WebAPIs em Node.js

2ª edição

LuizTools

SUMÁRIO

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), nove certificações em Métodos Ágeis de desenvolvimento de software por diferentes certificadoras (PSM-I, PSD-I, PACC-AIB, IPOF, ISMF, IKMF, CLF, DEPC, SFPC) e três certificações de coach profissional pelo IBC (Professional & Self Coach, Life Coach e Leader Coach).

Atuando na área de TI desde 2006, na maior parte do tempo como desenvolvedor, 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 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, como consultor em sua própria empresa, a DLZ Tecnologia e é autor do blog **www.luiztools.com.br**, onde escreve semanalmente sobre métodos ágeis e desenvolvimento de software, bem como mantenedor do canal **LuizTools**, com o mesmo propósito.

Entre em contato, o autor está sempre disposto a ouvir e ajudar seus leitores.

MEUS CURSOS

Curso online
NODE.JS e MONGODB

SAIBA MAIS...

Curso online
Scrum e métodos Ágeis

SAIBA MAIS...

Curso online
Jira

SAIBA MAIS...

Curso online
Web Full Stack JavaScript

SAIBA MAIS...

Curso online
React Native com Firebase

SAIBA MAIS...

Conheça todos os meus cursos

MEUS LIVROS

Programação Web com Node.js

SAIBA MAIS...

Programação Web com Node.js

SAIBA MAIS...

Node.js e Microservices

SAIBA MAIS...

MongoDB para Iniciantes

SAIBA MAIS...

Scrum e Métodos Ágeis

SAIBA MAIS...

Agile Coaching

SAIBA MAIS...

Criando apps para empresas com Android

SAIBA MAIS...

Java para iniciantes

SAIBA MAIS...

Conheça todos os meus livros

Aproveita e segue nas redes sociais:

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 meus livros e meus cursos, citados na seção anterior, que são muito mais completos neste sentido.

INTRODUÇÃO AO NODE.JS

1

“

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 webservers sem se preocupar com threading.

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 consequentemente 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%.

Ecossistema 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 2020, 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.

Fonte: **https://insights.stackoverflow.com/survey/2020#most-popular-technologies**

Outro ranking interessante, também de 2020, mostra a porcentagem de repositórios, por linguagem, hospedados no GitHub. Ou seja, quase um a cada cinco projetos hospedados no GitHub são projetos JavaScript.

# Ranking	Programming Language	Percentage (Change)	Trend
1	JavaScript	18.789% (-1.133%)	
2	Python	16.108% (-1.804%)	
3	Java	10.731% (+0.250%)	
4	Go	8.922% (+1.006%)	
5	C++	7.636% (+0.383%)	

Fonte: **https://madnight.github.io/githut/#/pull_requests/2020/2**

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

Eu realmente acredito que você deveria usar também.

