



PARA
INICIANTE

umbler

Sumário

1. INTRODUÇÃO	3
2. A PLATAFORMA NODE.JS	6
História do Node.js	7
Características do Node.js	8
Node.js não é uma linguagem de programação	9
Node.js não é um framework Javascript	9
3. CONFIGURAÇÃO E PRIMEIROS PASSOS	11
Node.js	12
Visual Studio Code	16
MongoDB	20
4. CRIANDO UMA APLICAÇÃO WEB	21
Routes e views	28
Event Loop	36
Task/Event/Message Queue	40
5. PERSISTINDO DADOS	41
Quando devo usar MongoDB?	43
Configurando o banco	44
Node.js + MongoDB	47
MongoDB Driver	49
Listando os clientes	53
Cadastrando novos clientes	56
Excluindo um cliente	58
6. SEGUINDO EM FRENTE	61

INTRODUÇÃO



Talk is cheap.
Show me the code.

- Linus Torvalds



Na época da escola, costumava montar grupos de estudos para ajudar os colegas que tinham mais dificuldade em algumas matérias. Como sempre fui nerd, ficar na escola no turno inverso, ensinando os colegas, era mais um hobby do que uma obrigação. Mas, jamais havia pensado em me tornar professor.

No entanto, em 2010 (quando estava me formando em Ciência da Computação), a empresa em que trabalhava na época, estava enfrentando enormes dificuldades para contratar profissionais para o seu time. Para ajudar a empresa, comecei a ministrar cursos de extensão em dezenas de universidades gaúchas sobre tecnologias que utilizávamos, como ASP.NET, C# e SQL Server. Formava turmas aos sábados, dezenas de alunos participavam e, em um mês, saíam uns poucos “formados” – sendo que, destes, contratávamos apenas os melhores de cada turma (geralmente apenas um).

Em 2013, virei professor regular de uma grande rede de ensino gaúcha, justamente na semana em que me formei na pós-graduação de Computação Móvel. Desde então nunca mais parei de ensinar, seja como professor regular do ensino superior, blogueiro de desenvolvimento ou até mesmo como evangelista técnico – meu papel atualmente na Umblar.

Ser um programador não é fácil. Trabalho nessa área há onze anos (sim, além de dar aula, trabalho também!), tive a oportunidade de trabalhar com várias tecnologias, em várias empresas, em dezenas de projetos de todos os tamanhos. Inclusive, tive minha própria startup. Já tive de aprender diversas tecnologias “do zero” e aplicá-las rapidamente em projetos de missão crítica para dezenas de milhares de clientes (às vezes, centenas de milhares). E, por mais que sempre me considerasse um autodidata, tudo ficava mais fácil quando tinha um bom professor ou, ao menos, um bom material de apoio.

O que é curioso é que quando ainda estamos “aprendendo a programar” (como se isso acabasse um dia!), olhamos para os professores como se eles tivessem nascido sabendo fazer tudo aquilo. Como se tivessem feito apenas escolhas certas de tecnologias e empresas durante toda sua carreira. Como se fossem algum tipo de herói.

Ledo engano!

Vivemos as mesmas inseguranças que os calouros. Devo aprender a linguagem X ou Y? É melhor o framework W ou Z? Será que a tecnologia K vai aguentar a demanda do meu sistema?

Quem não tem esse tipo de dúvida no dia a dia é porque parou no tempo, não evolui mais e só programa usando legado. Faça um favor a nós dois: ignore-os. Deixe-os acumuladndo teias de aranha.

Hoje, trago um apanhado do que acredito que todo iniciante em Node.js deve saber para entender de fato essa plataforma, saber quando e como usá-la e etc. Para que, após ler e fazer todos os exercícios deste e-book, um programador, mesmo que iniciante, seja capaz de responder perguntas como essa quando a conversa envolver Node.js.

Quando batem palmas em minhas palestras e cursos, costumo dizer que não há necessidade, uma vez que naquela sala somos todos “devs”. E esse e-book é isso: um material de “dev para dev”, sem cerimônias.

Espero que goste.

Talk is cheap. Show me the code.

Um abraço e sucesso.



Luiz Duarte
Dev Evangelist
Umbler

A PLATAFORMA NODE.JS



A language that doesn't affect the way you think about programming is not worth knowing.

- Alan J. Perlis



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 possam ser desenvolvidas 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.



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 na engine JavaScript V8 do Google, um event 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 para os 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. 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 exista 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.



CONFIGURAÇÃO E PRIMEIROS PASSOS



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

- Robert C. Martin



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. Eventualmente, você terá de buscar mais materiais do que esse singelo ebook - mas isso faz parte do jogo, não é mesmo?!

Portanto, vamos começar do princípio.

• • •

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:

<https://nodejs.org>

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

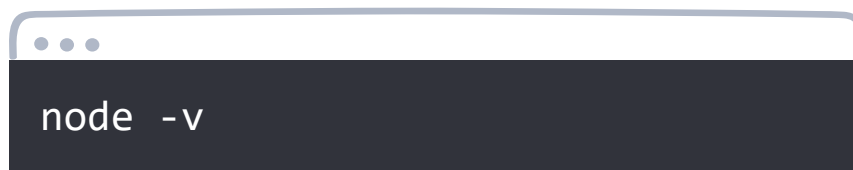
A instalação não requer nenhuma instrução especial, apenas avance cada uma das etapas e aceite o contrato de uso da plataforma.

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;

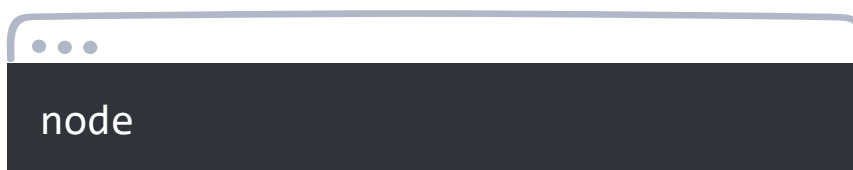
Deve ser observado que o Node.js não é um ambiente visual ou uma ferramenta integrada de desenvolvimento, embora mesmo assim seja possível o desenvolvimento de aplicações complexas apenas com o uso do mesmo, sem nenhuma ferramenta externa.

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:

A terminal window with a dark background and a light border. The command 'node -v' is entered in white text.

