de dar erro no connect. Em nosso exemplo apenas mandei imprimir no console a mensagem de erro.

A última linha do nosso db.js contém o module.exports que é a instrução que permite compartilhar objetos com o restante da aplicação. Como já compartilhei globalmente a nossa conexão com o MongoDB (conn), não há nada para colocar aqui por enquanto.

Agora abra o arquivo app.js do seu projeto Node.js e adicione a seguinte linha no início dele:

Código 51: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 global.db = require('../db')
```

Nesta linha nós estamos carregando o módulo db que acabamos de criar e guardamos o resultado dele em uma variável global. Ao carregarmos o módulo db, acabamos fazendo a conexão com o Mongo e retornamos aquele objeto vazio do module.exports, lembra? Usaremos ele mais tarde, quando possuir mais valor.

**Nota:** módulos do Node.js podem ser carregados apenas com o nome do módulo pois o Node procura primeiro na pasta node\_modules. Já módulos criados por você devem ser carregados passando o caminho relativo até eles, neste caso usei '../' para voltar uma pasta em relação à bin onde o www está salvo. Caso estivessem na mesma pasta eu usaria './'.

Continuando nosso app.js, definiremos algumas variáveis locais através do carregamento de alguns módulos (repeti a primeira linha para você entender onde devem ir as demais):

Código 52: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 global.db = require('../db')
2 const express = require('express')
3 const app = express()
4 const bodyParser = require('body-parser')
5 const port = 3000 //porta padrão
```



Quando carrego módulo db, ele automaticamente já fará a conexão com o banco de dados. Já o módulo express é carregado para criar o objeto da nossa aplicação (app). Antes de escrever o código que inicia o servidor web temos de configurar algumas coisas no app.js.

Primeiro, vamos configurar o app para usar o módulo body-parser visando a serialização e desserialização correta do body das requisições HTTP usando x-www-form-urlencoded e JSON:

Código 53: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

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

Agora, vamos configurar como irá funcionar o roteamento das requisições:

Código 54: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

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

Aqui criei uma rota GET default que apenas retorna um JSON avisando que a API está funcionando. Lembra que disse que não havia necessidade de usar o express-generator aqui pois ele ia gerar mais coisas do que precisaríamos? Pois é, um exemplo são as views.

Web APIs não possuem views, elas retornam no corpo das requisições JSON ou XML, ao invés de HTML. Sendo assim, note como na função de callback do router.get (que eu usei uma arrow function) eu usei res.json ao invés de res.render, pois o retorno será um objeto JSON.

Preste atenção ao código do `router.get`, é aqui que vamos definir as nossas outras rotas mais tarde.

E por fim, vamos escrever o código que inicializa o nosso servidor no final do `app.js`:

Código 55: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

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

Com isso, já podemos iniciar a nossa web API e ver se ela está funcionando. Curioso como em Node.js conseguimos criar coisas incríveis como um webserver HTTP com tão poucas linhas de código, não é mesmo?

Código 56: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 node app
```

Aqui usei o comando `node` ao invés de iniciar o projeto com o `npm start`, indicando qual o arquivo que deve ser executado para iniciar a aplicação/servidor.

O resultado no navegador, acessando `localhost:3000` deve ser:



Como não temos views, todos os testes de nossa API terão como resultado objetos JSON, então vá se acostumando.

## Listando os clientes na API

Agora que temos o projeto configurado e com a estrutura básica pronta, podemos programar e testar nossa API facilmente.

Para cada operação desejamos oferecer através da nossa API devemos criar uma rota em nosso `app.js`, junto ao `router.get` que já deixei lá por padrão no tópico anterior (lembra?).

Além disso, cada rota vai exigir a existência de uma função no módulo `db.js` para fazer o acesso ao banco de dados.

Assim, vamos começar com uma operação muito simples: listar todos os clientes do banco de dados.

Para isso, no `db.js` você deve incluir uma função `findCustomers` e exportá-la como abaixo:

Código 57: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 function findCustomers(callback) {  
2     global.conn.collection('customers').  
3     find().toArray(callback)  
4 }  
5  
6 module.exports = {findCustomers}
```

Nesta função ‘`findCustomers`’, esperamos uma função de callback por parâmetro que será executada quando a consulta no Mongo terminar. Isso porque as consultas no Mongo são assíncronas e o único jeito de conseguir saber quando ela terminou é executando um callback.