PRIMEIROS PASSOS

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

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 (LTS), eu gostaria que você baixasse a versão mais recente 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 2.1: disponível em https://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 2.2: disponível em https://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 2.3: disponível em https://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 2.4: disponível em https://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 2.5: disponível em https://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 2.6: disponível em https://www.luiztools.com.br/livro-node-api-fontes
node /documents/index.js

```
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.

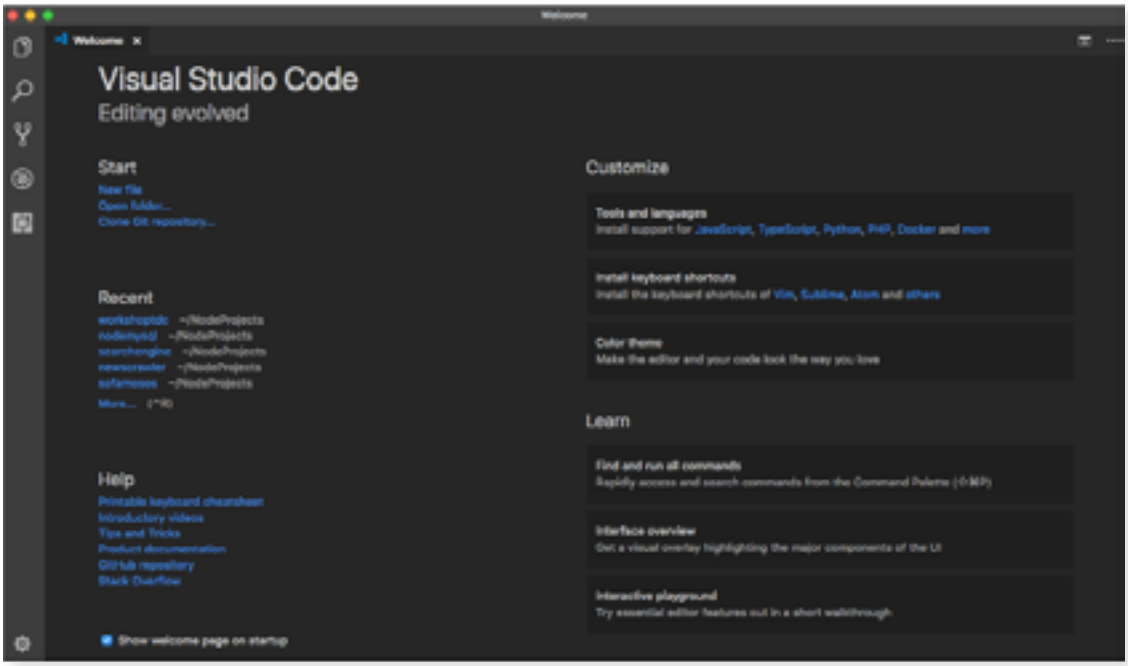
Outras excelentes ferramentas incluem o Visual Studio Community, Athom e o Sublime, sendo que o primeiro eu já utilizava quando era programador .NET e usei durante o início dos meus aprendizados com Node. No entanto, não faz sentido usar uma ferramenta tão pesada para uma plataforma tão leve, mesmo ela sendo gratuita na versão Community. O segundo (Athom) 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 ovperacional. 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 2.7: disponível em https://www.luiztools.com.br/livro-node-api-fontes

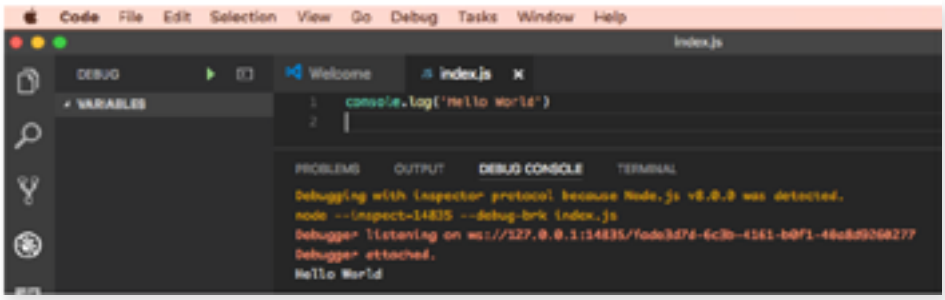
```
1 console.log('Hello World')
```

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).

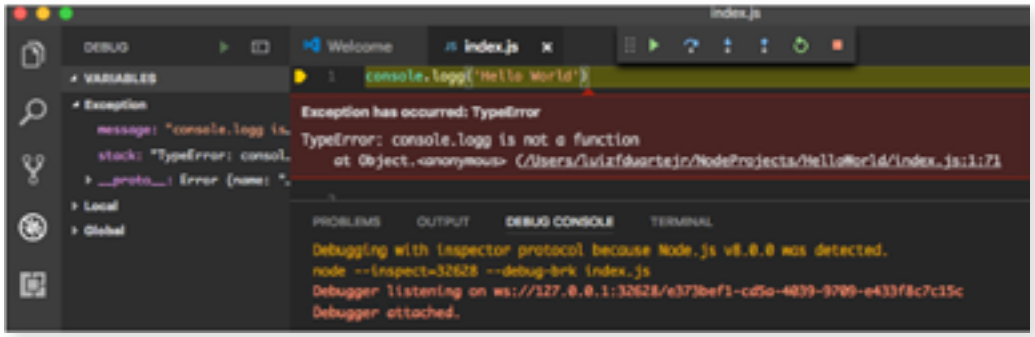
Salve o arquivo escolhendo File > Save ou usando o atalho Ctrl + S. Minha sugestão é que salve dentro de uma pasta NodeProjects/HelloWorld para manter tudo organizado e com o nome de index.js. Geralmente o arquivo inicial de um programa Node.js se chama index, enquanto que a extensão '.js' é obrigatória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

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:

Baixe e extraia o Community Server (free) no link abaixo. Não é necessária instalação, mas lembre-se de onde extraiu os arquivos.

https://www.mongodb.com/try/download/community

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

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 Node/NPM instalado (fizemos agora a pouco), 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):

Código 2.8: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
1 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 2.9: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
1 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 2.10: disponível em https://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 2.11: disponível em https://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 2.12: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
var app = require('../app');
var debug = require('debug')('workshop:server');
var http = require('http');

var port = normalizePort(process.env.PORT ||
'3000');
app.set('port', port);

var server = http.createServer(app);

server.listen(port);
server.on('error', onError);