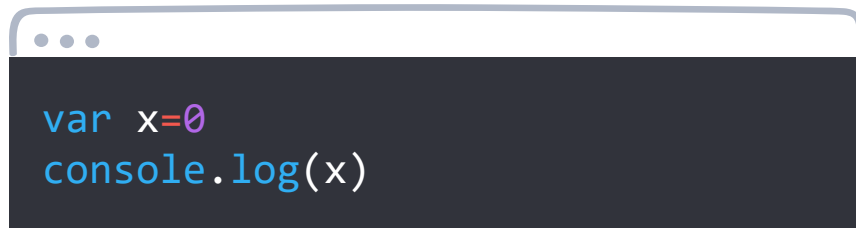
O resultado deve ser a versão do Node.js atualmente instalada na sua máquina, no meu caso, “v8.0.0”. Isso mostra que o Node está instalado na sua máquina e funcionando corretamente.

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:

A terminal window with a dark background and a light border. The command 'node' is entered in white text.

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

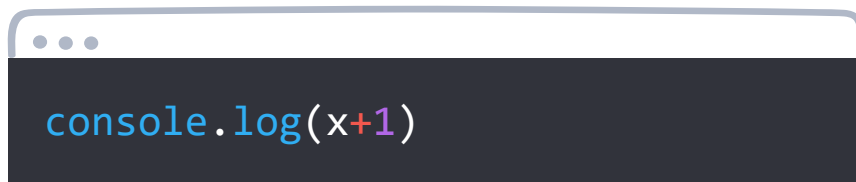
Apenas para fins ilustrativos, pois veremos “código de verdade” em mais detalhes neste e-book, inclua os seguintes códigos no terminal, pressionando Enter após digitar cada um:

A terminal window with a dark background and light blue text. It shows two lines of JavaScript code: `var x=0` and `console.log(x)`.

```
var x=0
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?

A terminal window with a dark background and light blue text. It shows a single line of JavaScript code: `console.log(x+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 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:

```
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:

```
node /documents/index.js
```

Isto executará o programa contido no arquivo /documents/index.js usando o runtime do Node. Note que aqui 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. Inclusive, recomendo que você crie uma pasta NodeProjects ou simplesmente Projects na sua pasta de usuário para guardar os exemplos deste e-book 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 e-book 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 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), nunca usei, mas tive boas recomendações, já o terceiro (Sublime) usei e sinceramente não gosto, especialmente na versão free que fica o tempo todo pedindo para fazer upgrade pra versão paga (minha opinião).

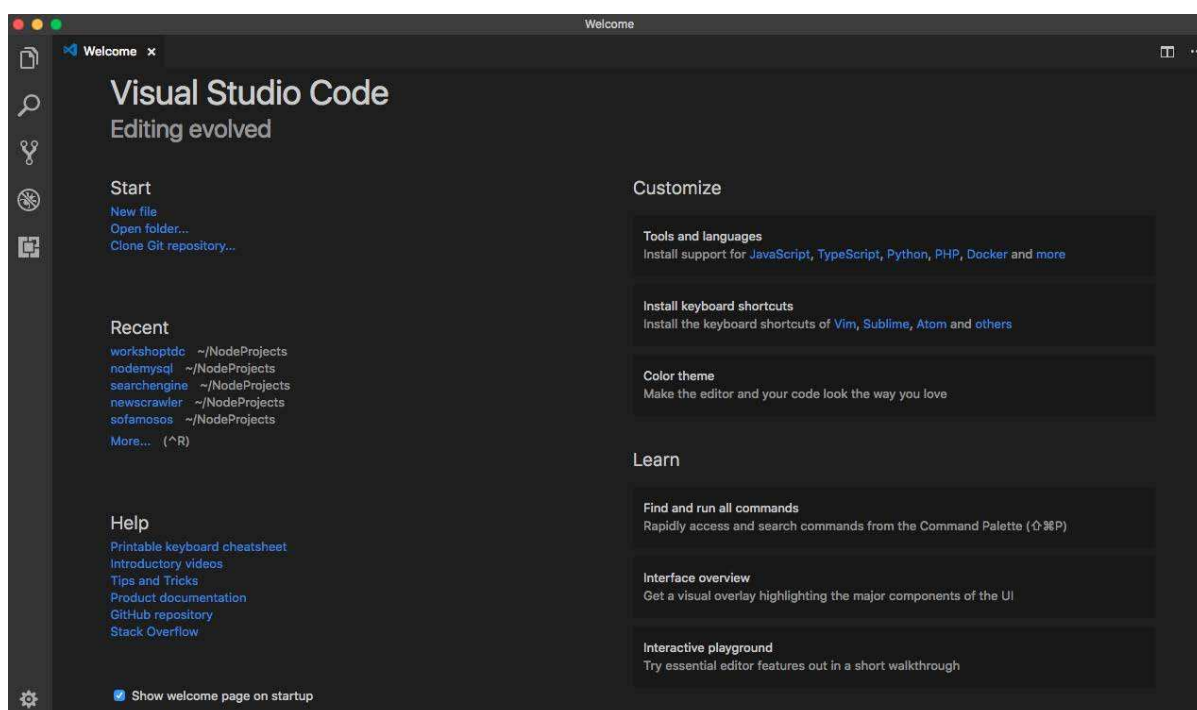


Para baixar e instalar o Visual Studio Code, acesse o seguinte link, no site oficial da ferramenta:

<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:

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

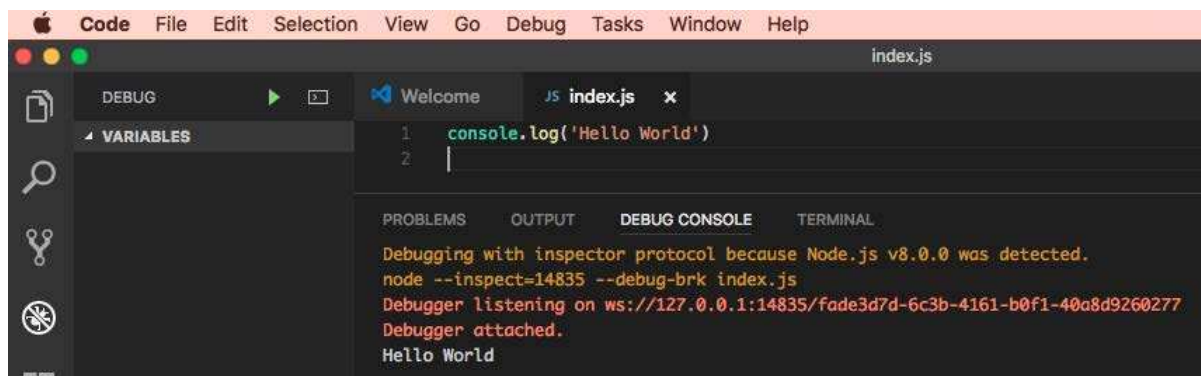
JavaScript é uma linguagem bem direta e sem muitos rodeios, permitindo que, com poucas linhas de código, façamos coisas incríveis. Ok, um olá mundo não é algo incrível, mas este mesmo exemplo em linguagens de programação como C e Java ocuparia muito mais linhas.