A consulta aqui é bem direta: usamos a conexão global `conn` para navegar até a `collection` de `customers` e fazer um `find` sem filtro algum. O resultado desse `find` é um cursor, então usamos o `toArray` para convertê-lo para um array e quando terminar, chamamos o callback para receber o retorno.

Agora no app.js, adicione a seguinte rota que tratará requisições GET / clientes na nossa API:

Código 58: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 // GET /clientes
2 router.get('/clientes', (req, res) => global.
3 db.findCustomers((err, docs) => {
4     if(err) res.status(500).json(err)
5     else res.json(docs)
6 })))
```

Note que em caso de erro (falha de conexão com o banco, por exemplo) nossa API vai retornar um código 500 (Internal Server Error) com um JSON informando a causa do erro. Isso nem sempre é o mais indicado, você pode logar o erro para si e devolver uma mensagem amigável ao requisitante, por exemplo.

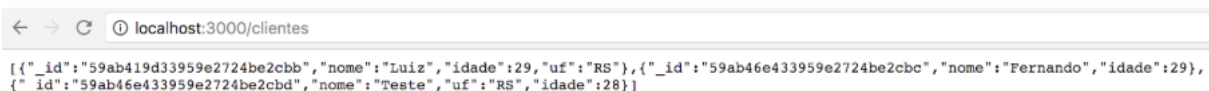
**Nota:** você não precisa definir o status das respostas quando elas forem um 200 OK, pois esse é o valor padrão de status.

Isto é o bastante para a listagem funcionar. Salve o arquivo e reinicie o servidor Node.js. Ainda se lembra de como fazer isso? Abra o prompt de comando, derrube o processo atual (se houver) com Ctrl+C e depois:

Código 59: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 npm start
```

Agora abra seu navegador, acesse <http://localhost:3000/> e maravilhe-se com o resultado. Como o acesso do navegador é sempre GET, você deve ver um array JSON com todos os clientes do seu banco de dados nele:



```
{ "_id": "59ab419d33959e2724be2cbb", "nome": "Luiz", "idade": 29, "uf": "RS" }, { "_id": "59ab46e433959e2724be2cbc", "nome": "Fernando", "idade": 29 }, { "_id": "59ab46e433959e2724be2cbd", "nome": "Teste", "uf": "RS", "idade": 28 }
```

Mas e se quisermos oferecer um endpoint que permita ver apenas um cliente específico?

Podemos fazer isso criando uma nova função no db.js (e exportando-a no module.exports, é claro):

Código 60: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 const ObjectId = require("mongodb").ObjectId
2 function findCustomer(id, callback) {
3     global.conn.collection('customers').
4     findOne(new ObjectId(id), callback)
5 }
6
7 module.exports = {findCustomers, findCustomer}
```

Aqui precisei carregar o objeto ObjectId do módulo mongodb para poder converter o id, que virá como string na requisição, para o tipo correto que o MongoDB entende como sendo a chave primária das suas coleções.

**Nota:** somente a primeira vez que usamos o comando require em um módulo é que ele realmente é carregado na aplicação. Nas demais vezes ele é chamado da memória, reaproveitando o carregamento inicial, então não se preocupe com requires repetidos em meu código.

Com a função pronta no db.js, vamos criar uma nova rota no app.js:

Código 61: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 // GET /clientes/{id}
2 router.get('/clientes/:id', (req, res) =>
3   global.db.findCustomer(req.params.id, (err,
4   doc) => {
5     if(err) res.status(500).json(err)
6     else res.json(doc)
7   } ) )
```

Existem poucas diferenças em relação à outra rota, como o parâmetro :id no caminho da URL e a chamada à função correspondente.

Para testar, pegue um dos \_ids dos clientes listados no teste anterior e experimente acessar uma URL localhost:3000/clientes/id trocando 'id' pelo respectivo id que deseja pesquisar. O retorno no navegador deve ser um único objeto JSON.

Caso o cliente não exista, deve aparecer null (o cliente não existir não é um erro). Se quiser você inclusive pode tratar isso facilmente com um if.

Poderíamos ficar aqui criando vários endpoints de retorno de clientes diferentes, usando os operadores para filtrar em nossos finds, mas acho que você já pegou a ideia.

Cadastrando novos clientes

Listar dados é moleza e salvar dados no MongoDB não é algo particularmente difícil.

Mas antes de mexer nas rotas novamente, vamos alterar nosso db.js para incluir uma nova função, desta vez para inserir clientes usando a conexão global e, novamente, executando um callback ao seu término:

E para salvar um novo cliente no banco de dados usando a nossa API?