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 2.13: disponível em https://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 2.14: disponível em https://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 2.15: disponível em https://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 2.16: disponível em https://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 2.17: disponível em https://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 2.18: disponível em https://www.luiztools.com.br/livro-node-api-fontes

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

router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

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 2.19: disponível em https://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 2.20: disponível em https://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 2.21: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/
style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</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 2.22: disponível em https://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.

Experimente mudar este texto 'Express' para ver o que acontece com a aplicação.

MONGODB

3

> Truth can only be found in one place: the code.
>
> *- Robert C. Martin*

Há mais de 10 anos atrás, 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 poliglotas de persistência de dados, ou seja, diferentes linguagens e mecanismos para lidar com os dados das aplicações, dentre eles, o MongoDB.

Introdução ao MongoDB

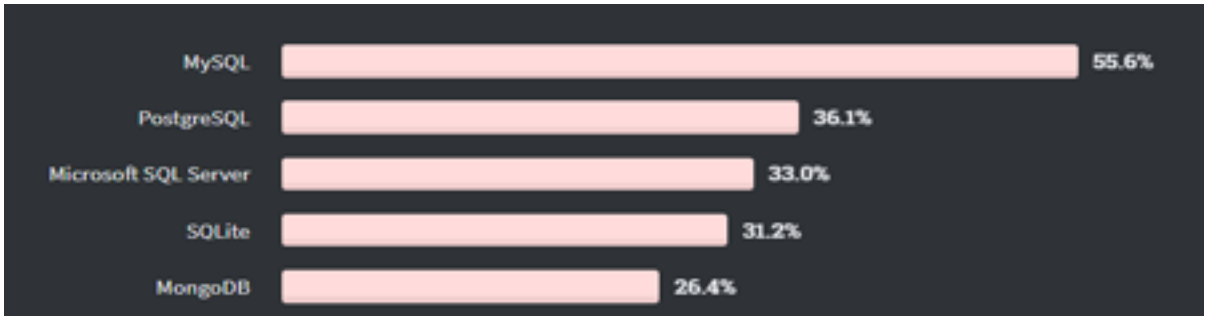
MongoDB é um banco de dados de código aberto, gratuito, de alta performance, sem esquemas e orientado a documentos lançado em fevereiro de 2009 pela empresa 10gen. Foi escrito na linguagem de programação C++ (o que o torna portável para diferentes sistemas operacionais) e seu desenvolvimento durou quase 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 2020 mostra.

Fonte: https://insights.stackoverflow.com/survey/2020#most-popular-technologies



Dentre todos os bancos não relacionais o MongoDB é o mais utilizado com ¼ 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.

Instalação e Testes

Diversos players de cloud computing fornecem versões de Mongo hospedadas e prontas para uso como **Umbler** e **Atlas**, 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 do Community Server (free) para o seu sistema operacional.

https://www.mongodb.com/try/download/community

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). Apenas dois nos interessam:

- mongod: inicializa o servidor de banco de dados;
- mongo: inicializa o cliente de banco de dados;

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 (essa pasta já deve estar criada).

Código 3.1: disponível em http://www.luiztools.com.br/livro-node-api-fontes

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

Isso irá iniciar o servidor do Mongo e irão correr uma série de comandos pela tela até parar e se tudo deu certo, sem nenhuma mensagem de erro. 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 3.2: disponível em http://www.luiztools.com.br/livro-node-api-fontes