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.

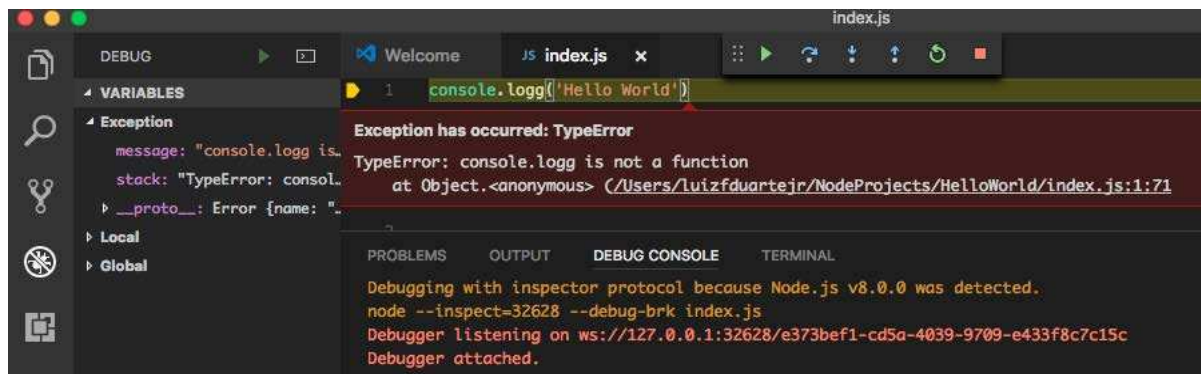
Salve o arquivo escolhendo File > Save ou usando o atalho Ctrl + S. Minha sugestão é que salve dentro de uma pasta NodeProjects/HelloWorld para manter tudo organizado e com o nome de index.js. Geralmente o arquivo inicial de um programa Node.js se chama index, enquanto que a extensão '.js' é obrigatória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

A screenshot of the Visual Studio Code interface in debug mode. The top menu bar includes 'Code', 'File', 'Edit', 'Selection', 'View', 'Go', 'Debug', 'Tasks', 'Window', and 'Help'. The editor window shows a file named 'index.js' with two lines of code: '1 console.log('Hello World');' and '2'. On the left, the 'DEBUG' sidebar is open, showing a 'VARIABLES' section. At the bottom, the 'DEBUG CONSOLE' tab is active, displaying the following output: 'Debugging with inspector protocol because Node.js v8.0.0 was detected.', 'node --inspect=14835 --debug-brk index.js', 'Debugger listening on ws://127.0.0.1:14835/fade3d7d-6c3b-4161-b0f1-40a8d9260277', 'Debugger attached.', and 'Hello World'. The 'TERMINAL' tab is also visible but empty.

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, 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!

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. No 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.



CRIANDO UMA APLICAÇÃO WEB



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

- **Max Kanat-Alexander**

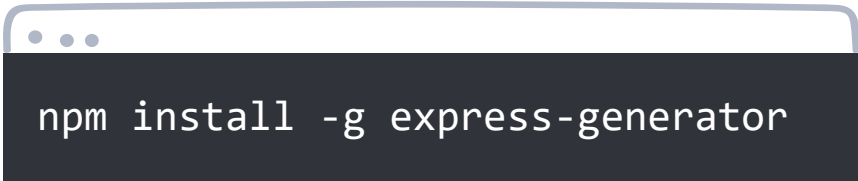


A primeira vez que vi Node.js, há uns dois anos (sim, eu não sei em que mundo estava antes disso), me empolguei com a ideia da plataforma, mas os primeiros exemplos e tutoriais não me ajudaram muito a gostar de mexer com a tecnologia.

Era tudo muito complicado e contraprodutivo. Não me agradava a ideia de ficar escrevendo servidores web na mão, tendo de ficar lidando com HTTP na “unha”, etc. Acabei logo deixando de lado essa fantástica tecnologia, até meados de 2016, quando conheci o ExpressJS.

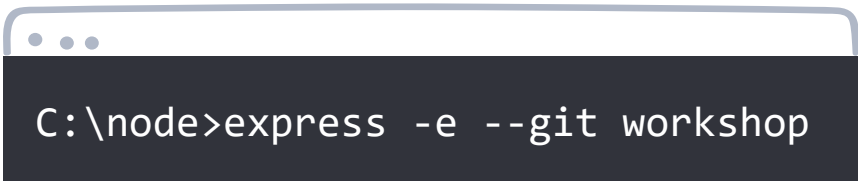
ExpressJS, ou apenas Express, é o web framework mais famoso atualmente para Node.js. Com ele você consegue criar aplicações e web APIs de forma rápida e fácil, sem ficar “reinventando a roda”. Além disso, ele se tornou um padrão de mercado, é muito rápido e muito simples de entender. É ele que usaremos para criar nossas primeiras aplicações web (embora use-o em todas aplicações que faço).

Uma vez com o npm instalado (ele instala junto do Node.js), podemos instalar um módulo que lhe será muito útil em diversos momentos deste e-book: rode o seguinte comando com permissão de administrador no terminal (‘sudo’ em sistemas Unix-like).



```
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 e que o nome desse novo projeto será workshop):

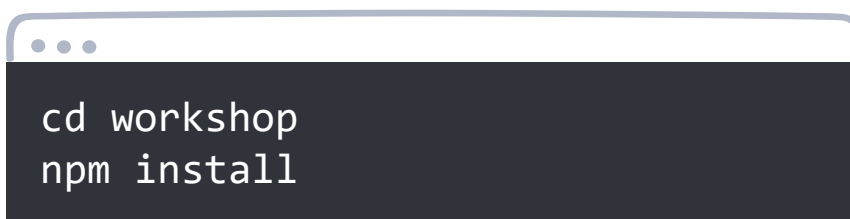