Como sempre, vamos começar pela função no db.js, que não tem nada de mais:

Código 62: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 function insertCustomer(customer, callback){
2     global.conn.collection('customers').
3     insert(customer, callback)
4 }
5
6 module.exports = {findCustomers, findCustomer,
7 insertCustomer}
```

Para em seguida, criar a nova rota no app.js:

Código 63: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 // POST /clientes
2 router.post('/clientes', (req, res) => {
3     const customer = req.body
4     global.db.insertCustomer(customer, (err,
5 result) => {
6         if(err) res.status(500).json(err)
7         else res.json({ message: 'Cliente
8 cadastrado com sucesso!'})
9     })
10 })
```

Note que este código já começa diferente dos demais, com `router.post` ao invés de `router.get`, indicando que esta rota tratará POSTs no endpoint / clientes. Na sequência, pegamos o body da requisição onde está o JSON do cliente a ser salvo na base de dados. Obviamente no mundo real você irá querer colocar validações, tratamento de erros e tudo mais.

Note que aqui estou usando JSON como padrão para requisições e respostas. Caso seja enviado dados em outro formato, causará erros de comunicação com a API.

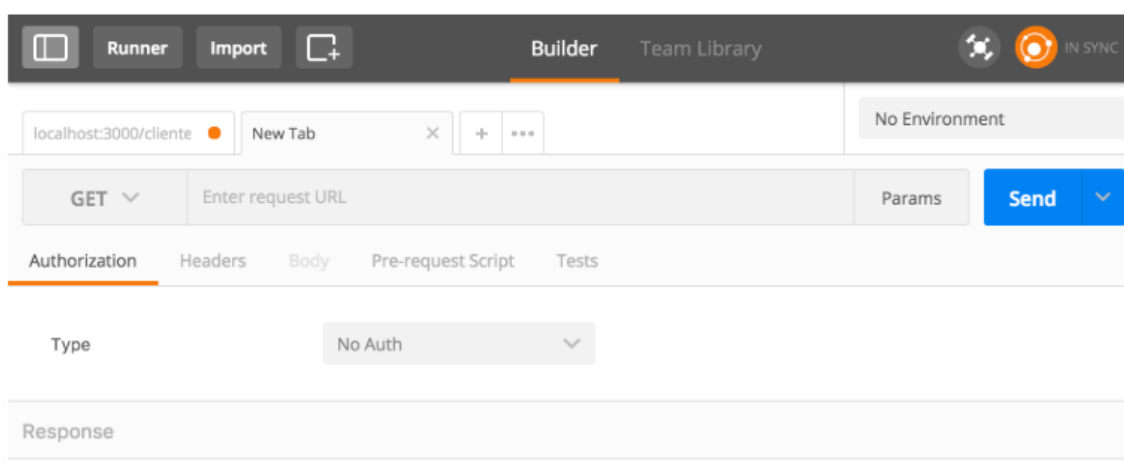
Mas e agora, como vamos testar um POST? Pela barra de endereço do navegador não dá, pois ela só faz GET.

Temos diversas alternativas no mercado, sendo que vou falar de duas aqui.

Primeiro, usando o POSTMAN. Eu já falei dele lá no início do livro, no capítulo em que montamos o ambiente completo para todos exercícios do livro. Caso não tenha baixado e instalado ainda, faça-o usando o link abaixo:

<https://www.getpostman.com/>

Basicamente o POSTMAN é uma ferramenta que lhe ajuda a testar APIs visualmente. Você verá a tela abaixo ao abrir o POSTMAN:



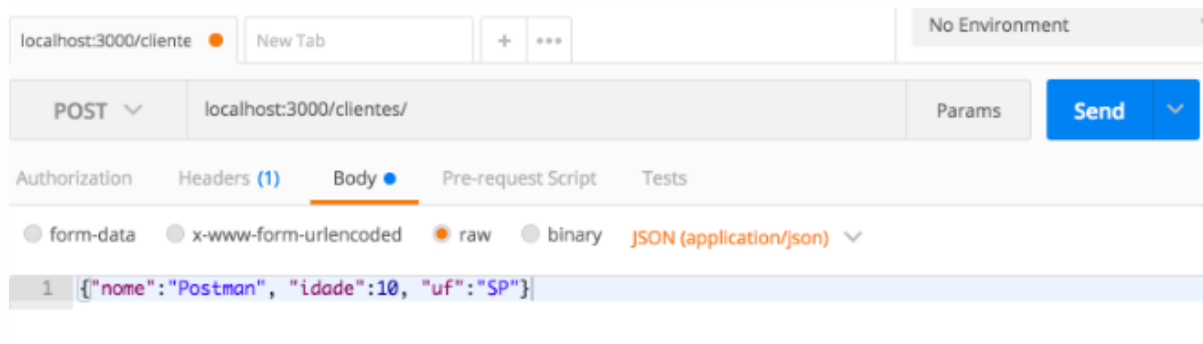


No primeiro select à esquerda você encontra os verbos HTTP, selecione POST nele.

Na barra de endereço, digite o endpoint no qual vamos fazer o POST, em nosso caso <http://localhost:3000/clientes>.

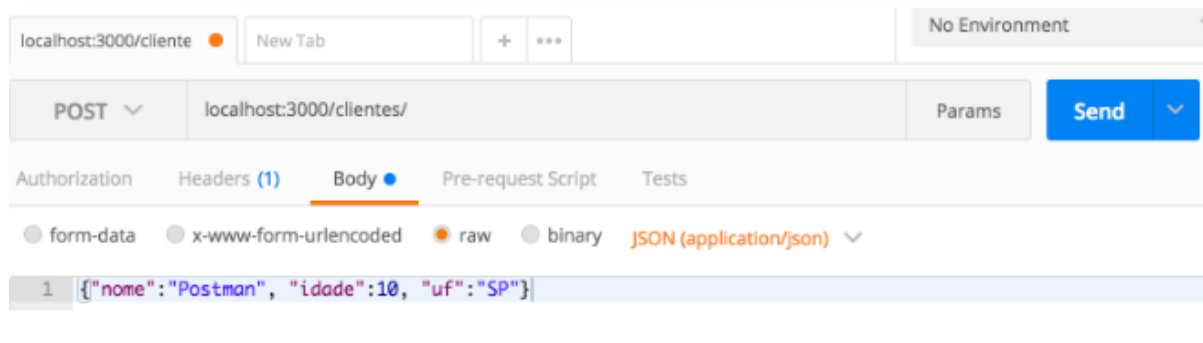
Logo abaixo temos algumas abas. Vá na aba Headers (cabeçalhos) e adicione uma linha com a seguinte configuração: key = Content-Type e value = application/json. Isso diz ao POSTMAN que essa requisição está enviando dados no formato JSON em seu corpo (body).

E por fim, vá na aba Body, marque a opção 'raw' e escreva na área de texto abaixo um objeto JSON de cliente (não esqueça de colocar aspas entre os nomes das propriedades aqui), como os inúmeros que já escrevemos antes.



Agora sim podemos testar!

Considerando que você já esteja com seu servidor atualizado e rodando, clique no enorme botão Send do POSTMAN e sua requisição será enviada. Quando isso acontece, o POSTMAN exibe em uma área logo abaixo da requisição, a resposta (response) da requisição, como abaixo:



Note que é mostrado o corpo da resposta (nesse caso o JSON de sucesso) e o status da mesma (nesse caso um 200 OK).

Mas será que funcionou mesmo?

Basta digitarmos localhost:3000/clientes em nosso navegador (ou construir uma requisição GET no POSTMAN) e veremos que nosso novo customer está lá!\

**Nota:** Outra forma de testar, que alguns consideram muito mais prática é usando o cURL via linha de comando, como no exemplo abaixo:

Código 64: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 curl -X POST -d '{"nome':'Curl', 'idade': 11,  
2 'uf': 'RJ'}" http://localhost:3000/clientes
```

Fica a seu critério como deseja testar. O POSTMAN tem algumas vantagens na minha opinião, como permitir salvar as requisições, automatizar testes e sincronização com sua Google Account.

## Atualizando clientes

Mas e se eu quiser atualizar um cliente?

Se for uma atualização de um cliente inteiro, neste caso deve ser usado um PUT. Para fazer isso, adivinha, basta criar uma nova função no db.js:

```
1 function updateCustomer(id, customer,  
2   callback) {  
3     global.conn.collection('customers').  
4     update({_id: new ObjectId(id)}, customer,  
5     callback)  
6   }  
7  
8   module.exports = {findCustomers, findCustomer,  
9   insertCustomer, updateCustomer}
```

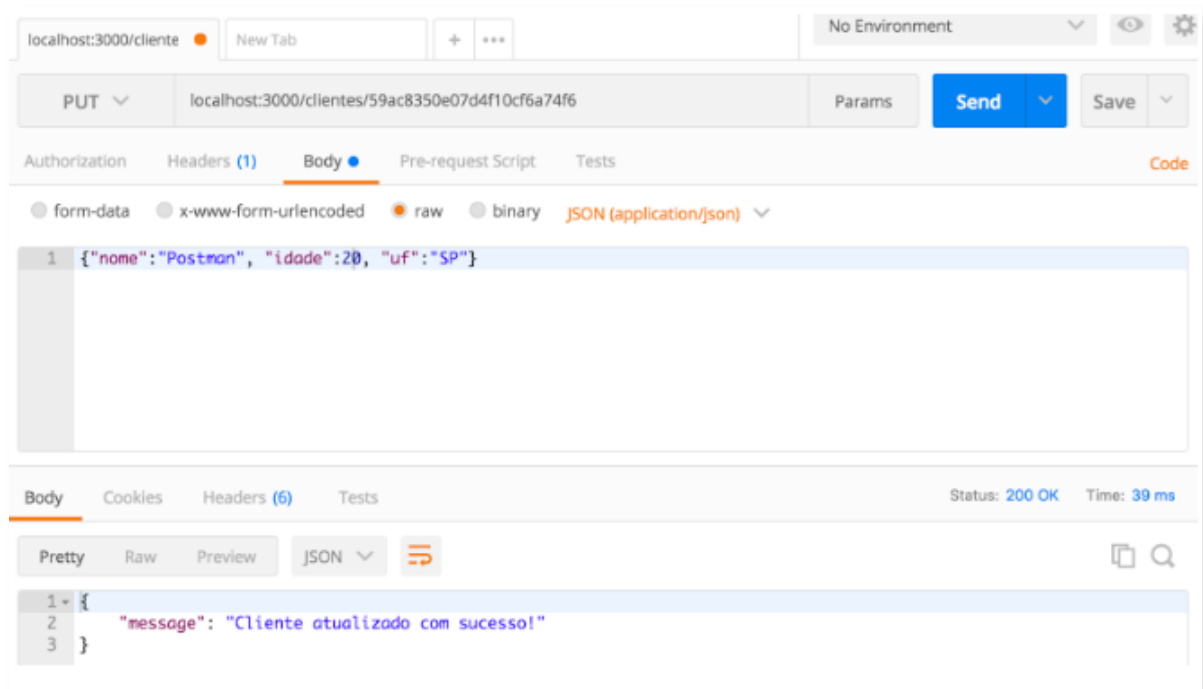
Aqui eu aproveitei o mesmo objeto ObjectId que havia carregado anteriormente para fazer o insertCustomer, lembra? Isso porque o update será baseado no \_id do customer, para evitar enganos.

Agora no nosso app.js...

```
1 // PUT /clientes/{id}  
2 router.put('/clientes/:id', (req, res) => {  
3   const id = req.params.id  
4   const customer = req.body  
5   global.db.updateCustomer(id, customer,  
6   (err, result) => {  
7     if(err) res.status(500).json(err)  
8     else res.json({ message: 'Cliente  
9   atualizado com sucesso!'})  
10   })  
11 })
```

Essa nova rota é praticamente idêntica à do `router.post`, com pequenas variações. É muito importante lembrar que o `update` substitui o `customer` com o `id` passado por parâmetro pelo objeto JSON passado pelo `body` da requisição. Então cuidado!

Para testar, configure uma requisição PUT no POSTMAN como abaixo, não esquecendo que o `id` da URL você deve pegar de algum cliente da sua base, pois eles não serão iguais. Além disso, a aba Headers tem o Content-Type definido como `application/json` e o verbo é PUT:



Incluí no mesmo print acima a resposta à minha requisição.

**Nota:** Você pode obter o mesmo resultado usando `cURL`:

Código 67: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 curl -X PUT -d '{"nome':'Postman', 'idade':  
2 20, 'uf': 'SP'}" http://localhost:3000/  
3 clientes/59ac8350e07d4f10cf6a74f6
```

Update: check!

Para fazer um update parcial é um pouquinho mais complicado, mas não muito, pois você terá de usar o operador \$set do MongoDB e usar o verbo PATCH.