```
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 3.3: disponível em http://www.luiztools.com.br/livro-node-api-fontes

```
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 3.4: disponível em http://www.luiztools.com.br/livro-node-api-fontes

```
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 3.5: disponível em http://www.luiztools.com.br/livro-node-api-fontes

```
show collections
```

Isso também não deve listar nada, mas não se importe com isso também.
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.
Find e Insert

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 3.6: disponível em http://www.luiztools.com.br/livro-node-api-fontes

```
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 3.7: disponível em http://www.luiztools.com.br/livro-node-api-fontes

```
1 db.customers.insert({ nome: "Luiz", idade: 32 })
```

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.

Outra opção é a função insertOne, que retorna também o _id do documento recém inserido.

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

Find com filtro

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 3.8: disponível em https://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 usa JavaScript que é case-sensitive (sensível a maiúsculas e minúsculas) ao contrário da linguagem SQL, 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".

Obviamente o find do MongoDB é muito mais poderoso que isso, mas temos de avançar para conhecer os demais comandos.

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 de atualizar um documento é chamando a função update na coleção. Esta função possui dois parâmetros obrigatórios:

- filtro para saber qual documento será atualizado;
- novo documento que substituirá o antigo;

Como em:

Código 3.9: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
1 db.customers.update({nome: "Luiz"},
2 {nome: "Luiz", idade: 32, uf: "RS"})
```

Neste comando estou dizendo para substituir (atualizar completamente) somente o primeiro documento cujo nome seja (literalmente) Luiz pelo objeto presente no segundo parâmetro.

Como resultado você deve ter um nModified igual a 1, mostrando que um documento foi atualizado.
Neste comando estou dizendo para substituir (atualizar completamente) somente o primeiro documento cujo nome seja (literalmente) Luiz pelo objeto presente no segundo parâmetro.

Como resultado você deve ter um **nModified** igual a 1, mostrando que um documento foi atualizado.

E aqui vai um cuidado que você deve ter: esta função de update substitui completamente o documento filtrado com o JSON passado como segundo argumento. Ou seja, ela exige que você passe o documento completo a ser atualizado no segundo parâmetro, pois ele substituirá o original!

Existem formas mais avançadas de usar o update, mas por ora, essa já nos atende.

Delete

Pra encerrar o nosso conjunto de comandos mais elementares do MongoDB falta o delete.
Existe uma função deleteOne, o que a essa altura do campeonato você já deve imaginar o que ela faz. Além disso, assim como o find e o update, o primeiro parâmetro do delete é o filtro que vai definir qual documento vai ser deletado e todos os filtros normais do find são aplicáveis.

Sendo assim, de maneira bem direta:

Código 3.10: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
1 db.customers.deleteOne({nome: "Luiz"})
```

Vai excluir o primeiro cliente que encontrar cujo nome seja igual a "Luiz". Note que para que o deleteOne seja mais preciso, recomenda-se que o filtro utilizado seja um índice único, como o _id, por exemplo. 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.

WEB APIS

4

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

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.

Existem diversas formas de se criar uma web API com Node.js. Mesmo definindo que vamos usar Express, ainda assim existem diferentes maneiras, das mais básicas às mais profissionais. Existe um outro livro meu chamado Node.js e Microservices completamente destinado à construção de web APIs profissionais, de escala corporativa.

Aqui nós iremos pelo caminho mais fácil. Mais pra frente, você se sentindo mais confiante, pode buscar jeitos mais profissionais e complexos.

Vamos começar voltando ao nosso projeto criado com Express anteriormente (workshop). Usaremos ele como base para nossa Web API.

MongoDB Driver

> **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.*

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, cuja dependência você deve instalar usando o comando abaixo, uma vez dentro da pasta workshop.

Código 4.1: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
npm install mongodb
```

Agora, entre na pasta workshop/routes e renomeie o arquivo users.js para customers.js. Esse arquivo vai ser a nossa web api de customers, voltaremos nele daqui a pouco. Poderíamos ter criado um novo arquivo, mas acho que vai ser mais simples assim.