```
C:\node>express -e --git workshop
```

O “-e” é para usar a view-engine (motor de renderização) EJS, ao invés do tradicional Jade/Pug (que também é bom, mas não usa HTML tradicional, daí já viu né?). Já o “--git” deixa seu projeto preparado para versionamento com Git (o que, além de versionar seu código, lhe permite fazer [deploy na Umblar](#) muito facilmente).

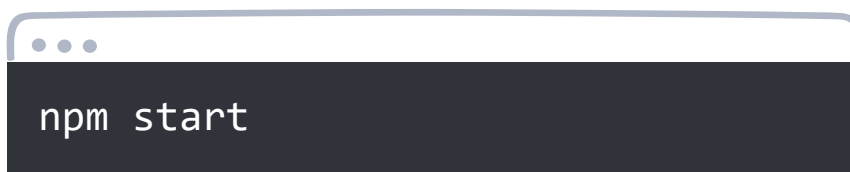
Aperte Enter e o projeto será criado (talvez ele peça uma confirmação, apenas digite ‘y’ e confirme).

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

A terminal window with a dark background and light text. It shows two lines of code: 'cd workshop' and 'npm install'.

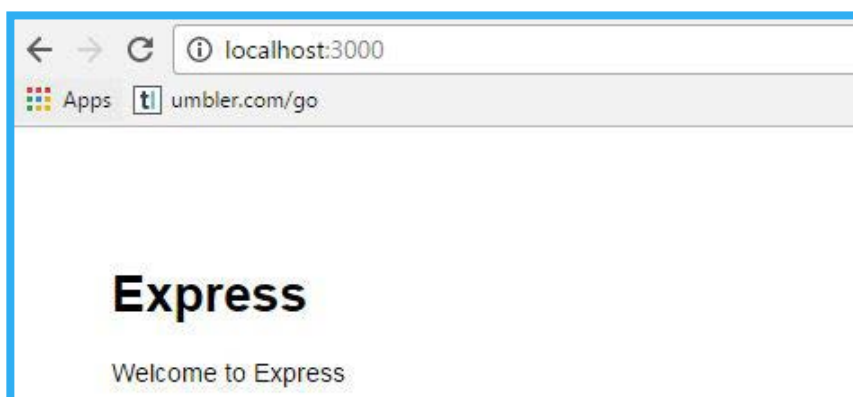
```
cd workshop  
npm install
```

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

A terminal window with a dark background and light text. It shows one line of code: 'npm start'.

```
npm start
```

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



Antes de sair criando nossas próprias aplicações, vamos entender o framework Express.

Entre na pasta bin do projeto 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:

```
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 predefinida. 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:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var index = require('./routes/index');
var users = require('./routes/users');
```

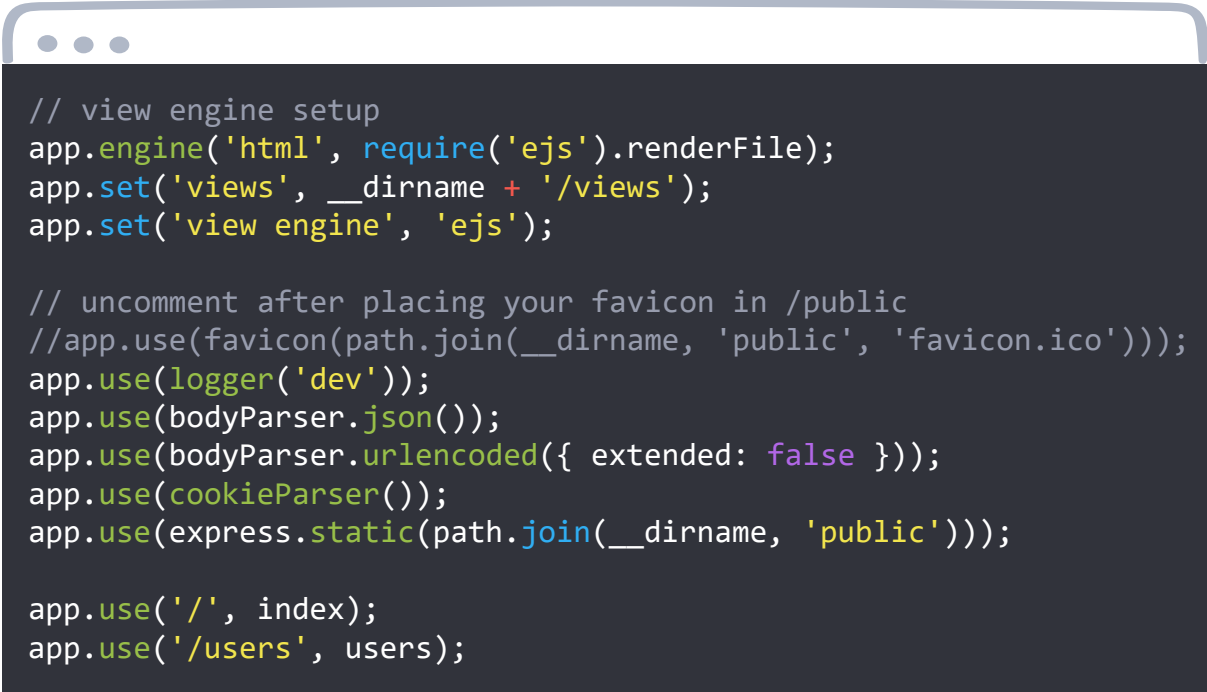
Isto define um monte de variáveis JavaScript e referencia elas a alguns pacotes, dependências, funcionalidades do Node e rotas. Rotas direcionam o tráfego e contém também alguma lógica de programação (embora você consiga, se quiser, usar padrões mais “puros” como MVC se desejar). Quando criamos o projeto 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:



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

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



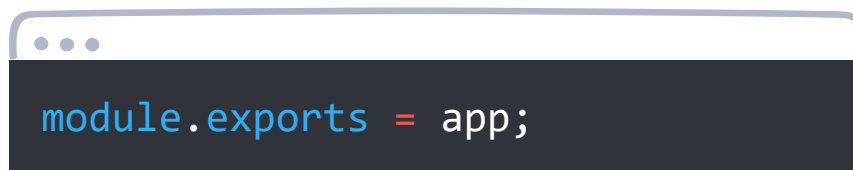
```
// view engine setup
app.engine('html', require('ejs').renderFile);
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', index);
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\workshoptdc\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.



```
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, assim como a Umblar.



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:

```
// códigos...
var index = require('./routes/index');
var users = require('./routes/users');

// mais códigos...