Vamos começar criando a função no db.js:

Código 68: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

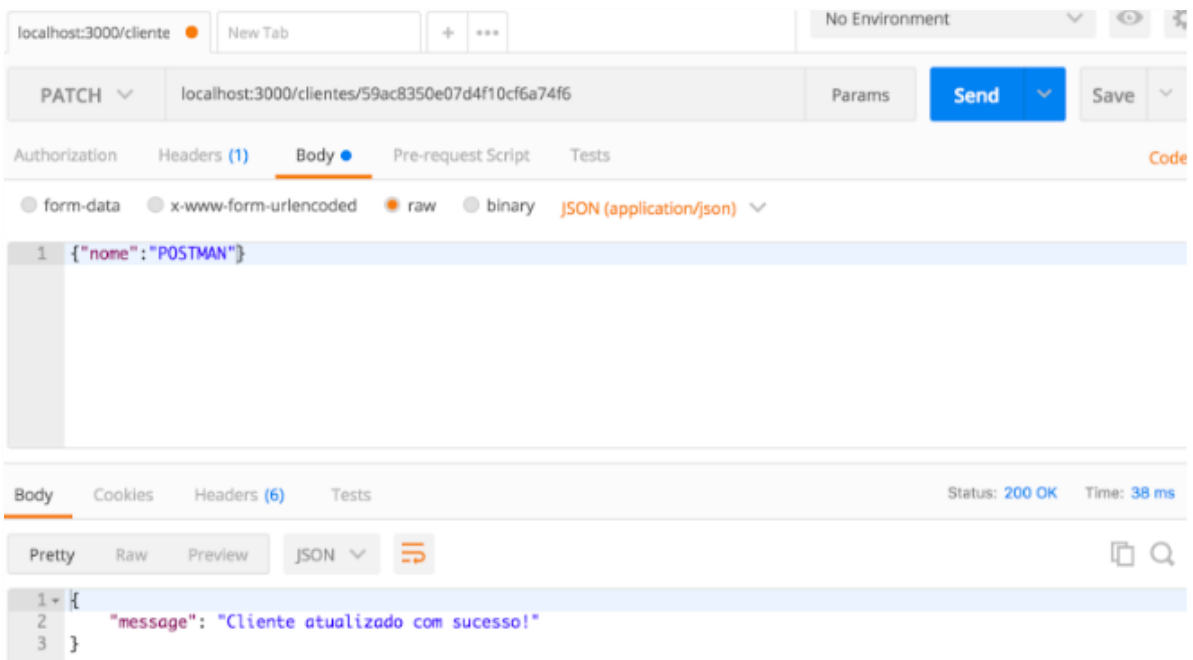
```
1 function patchCustomer(id, updates, callback)
2 {
3     global.conn.collection('customers').
4     update({_id: new ObjectId(id)}, { $set:
5     updates }, callback)
6 }
7
8 module.exports = {findCustomers, findCustomer,
9 insertCustomer, updateCustomer,
10 patchCustomer}
```

Note que como segundo parâmetro, ao invés de passar o objeto diretamente para função update eu usei o operador \$set e passarei o objeto pra ele. Isso fará com que o objeto updates possa conter somente os campos que eu desejo alterar nesse cliente, deixando os demais intactos. Muito mais prático e seguro do ponto de vista da integridade dos dados.

Depois faça a rota no app.js:

Código 69: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 // PATCH /clientes/{id}
2 router.patch('/clientes/:id', (req, res) => {
3   const id = req.params.id
4   const updates = req.body
5   global.db.patchCustomer(id, updates, (err,
6 result) => {
7     if(err) res.status(500).json(err)
8     else res.json({ message: 'Cliente
9 atualizado com sucesso!'})
10   })
11 })
```



**Nota:** via cURL você pode testar usando:

Código 70: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 curl -X PATCH -d '{"nome':'POSTMAN'}"  
2 http://localhost:3000/  
3 clientes/59ac8350e07d4f10cf6a74f6
```

## Excluindo clientes

Para encerrar a construção da nossa API REST, falta nosso delete que é muito simples e rápido de fazer. A começar pelo db.js:

Código 71: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```
1 function deleteCustomer(id, callback){  
2     global.conn.collection('customers').  
3     deleteOne({_id: new ObjectId(id)}, callback)  
4 }  
5  
6 module.exports = {findCustomers, findCustomer,  
7 insertCustomer, updateCustomer,  
8 patchCustomer, deleteCustomer}
```

E finalmente, a última rota no nosso app.js:

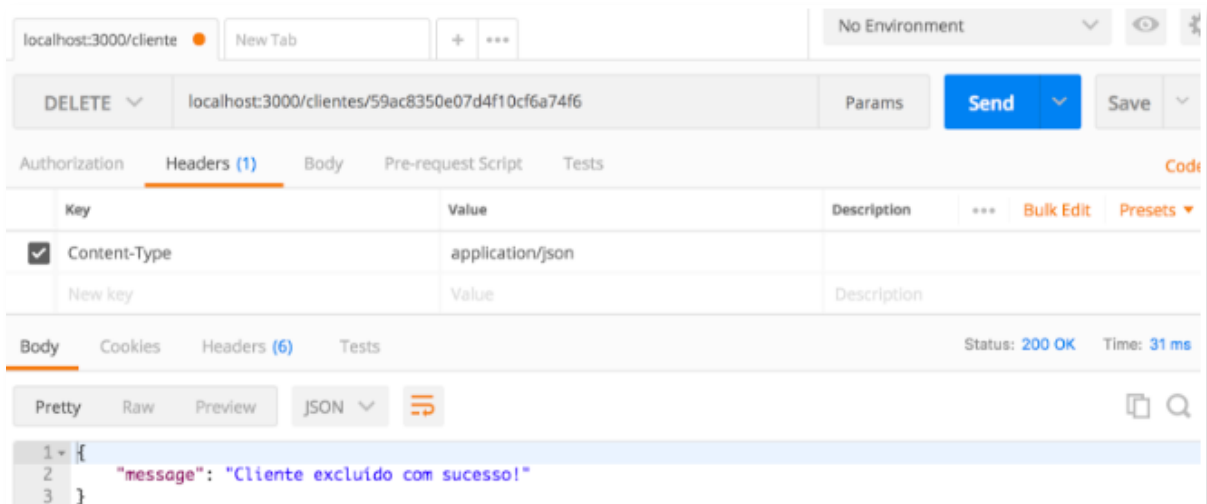
Código 72: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>

```

1 // DELETE /clientes/{id}
2 router.delete('/clientes/:id', (req, res) =>
3 {
4     const id = req.params.id
5     global.db.deleteCustomer(id, (err,
6 result) => {
7         if(err) res.status(500).json(err)
8         else res.json({ message: 'Cliente
9 excluído com sucesso!'})
10     })
11 })

```

Para testar vamos recorrer novamente ao POSTMAN que apenas precisa ter o verbo definido como DELETE e a URL informando o id do cliente a ser excluído, nada além disso.



**Nota:** e em cURL:

Código 73: disponível em <http://www.luiztools.com.br/livro-node-api-fontes>



```
1 curl -X DELETE http://localhost:3000/  
2 clientes/59ac8350e07d4f10cf6a74f6
```

E com isso encerramos a criação de nossa web API REST usando Node.js, Express e MongoDB.

# SEGUINDO EM FRENTE

---

“

A code is like love, it has created with clear intentions at the beginning, but it can get complicated.

- **Gerry Geek**

”

Este livro termina aqui.

Pois é, certamente você está agora com uma vontade louca de aprender mais e criar aplicações incríveis com Node.js, que resolvam problemas das empresas e de quebra que o deixem cheio de dinheiro na conta bancária, não é mesmo?

Pois é, eu também! :)

Este livro é pequeno se comparado com o universo de possibilidades que o Node.js nos traz. Como professor, costumo dividir o aprendizado de alguma tecnologia (como Node.js) em duas grandes etapas: aprender o básico e executar o que foi aprendido no mercado, para alcançar os níveis intermediários e avançados. Acho que este guia atende bem ao primeiro requisito, mas o segundo só depende de você.

De nada adianta saber muita teoria se você não aplicar ela. Então agora que terminou de ler este livro e já conhece o básico sobre criar aplicações com esta fantástica plataforma, inicie hoje mesmo (não importa se for tarde) um projeto de aplicação que use o que aprendeu. Caso não tenha nenhuma ideia, cadastre-se agora mesmo em alguma plataforma de freelancing. Mesmo que não ganhe muito dinheiro em seus primeiros projetos, somente chegarão os projetos grandes, bem pagos e realmente interessantes depois que você tiver experiência.

Me despeço de você leitor com uma sensação de dever cumprido. Caso acredite que está pronto para ainda mais tutoriais bacanas, sugiro dar uma olhada em meu blog <http://www.luiztools.com.br> ou no meu livro ainda mais completo de Node.js, o **[Programação Web com Node.js](#)**.

Caso tenha gostado do material, indique esse livro a um amigo que também deseja aprender a programar com Node. Não tenha medo da concorrência e abrace a ideia de ter um sócio que possa lhe ajudar nos projetos.

Caso não tenha gostado tanto assim, envie suas dúvidas, críticas e sugestões para [contato@luiztools.com.br](mailto:contato@luiztools.com.br) que estou sempre disposto a melhorar.

Um abraço e até a próxima!

## Curtiu o Livro?

Aproveita e me segue nas redes sociais:



[Facebook.com/luiztools](https://Facebook.com/luiztools)



[Twitter.com/luiztools](https://Twitter.com/luiztools)



[LinkedIn.com/in/luizfduartejr](https://LinkedIn.com/in/luizfduartejr)

Conheça meus outros livros:



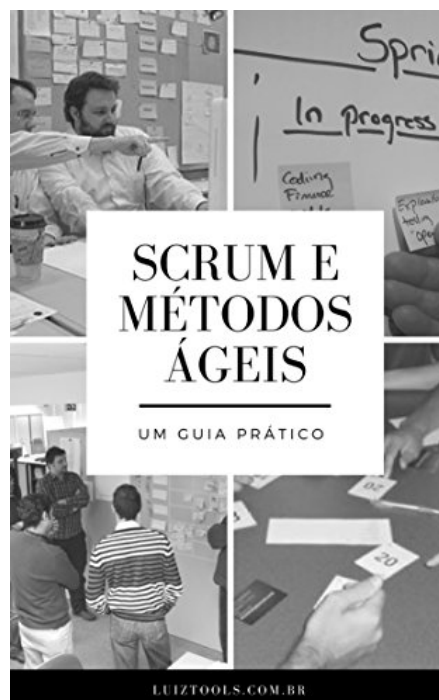
[Programação Web  
com Node.js](#)



[MongoDB para  
Iniciantes](#)

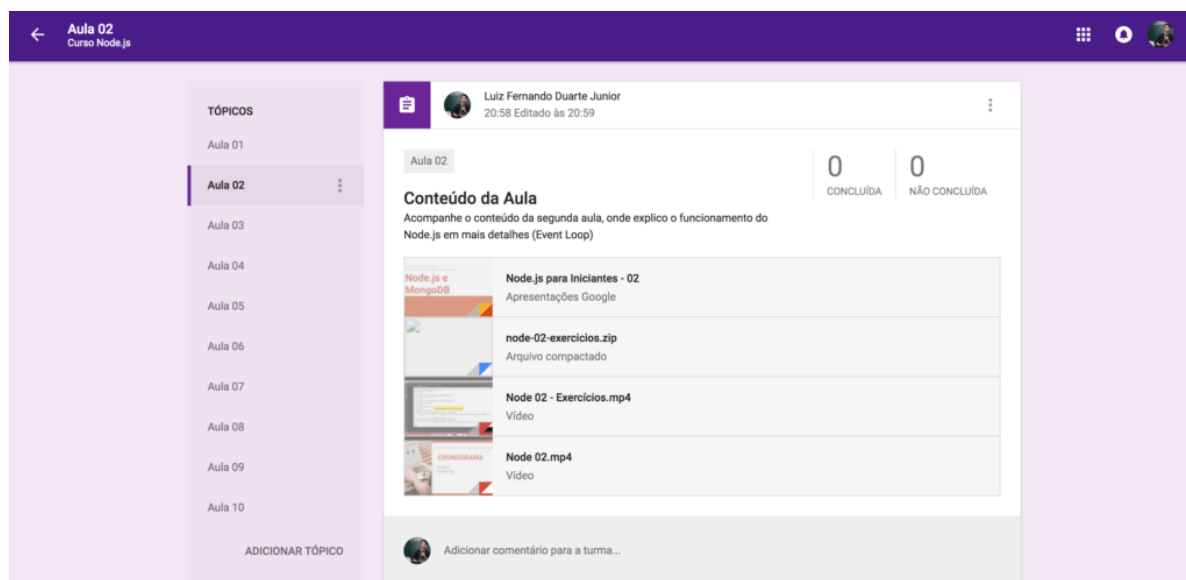


**Criando apps para  
empresas com  
Android**



**Scrum e Métodos  
Ágeis: Um Guia Prático**

**Meus Cursos:**



**Node.js e MongoDB**