Voltando à raiz do projeto, abra o arquivo app.js e procure onde ele faz referência ao antigo users.js, substituindo pelo novo customers.js. Também alterei o nome da variável usersRouter para customersRouter, para ficar mais coerente.

São dois trechos do app.js que serão alterados, o do início e do fim do trecho a seguir.

Código 4.2: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
var indexRouter = require('./routes/index');
var customersRouter = require('./routes/customers');

//...

app.use('/', indexRouter);
app.use('/customers', customersRouter);
```

Essa configuração vai garantir que, todas as chamadas à API de customers será feita através do caminho /customers do nosso servidor.

Agora, volte ao workshop/routes/customers.js pois vamos realizar a conexão com nosso banco de dados. Adicione estas linhas nele, bem no início do arquivo:

Código 4.3: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
const {MongoClient} = require("mongodb");
async function connect(){
  if(global.db) return global.db;
    const conn = await MongoClient.
    connect("mongodb://localhost:27017/", {
useUnifiedTopology: true });
    if(!conn) return new Error("Can't connect");
  global.db = await conn.db("workshop");
  return global.db;
}
```

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 de mesmo nome. Usaremos esta variável na função connect que precisará ser criada.

Antes de qualquer coisa, eu verifico se existe uma variável global chamada db. Se existir, isso quer dizer que já temos uma conexão estabelecida e retornamos ela. Fim.

Agora, se é a primeira conexão, chamamos a função MongoClient. 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:

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

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

Existem duas palavras reservadas aí que podem lhe causar alguma confusão e elas são async e await. Async diz que nossa função é assíncrona e é um pré-requisito para conseguirmos usar await na sequência. O Await bloqueia a execução daquela linha até que termine o processamento. 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 padrão, por isso usamos o await para dizer a ele esperar pelo banco, pois não podemos avançar sem essa conexão.

Essa conexão retorna um objeto através do qual nós podemos selecionar o banco de dados que queremos, e vamos armazenar ele em uma variável global, para que não tenhamos de ficar abrindo múltiplas conexões sem necessidade no banco de dados.

```
1  global.db = await conn.db("workshop");
2   return global.db;
```

Esse processo será necessário em vários pontos da nossa aplicação, por isso que resolvi criar uma função encapsulando tudo, para aumentar o reuso de código e facilitar possíveis manutenções futuras.

Agora, vamos configurar um retorno simples para a nossa rota, neste mesmo arquivo (substitua o código antigo da rota):

Código 4.4: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
1  router.get('/', function(req, res, next) {
2   res.json(connect());
3 });
```

Aqui criei uma rota GET default que apenas retorna um JSON com o conteúdo da conexão. Se tudo ocorrer bem, será um objeto JSON vazio, sem erros. Preste atenção ao código do router.get, é assim que vamos definir as nossas outras rotas mais tarde.

Com isso, já podemos iniciar a nossa web API e ver se ela está funcionando, usando o comando NPM start uma vez dentro da pasta raiz do projeto.

Código 4.5: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
npm start
```

O resultado no navegador, acessando localhost:3000 deve ser um objeto JSOn vazio.

Como não temos views em Web APIs, 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 customers.js, a começar por aquele router.get que já deixei lá por padrão.

Assim, vamos começar com uma operação muito simples: listar todos os clientes do banco de dados. Para isso, modifique o conteúdo do router.get para que chame a função connect e use essa conexão para ir no banco de dados buscar os clientes, como abaixo:

Código 4.6: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
router.get('/', async function(req, res, next) {
    try{
      const db = await connect();
      res.json(await db.collection("customers").
find().toArray());
    }
    catch(ex){
```