app.use('/', index);
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.

```
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, 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.

A função anônima passada como segundo parâmetro do `router.get` será disparada toda vez que chegar um GET na raiz da aplicação. Esse comportamento se chama `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 ebook.

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

req: contém informações da requisição HTTP que disparou essa `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 não será explorada neste material, que é para iniciantes.

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):

A code editor window with a dark background and light blue borders. It contains a single line of JavaScript code: `res.render('index', { title: 'Express' });`. The code is written in a monospace font, with `res` in green, `render` in yellow, and the string literals in white. The window has three small circles in the top left corner, representing a standard OS window design.

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

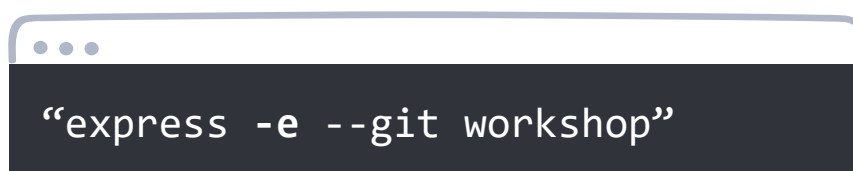
Aqui dizemos que deve ser renderizado na resposta (`res.render`) a view 'index' com o model entre chaves (`{}`). Se você já estudou o padrão MVC antes, deve estar se sentindo em casa e entendendo que o router é o controller que liga o model com a view.

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 '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 ebook usarei EJS do início ao fim para não confundir ninguém.

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

```
// view engine setup
app.engine('html', require('ejs').renderFile);
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
```

Aqui dissemos que vamos renderizar HTML 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:

```
<!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>
```


Espero que já conheça HTML, conteúdo-base para se trabalhar com programação web. Além das tags HTML normais, temos as tags `<% %>` que 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'. Hmmmm, onde que vimos essa variável title antes?

Dentro do routes/index.js!!!

```
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, proponho um desafio: crie uma nova view que deve ser exibida quando o usuário acessar '/new' no navegador. Como o objetivo aqui não é ensinar HTML, use o abaixo (ele vai fazer mais sentido no próximo capítulo) para criar a view new.ejs:

```
<!DOCTYPE html>
<html>
<head>
  <title>CRUD de Clientes</title>
  <link rel="stylesheet" href="/stylesheets/style.css" />
</head>
<body>
  <h1><%= title %></h1>
  <p>Preencha os dados abaixo para salvar o cliente.</p>
  <form action="/new" method="POST">
    <p>
      <label>Nome: <input type="text" name="nome" /></label>
    </p>
    <p>
      <label>Idade: <input type="number" name="idade" /></label>
    </p>
    <p>
      <label>UF:
        <select name="uf">
          <option>RS</option>
          <option>SC</option>
          <option>PR</option>
          <!-- coloque os estados que quiser -->
        </select>
      </label>
    </p>
    <p>
      <a href="/">Cancelar</a> | <input type="submit" value="Salvar" />
    </p>
  </form>
</body>
</html>
```

Se não consegue pensar no passo-a-passo necessário sozinho para fazer essa view funcionar, 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):

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

router.get('/', function(req, res, next) {
  res.render('new', { title: 'Novo Cadastro' });
});

module.exports = router;
```

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

```
var new = require('./routes/new');
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.

Para testar, basta mandar executar sua aplicação com 'npm start' e depois acessar localhost:3000/new no seu navegador, que deve exibir algo parecido com a imagem abaixo:



A screenshot of a web browser window. The address bar shows 'localhost:3000/new'. The page title is 'Cadastro de Cliente'. Below the title, it says 'Preencha os dados abaixo para salvar o cliente.' There are three input fields: 'Nome:' followed by a text input, 'Idade:' followed by a text input, and 'UF:' followed by a dropdown menu showing 'RS'. At the bottom, there are two buttons: 'Cancelar' (a link) and 'Salvar' (a button).

...

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](#).

A maioria dos sistemas operacionais 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. Ao definir x como 0 (o que pode parecer algo inútil, mas na verdade é extremamente útil), forçamos 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` (veremos estes dois últimos 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.

```
//coloque todo o conteúdo abaixo dentro de um arquivo index.js
//rode o comando 'npm init' na mesma pasta do index.js e apenas aperte Enter para tudo
//rode os comandos 'npm install -S express fs' para instalar as dependências
//use o comando 'node index' na pasta do index.js para iniciar esse programa
const express = require('express')
const fs = require('fs') //fs é o módulo file-system, para ler arquivos
const app = express()

app.get('/', processRequest)

function processRequest (request, response) {
  readText(request, response)
  console.log('requisição terminou')
}

function readText (request, response) {
  //salve um arquivo teste.txt junto a esse arquivo com qualquer coisa dentro
  fs.readFile('teste.txt', function(err, data) {
    if (err) {
      console.log('erro na leitura')
      return response.status(500).send('Erro ao ler o arquivo.')
    }
    response.write(data)
    response.end();
    console.log('leu arquivo')
  });

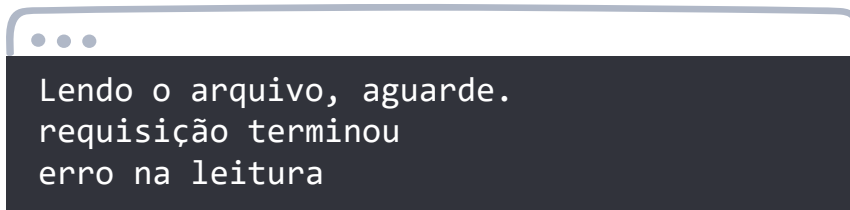
  console.log('Lendo o arquivo, aguarde.')
}

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.

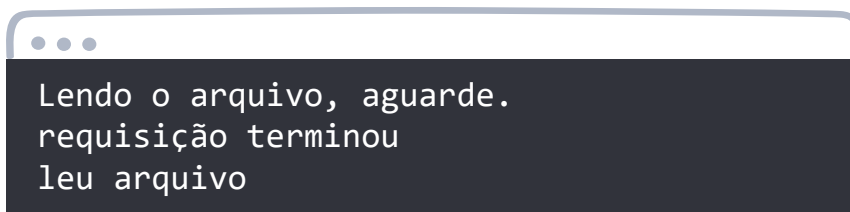
O que será impresso quando uma requisição é enviada para localhost:3000?

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.

PERSISTINDO DADOS



Not all roots are buried down in the ground some are at the top of the tree.a minor detail.

- Jinvirle



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 dois anos, tendo iniciado em 2007.

Por ser orientado a documentos JSON (armazenados em modo binário, apelidado 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.

Bancos não-relacionais document-based (que armazenam seus dados em documentos) são os mais comuns dentre os bancos NoSQL ou não-relacionais, 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.



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" ]
}
```

← field: value
← field: value
← field: value
← field: value

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.

Configurando o banco

Diversos players de cloud computing fornecem versões de Mongo hospedadas e prontas para uso, como a própria [Umbler](#). Para não nos demorarmos com questões de execução e configuração de servidor, parto do pressuposto aqui que você possui uma conta na Umbler com um site Node.js criado. **Se não tem, crie, você já sai com uma quantidade de créditos mais que suficiente para fazer os testes desse ebook.**

Na seção de bancos de dados do seu site, crie um banco MongoDB. Anote o host, usuário, senha, etc para usar depois.