```
        console.log(ex);
        res.status(400).json({erro: `${ex}`});
    }
```

Aqui nós chamamos o connect com um await, para aguardar o término da sua execução. Na sequência usamos a função collection para selecionar a coleção de customers, seguido da função find para nos trazer tudo e por úlltimo a toArray para converter o resultado para um array de clientes.

Como essa operação também deve demorar alguns milisegundos (dependendo do volume de clientes cadastrados), usamos o await aqui também para segurar o Node.js.

Note que em caso de erro (falha de conexão com o banco, por exemplo) nossa vAPI vai retornar um código 400 com um JSON informando a causa do erro.

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

O resultado da consulta vai ser devolvido em formato JSON usando res.json. 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 npm start novamente.

Agora abra seu navegador, acesse http://localhost:3000/customers 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:

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

Podemos modificar a nossa rota GET para que ela receba por parâmetro um id opcionalmente. Neste caso, ela deverá retornar somente um cliente ao invés de todos.

Antes de mexer na rota, precisaremos de mais um objeto, então volte ao início do arquivo customer.js para editar a nossa chamada ao módulo mongodb.

Código 4.7: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
const {MongoClient, ObjectId} = require("mongodb");
```

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. Você verá isso na prática no próximo trecho de código.

Com este código pronto, agora podemos fazer alguns ajustes na router.get:

Código 4.8: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
router.get('/:id?', async function(req, res, next)
{
    try{
      const db = await connect();
      if(req.params.id)
        res.json(await db.collection("customers").
findOne({_id: new ObjectId(req.params.id)}));
      else
        res.json(await db.collection("customers").
find().toArray());
```

▲

```
11  }
12      catch(ex){
13          console.log(ex);
14          res.status(400).json({erro: `${ex}`});
15      }
16   })
```

Existem poucas diferenças em relação à outra rota, como o parâmetro :id no caminho da URL e a chamada à função findOne quando veio o id. Note que usei uma interrogação logo após o :id, isso quer dizer que ele é opcional. Para acessar essa informação que veio na URL, basta usar req.params.<nome_do_parametro>.

Para testar, pegue um dos _ids dos clientes listados no teste anterior e experimente acessar uma URL localhost:3000/customers/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 filtros 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.

Para isso, vamos criar uma nova rota no customers.js com router.post, uma vez que POST é o verbo HTTP usado para cadastrar coisas:

Código 4.9: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
router.post('/', async function(req, res, next){
  try{
    const customer = req.body;
    const db = await connect();
    res.json(await db.collection("customers").
insert(customer));
  }
  catch(ex){
    console.log(ex);
    res.status(400).json({erro: `${ex}`});
  }
})
```

Note que este código já começa diferente dos demais, com router.post ao invés de router.get, indicando que está rota tratará POST no endpoint /customers. 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.

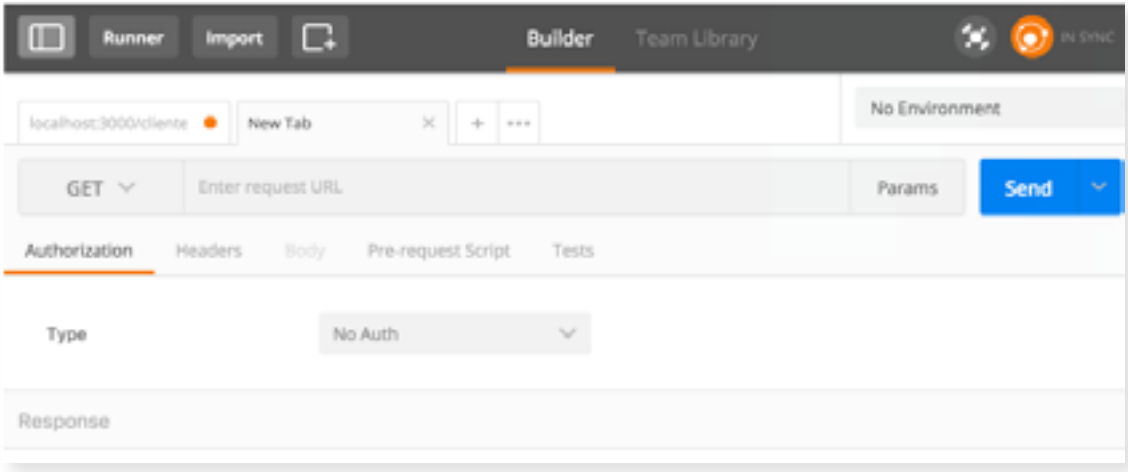
Mas e agora, como vamos testar um POST? Pela barra de endereço do navegador não dá, pois ela só faz GET.