Criando banco de dados...



Nome do Banco de Dados

Usuário

Senha

Tamanho

R\$ 6 ☒

R\$ 12 ☐

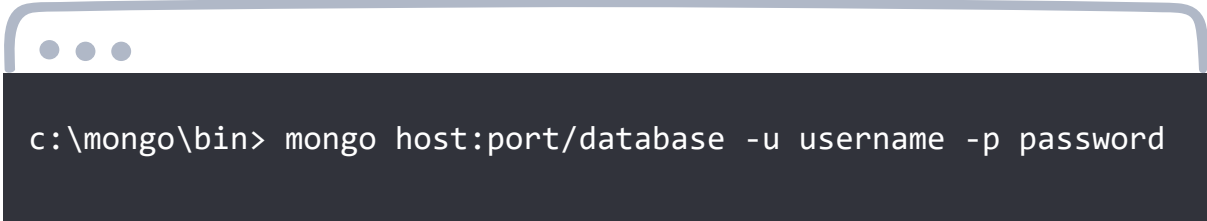
R\$ 24 ☐

R\$ 48 ☐

No capítulo de configuração de ambiente, disse para baixar e extrair o MongoDB para uma pasta, certo? 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);
- entre outros.

Agora abra um prompt de comando e navegue até a pasta bin do MongoDB e digite:



```
c:\mongo\bin> mongo host:port/database -u username -p password
```

Obviamente, troque cada uma das informações pelas que você cadastrou na Umbler durante a criação do seu MongoDB.

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.

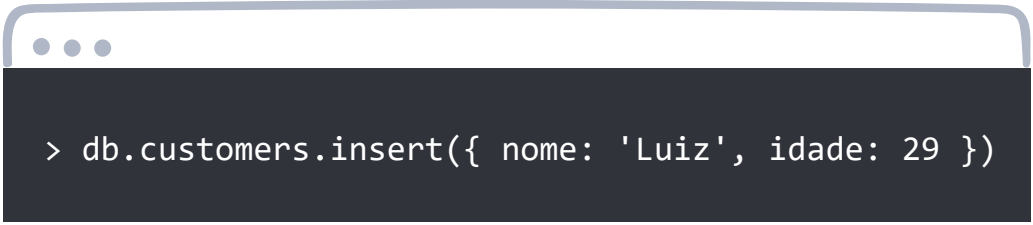
Ao contrário dos bancos relacionais, no MongoDB você não precisa construir a estrutura do seu banco previamente antes de sair utilizando ele. Tudo é criado conforme você for usando, o que não impede, é claro, que você planeje um pouco o que pretende fazer com o Mongo.

Seu banco nem mesmo existe ainda (mas não se preocupe com isso!) e mesmo assim você pode acessá-lo usando uma variável global db. Essa variável “db” representa o banco (que eu chamei de workshop) e 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:



```
> 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:



```
> 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.

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

...

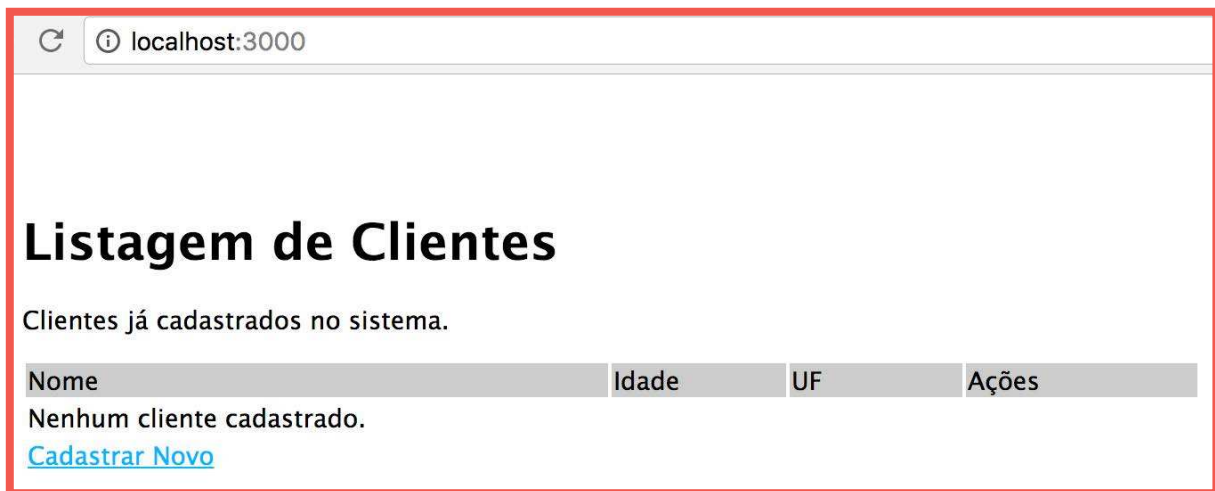
Node.js + MongoDB

Agora que entendemos como criar uma aplicação Node.js usando Express e, neste capítulo, como manipular o banco de dados MongoDB, é hora de juntar as duas pontas!

O que vamos fazer agora é continuar o nosso projeto chamado workshop, do capítulo anterior. Basicamente temos duas telas: uma tela inicial e outra de cadastro (a tela new). Essa tela inicial (views/index.ejs) deve ser modificada para se tornar uma tela de listagem, o que pode ser feito com o HTML abaixo:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cadastro de Clientes</title>
    <link rel="stylesheet" href="/stylesheets/style.css" />
  </head>
  <body>
    <h1>Listagem de Clientes</h1>
    <p>Clientes já cadastrados no sistema.</p>
    <table style="width:50%">
      <thead>
        <tr style="background-color: #CCC">
          <td style="width:50%">Nome</td>
          <td style="width:15%">Idade</td>
          <td style="width:15%">UF</td>
          <td style="width:20%">Ações</td>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td colspan="4">Nenhum cliente cadastrado.</td>
        </tr>
      </tbody>
      <tfoot>
        <tr>
          <td colspan="4">
            <a href="/new">Cadastrar Novo</a>
          </td>
        </tr>
      </tfoot>
    </table>
  </body>
</html>
```


O resultado você confere executando novamente o seu projeto e acessando a página inicial a partir do seu navegador:



Mais pra frente vamos tornar ambas telas, listagem e cadastro, dinâmicas.

Abra outro terminal e navegue até a pasta do projeto, começaremos este tutorial a partir daí.

...

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, rode o seguinte comando que além de baixar a dependência também salva a mesma no seu packages.json:

```
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, 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:

```
var MongoClient = require('mongodb').MongoClient;
mongoClient.connect('mongodb://username:password@host:port/database')
    .then(conn => global.conn = conn)
    .catch(err => console.log(err))

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:

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

Preencha conforme os dados que você cadastrou na Umblar durante a criação do MongoDB.

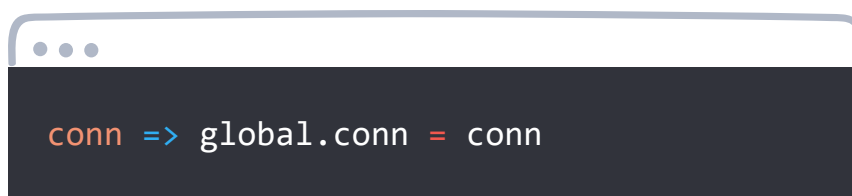
Nota: existem formas mais elegantes de armazenar essas informações, mas deixarei para ebooks posteriores.

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).

(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:



```
conn => global.conn = conn
```

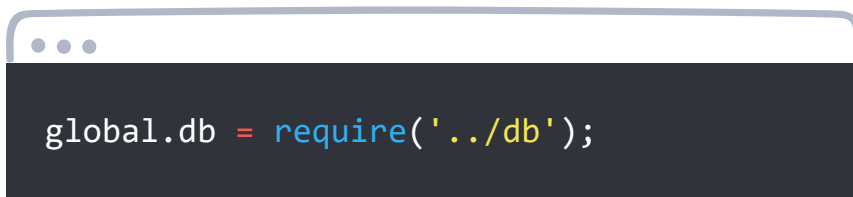
Essa é uma sugestão bem comum, fazer uma única conexão com o MongoDB e compartilhá-la globalmente com toda sua aplicação.

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 www que fica na pasta bin do seu projeto Node.js e adicione a seguinte linha no início dele:



```
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 './'.

A seguir, vamos modificar a nossa aplicação para que ela mostre os customers cadastrados do banco de dados na tela inicial, usando esse db.js que acabamos de criar.

Listando os clientes

Para conseguirmos fazer uma listagem de clientes, o primeiro passo é ter uma função que retorne todos os clientes em nosso módulo db.js, assim, adicione a seguinte função ao seu db.js (antes do module.exports):

```
function findAll(callback){  
  global.conn.collection('customers').find({}).toArray(callback);  
}
```

Nesta função 'findAll', 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 final do mesmo db.js, modifique o module.exports para retornar a função findAll. Isso é necessário para que ela possa ser chamada fora deste arquivo:

```
module.exports = { findAll }
```

Agora , vamos programar a lógica que vai usar esta função. Abra o arquivo routes/index.js e edite a rota padrão GET / para que quando essa página for acessada, ela faça a consulta por todos os customers no banco, da seguinte maneira:

```
/* GET home page. */
router.get('/', function(req, res) {
  global.db.findAll((e, docs) => {
    if(e) { return console.log(e); }
    res.render('index', { docs });
  })
})
```

router.get define a rota que trata essas requisições com o verbo GET, já vimos isso antes. Quando recebemos um GET /, a função de callback dessa rota é disparada e com isso usamos o findAll que acabamos de programar. Por parâmetro passamos a função callback que será executada quando a consulta terminar, exibindo um erro (e) ou renderizando a view index com o array de documentos (docs) como model.

Isso ainda não lista os clientes pois não preparamos a view para tal, mas já envia os dados para ela.

Agora vamos arrumar a nossa view para listar os clientes. Abra a view views/index.ejs e altere a TR em que é exibida uma mensagem de que 'não há clientes cadastrados' para que essa mensagem somente seja exibida no caso de não haverem de fato clientes no model:

```
<% if(!docs || docs.length == 0) { %>
  <tr>
    <td colspan='4'>Nenhum cliente cadastrado.</td>
  </tr>
<% } %>
```

E agora, junto à chave que fecha o if no bloco anterior, adicione um else para que execute um laço sobre o array de documentos do model e, dessa maneira, construa a nossa tabela:

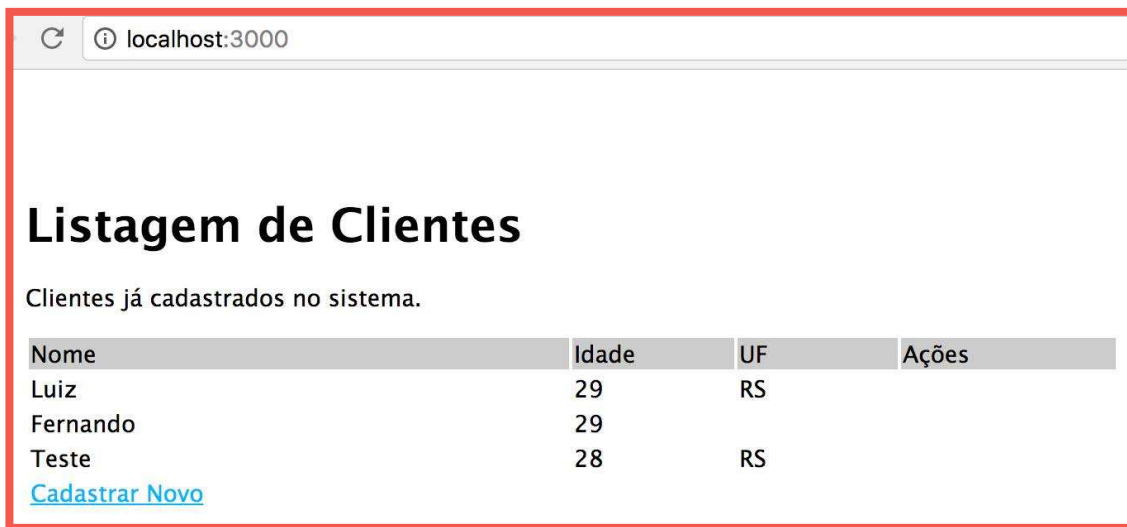
```
<% } else {  
  docs.forEach(function(customer){ %>  
    <tr>  
      <td style='width:50%'><%= customer.nome %></td>  
      <td style='width:15%'><%= customer.idade %></td>  
      <td style='width:15%'><%= customer.uf %></td>  
      <td style='width:20%'><!-- em breve --></td>  
    </tr>  
  <% })  
}%>
```

Aqui estamos dizendo que o objeto docs, que será retornado pela rota que criamos no passo anterior, será iterado com um forEach e seus objetos utilizados um-a-um para compor linhas na tabela com seus dados.

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:

```
npm start
```

Agora abra seu navegador, acesse <http://localhost:3000/> e maravilhe-se com o resultado.



A screenshot of a web browser window with the address bar showing 'localhost:3000'. The page title is 'Listagem de Clientes'. Below the title, it says 'Clientes já cadastrados no sistema.' followed by a table with four columns: 'Nome', 'Idade', 'UF', and 'Ações'. The table contains three rows of data: Luiz (29, RS), Fernando (29, RS), and Teste (28, RS). Below the table is a blue link that says 'Cadastrar Novo'.

Nome	Idade	UF	Ações
Luiz	29	RS	
Fernando	29	RS	
Teste	28	RS	

[Cadastrar Novo](#)

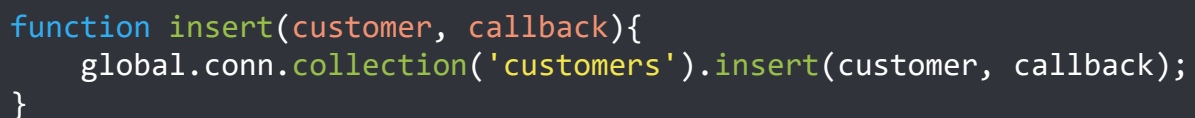
Se você viu a página acima é porque sua conexão com o banco de dados está funcionando! Note que deixei a coluna Ações vazia por enquanto. Vamos cuidar dela depois.

...

Cadastrando novos clientes

Listar dados é moleza e salvar dados no MongoDB não é algo particularmente difícil. No tutorial do capítulo anterior nós deixamos uma rota GET /new que renderiza a view new.ejs.

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:



```
function insert(customer, callback){
  global.conn.collection('customers').insert(customer, callback);
}
```


Não esqueça de adicionar essa nova função no `module.exports` no final do arquivo:

```
module.exports = { findAll, insert }
```

Agora abra o seu `routes/index.js` e vamos criar uma rota POST `/new` que vai receber o POST do HTML FORM da view `new.ejs`. Nós chamaremos o objeto global `db` para salvar os dados no Mongo adicionando o seguinte bloco de código:

```
/* POST new page. */
router.post('/new', function(req, res, next) {
  const nome = req.body.nome;
  const idade = parseInt(req.body.idade);
  const uf = req.body.uf;
  global.db.insert({nome, idade, uf}, (err, result) => {
    if(err) { return console.log(err); }
    res.redirect('/');
  })
});
```

Obviamente, no mundo real, você irá querer colocar validações, tratamento de erros e tudo mais. Aqui, apenas pego os dados que foram postados no body da requisição HTTP usando o objeto `req` (request/requisição). Crio um JSON com essas variáveis e envio para função `insert` que criamos agora a pouco.

Nota: o nome das variáveis que vem no corpo da requisição (`req.body`) são definidas no atributo `name` dos campos do formulário.

Na função de callback exigida pelo insert colocamos um código que imprime o erro se for o caso ou redireciona para a index novamente para que vejamos a lista atualizada.

Atenção: note que usei `parseInt` para converter o `req.body.idade`. Isso porque todos os dados vem como string no corpo das requisições HTTP e para que o dado seja salvo corretamente no Mongo, devemos garantir que ele está com o tipo certo (o Mongo faz inferência de tipo). Caso eu não fizesse o `parseInt`, o Mongo salvaria a idade como texto, o que complicaria em futuras consultas.

Se você reiniciar sua aplicação agora e testar o cadastro, verá que ele já está funcionando corretamente, concluindo a segunda etapa.

...

Excluindo um cliente

Agora em nossa última parte do tutorial, faremos a exclusão!

Vamos voltar à `index.ejs` (tela de listagem) e adicionar um link específico para exclusão, na coluna Ações, como abaixo:

```
<td style='width:20%'>
  <a href='/delete/<%= customer._id %>' onclick='return confirm('Tem
  certeza?');'>Excluir</a>
</td>
```

Aqui, o link de cada cliente aponta para uma rota `/delete` passando `_id` do mesmo. Essa rota será acessada via GET, afinal é um link, e devemos configurar isso mais tarde.

Tomei a liberdade de incluir um confirm JavaScript para solicitar uma confirmação de exclusão. Isso evita que um clique acidental no link de exclusão jogue fora o cadastro de um cliente.

Teste e entenderá o que estou falando.

Vamos no db.js adicionar nossa última função, de delete:

```
function deleteOne(id, callback){
  global.conn.collection('customers').deleteOne({_id: new ObjectId(id)},
  callback);
}

module.exports = { findAll, insert, deleteOne }
```

Essa função é bem simples de entender se você fez todas as operações anteriores. Na sequência, vamos criar a rota GET /delete no routes/index.js:

```
/* GET delete page. */
router.get('/delete/:id', function(req, res) {
  var id = req.params.id;
  global.db.deleteOne(id, (e, r) => {
    if(e) { return console.log(e); }
    res.redirect('/');
  });
});
```

Nessa rota, após excluirmos o cliente usando a função da variável `global.db`, redirecionamos o usuário de volta à tela de listagem para que a mesma se mostre atualizada.

E com isso finalizamos nosso CRUD!

Espero que tenha gostado desse capítulo, porque eu gostei bastante de escrevê-lo!



SEGUINDO EM FRENTE



Think twice, code once.

- Waseem Latif



Este ebook termina aqui, mas a sua missão de se tornar um programador Node.js está longe de acabar!

Se eu fiz bem o meu trabalho, você está agora com uma vontade louca de aprender mais e criar aplicações incríveis com Node.js, que o tornem um profissional diferenciado no mercado. A demanda por programadores desta fantástica tecnologia ainda é pequena no Brasil, mas isso está mudando, e rápido.



E meus parabéns, você está saindo na frente!

Este guia é propositalmente pequeno, com pouco mais de 50 páginas. 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 guia e já conhece o básico de como criar aplicações com esta plataforma, inicie hoje mesmo (não importa se for tarde) um projeto de aplicação que as use. 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 conceitos mais avançados, sugiro dar uma olhada em meu blog <http://www.luiztools.com.br> e na página <http://academy.umbler.com/>.

Outras fontes excelentes de conhecimentos específico de Node é o blog <http://blog.risingstack.com>, enquanto que sobre JavaScript a <https://developer.mozilla.org/pt-BR/> possui tudo que você pode precisar!

SEGUINDO EM FRENTE

Caso tenha gostado do material, indique esse ebook a um amigo que também deseja começar a aprender 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 amigos@umbl.com que estamos sempre dispostos a melhorar.

Um abraço e até a próxima!