Temos diversas alternativas no mercado, sendo que vou falar de uma aqui: 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:

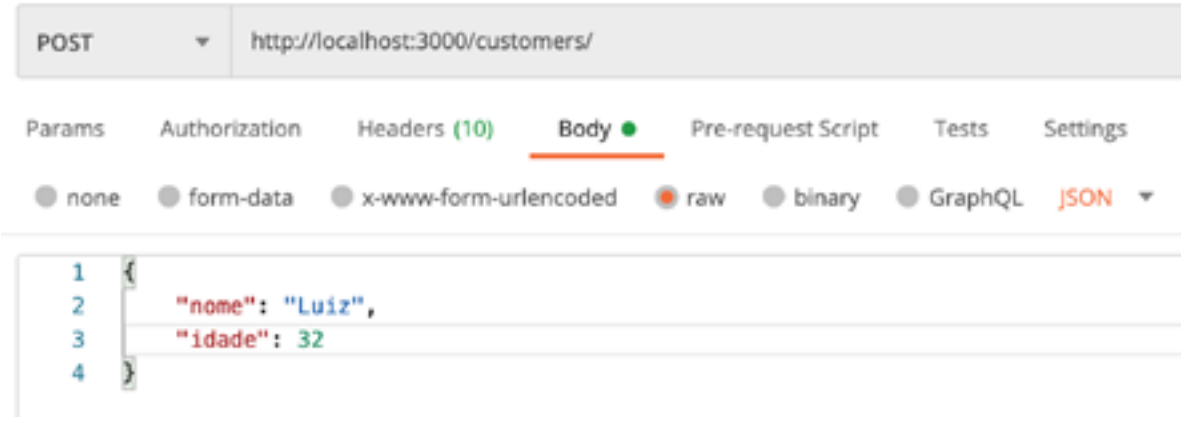


No primeiro select à esquerda você encontra os verbos HTTP, selecione POST nele.

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

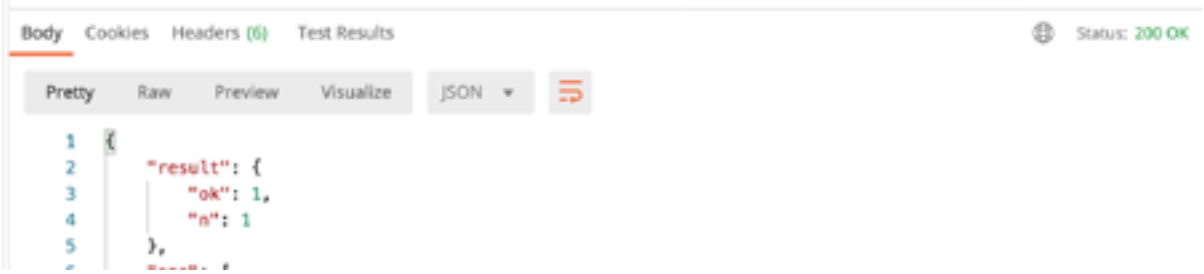
Logo abaixo temos algumas abas. Vá na aba Headers (cabeçalhos) e adicione uma linha com a seguinte configuração: key = Content-Type e value = application/json. Isso diz ao POSTMAN que essa requisição está enviando dados no formato JSON em seu corpo (body).

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/customers em nosso navegador (ou construir uma requisição GET no POSTMAN) e veremos que nosso novo customer está lá!

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 rota no customers.js:

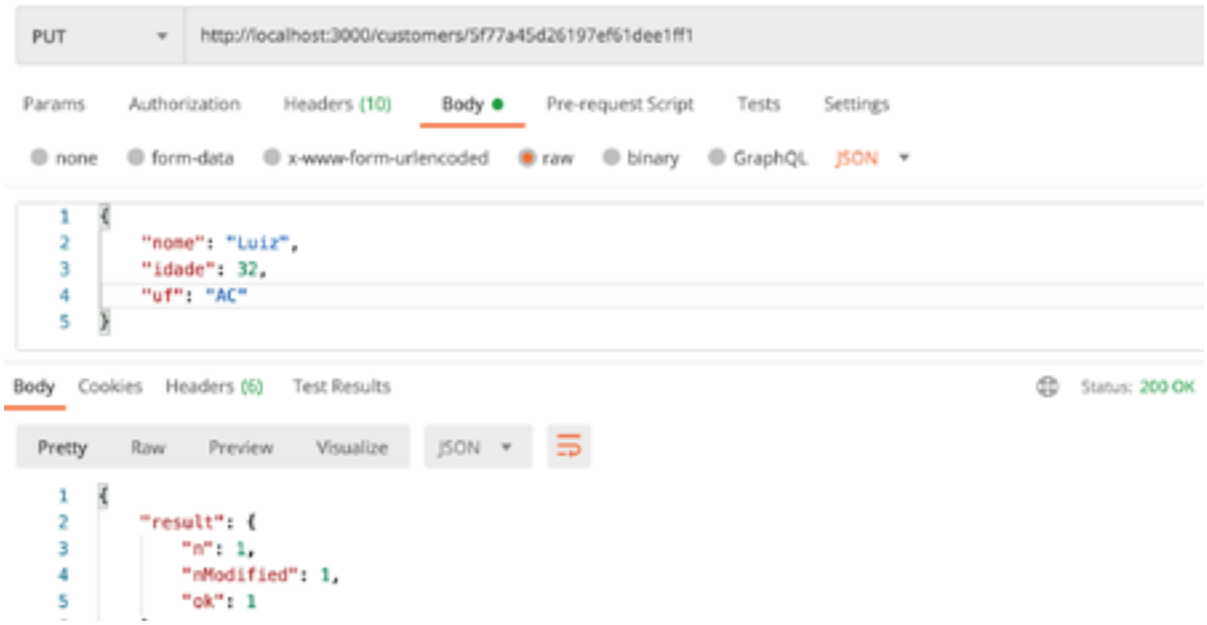
Código 4.10: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
router.put('/:id', async function(req, res, next){
  try{
    const customer = req.body;
    const db = await connect();
    res.json(await db.collection("customers").
```

```
6   update({_id: new ObjectId(req.params.id)},
7   customer));
8     }
9     catch(ex){
10      console.log(ex);
11      res.status(400).json({erro: `${ex}`});
12    }
13  })
```

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.

Update: check!

Excluindo clientes

Para encerrar a construção da nossa API REST, falta nosso delete que é muito simples e rápido de fazer: cria-se uma nova rota com o verbo DELETE.

Código 4.11: disponível em https://www.luiztools.com.br/livro-node-api-fontes

```
router.delete('/:id', async function(req, res,
next){
    try{
      const db = await connect();
      res.json(await db.collection("customers").
deleteOne({_id: new ObjectId(req.params.id)}));
    }
    catch(ex){
      console.log(ex);
      res.status(400).json({erro: `${ex}`});
    }
})
```

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.

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 que estou sempre disposto a melhorar.
Um abraço e até a próxima!

MEUS CURSOS

Curso online
NODE.JS e MONGODB

SAIBA MAIS...

Curso online
Scrum e métodos Ágeis

SAIBA MAIS...

Curso online
Jira

SAIBA MAIS...

Curso online
Web Full Stack JavaScript

SAIBA MAIS...

Curso online
React Native com Firebase

SAIBA MAIS...

Conheça todos os meus cursos

MEUS LIVROS

Programação Web com Node.js

SAIBA MAIS...

Programação Web com Node.js

SAIBA MAIS...

Node.js e Microservices

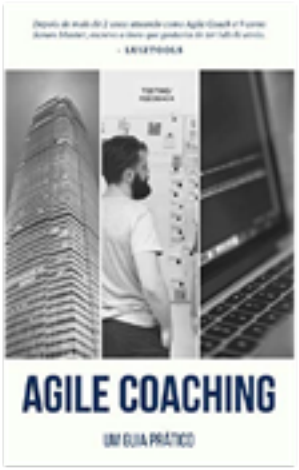
SAIBA MAIS...

MongoDB para Iniciantes

SAIBA MAIS...

Scrum e Métodos Ágeis

SAIBA MAIS...



Agile Coaching

SAIBA MAIS...

Criando apps para empresas com Android

SAIBA MAIS...

Java para iniciantes

SAIBA MAIS...

Conheça todos os meus livros

Aproveita e segue nas redes sociais: