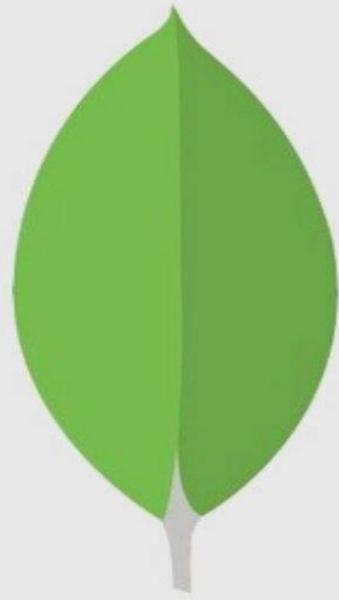


HTML



JS

B



Programação Web com

NODE.JS

LUIZTOOLS

Luiz Fernando Duarte Júnior

Programação web com Node.js

Completo, do front-end ao back-end

3^a edição

Gravataí/RS
Edição do Autor
2020

Copyright © Luiz Fernando Duarte Júnior, 2020

Programação Web com Node.js

Completo, do front-end ao back-end

Luiz Duarte

Duarte, Luiz,

Programação Web com Node.js - Completo, do front-end ao back-end - 3^a edição

Luiz Duarte - Gravataí/RS:2020

ISBN 978-65-900538-4-8

1. Programação de Computadores. 2. Computação, Internet e Mídia Digital.

Reservados todos os direitos.

Índice

[Programação Web com Node.js](#)

[Completo, do front-end ao back-end](#)

[Reservados todos os direitos.](#)

[Índice](#)

[Sobre o autor](#)

[Antes de começar](#)

[Para quem é este livro](#)

[1 Introdução à Programação Web](#)

[Um breve histórico](#)

[Referências](#)

[2 Configurando o ambiente](#)

[Node.js](#)

[Visual Studio Code](#)

[Google Chrome](#)

[MongoDB](#)

[MongoDB Compass](#)

[Postman](#)

[Referências](#)

[3 A Linguagem JavaScript](#)

[Características do JavaScript](#)

[Declaração de Variáveis](#)

[Tipos de dados](#)

[Comentários](#)

[Operadores](#)

[Functions](#)

[O tipo String](#)

[Estruturas de Controle de Fluxo](#)

[Arrays](#)

[O tipo Object](#)

[Referências](#)

[4 A plataforma Node.js](#)

[História do Node.js](#)

[Características do Node.js](#)

[Express](#)

[Routes e Views](#)

[Event Loop](#)

[Referências](#)

[5 Front-end: HTML](#)

[Introdução ao HTML](#)

[As tags HTML](#)

[Formulários HTML](#)

[Node.js + HTML](#)

[Referências](#)

[6 Back-end: MongoDB](#)

[Introdução ao MongoDB](#)

[Quando devo usar MongoDB?](#)

[Quando não devo usar MongoDB?](#)

[Instalação e Testes](#)

[Comandos elementares](#)

[Node.js + MongoDB](#)

[Referências](#)

[7 Back-end: Web APIs](#)

[Introdução ao HTTP](#)

[Verbos HTTP e o protocolo REST](#)

[Formato de Dados](#)

[Criando uma Web API](#)

[Referências](#)

[8 Front-end: JavaScript Client-Side](#)

[A tag SCRIPT](#)

[Document Object Model](#)

[Eventos JavaScript](#)

[Manipulando o DOM](#)

[Ajax](#)

[Referências](#)

[9 Front-end: CSS](#)

[CSS Inline](#)

[Classes de Estilos em CSS Internal](#)

[CSS e JavaScript](#)

[Exercitando](#)

[Bootstrap](#)

[JavaScript e Bootstrap com Node.js](#)

[Referências](#)

[10 Boas práticas com Node.js](#)

[Bons Hábitos e Princípios Gerais](#)

[Boas Práticas na Codificação](#)

[Boas Práticas de Testes](#)

[Boas Práticas de Deploy](#)

[Referências](#)

[Seguindo em frente](#)

[Apêndice 1: Módulos Recomendados](#)

[Apêndice 2: jQuery](#)

Sobre o autor

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

Carrega ainda um diploma de Reparador de Equipamentos Eletrônicos (SENAI, 2005) e cerca de uma dezena de certificações em métodos ágeis (Scrum.org, 2010 e mais) e coaching (IBC, 2018 e mais) obtidas em uma década estudando e praticando estes assuntos, sendo algumas delas com reconhecimento internacional, tais como: PSM-I, PSD-I, PSC-IBC, BEC-IBC, CLF, PACC-AIB, SFPC, DEPC, ISMF, IPOF, IKMF, Kanban-ASC e SAAC.

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

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

Trabalhando com Node.js desenvolveu diversos projetos para empresas de todos os tamanhos, desde grandes empresas como Softplan e Banco Topázio até startups como Busca Acelerada, Opideia 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, é fundador da DLZ, uma consultoria gaúcha de tecnologia, é professor de disciplinas de pós-graduação e é autor do blog <https://www.luiztools.com.br>, onde escreve semanalmente sobre empreendedorismo e desenvolvimento de software, bem como mantenedor de páginas em redes sociais (LuizTools) com o mesmo propósito.

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

<https://about.me/luiztools>

Conheça meus outros livros:

[Criando apps para empresas com Android](#)
[Scrum e Métodos Ágeis: Um Guia Prático](#)
[Agile Coaching: Um Guia Prático](#)
[Java para Iniciantes](#)
[Node.js e Microservices: Um Guia Prático](#)
[Programação Web com Node.js: Edição MySQL](#)
[MongoDB para Iniciantes](#)

Conheça meus cursos online:

[Curso de Scrum e Métodos Ágeis](#)
[Curso de Node.js e MongoDB](#)
[Curso de Web FullStack JS](#)
[Curso de React Native e Google Firebase](#)
[Curso de Gestão de Backlog com Jira](#)

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 livro vai lhe ensinar a programar para a web, com um foco em Javascript usando 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).

Segundo, este livro exige que você já tenha conhecimento técnico prévio sobre computadores, que ao menos saiba mexer em um e que preferencialmente possua um.

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 diversos aspectos da programação web para quem não é programador web ou está apenas começando nessa plataforma, com foco na tecnologia Node.js e na linguagem JavaScript.

Ao término deste livro você estará apto a construir softwares web usando qualquer sistema operacional que suporte a plataforma Node.js, bem como aprenderá como usar JavaScript no lado do cliente, no lado do servidor e no banco de dados MongoDB. Construirá web APIs, aplicará estilos e dinamismo em páginas HTML e as operações elementares em um sistema web (CRUD).

Também aplicará estilos profissionais no front-end usando Bootstrap.

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

1 Introdução à Programação Web

Good software, like wine, takes time.

- Joel Spolsky

A Internet é uma rede global de computadores interligados que utilizam um conjunto de protocolos padrões para servir vários bilhões de usuários no mundo inteiro. É uma rede de várias outras redes, que consiste de milhões de empresas privadas, públicas, acadêmicas e de governo, com alcance local e global e que está ligada por uma ampla variedade de tecnologias de rede eletrônica, sem fio e ópticas.

A Internet é muito mais do que o que vemos em nossos navegadores (os chamados browsers). Esta é a World Wide Web (ou simplesmente web hoje em dia), e é apenas uma das muitas faces da Internet. Dentro da Internet temos redes ponto-a-ponto, infraestrutura de apoio à e-mails e muito mais do que apenas os sites públicos que costumamos acessar todos os dias.

Programar para a Internet é um desafio muito maior do que criar softwares que rodam apenas em uma máquina local ou até mesmo em uma rede privada. Programar para Internet é ter de lidar com dispositivos heterogêneos, larguras de banda variadas, distâncias inimagináveis e diversas outras limitações. Mas ao mesmo tempo programar para a Internet lhe dá um alcance, um poder, muito maior do que visto nas disciplinas mais tradicionais de programação.

Em linhas gerais, esta disciplina irá trabalhar as competências e tecnologias necessárias para se programar sistemas para Internet, mais especificamente para a web, ou Internet comercial, essa que usamos tradicionalmente em nossos computadores e celulares, mas focando mais nos primeiros.

A tabela comparativa abaixo cita algumas vantagens e desvantagens em relação à programação tradicional para uma máquina desktop, que foi vista em disciplinas anteriores. Estes são fatos gerais, embora existam casos em que podemos criar sistemas para a Internet arrastando e soltando componentes, ou que podemos publicar um software desktop de maneira remota. É apenas para termos alguma ideia das diferenças gerais entre eles.

Programação Desktop	Programação Web
Construção de interfaces arrastando e soltando	Linguagem de marcação específica para construir interfaces
Curva de aprendizagem menor	Mais coisas para aprender
Publicação local, tem de estar em frente à máquina do cliente	Publicação remota, feita através da própria Internet
Pouca preocupação com segurança no sistema (geralmente AD/LDAP já dão conta de tudo)	Segurança é sempre preocupante pois o sistema está aberto ao mundo
Consumo de recursos do desktop não aumenta conforme o número de usuários	Consumo de recursos do servidor aumenta conforme o número de usuários
Somente acessa onde está instalado, máquina local ou rede empresarial	Acessa de qualquer navegador de qualquer dispositivo

Um breve histórico

O embrião da Internet nasceu de motivos militares para comunicação entre as tropas, mas hoje seu uso é global e nada se assemelha ao que era no passado. A seguir, uma breve timeline dos fatos mais marcantes da história da Internet e da web para nós:

Década de 1960, Estados Unidos encomenda pesquisa para criar tecnologia que permitisse conectar seus computadores visando comunicação eficiente, rápida e barata entre longas distâncias.

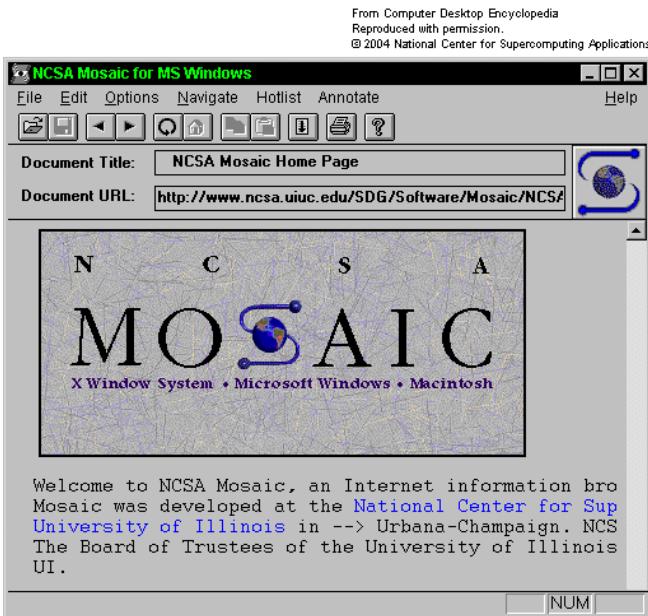
Década de 1980, universidades de diversos países que possuíam pesquisas com redes de computadores passam a se comunicar corriqueiramente usando terminais de Internet, apenas em modo texto (console). Aqui passou-se a investir muito dinheiro na aquisição de backbones para comunicação entre as redes de computadores internacionais.

1989 a 1991, Tim Berners-Lee criou um projeto hipertexto que permitia documentos com informações "linkasse" uns aos outros através da Internet, criando o embrião do que mais tarde se tornaria a World Wide Web, ou WWW. Berners-Lee ainda montou a W3C em 1994, consórcio de empresas e entidades que regulamentam os padrões da web mundial, principalmente no quesito hipertexto.



1993: é lançado o Mosaic, primeiro navegador web que daria origem ao famoso Netscape Navigator, e que mais tarde licenciado pela

Microsoft dando origem ao famoso Internet Explorer, que viria a “matar” seu “irmão”.



Década de 1990, a venda comercial em larga escala de conexões à Internet torna possível a não-militares e não-acadêmicos se conectarem à rede mundial à partir de suas casas. A partir daqui a Internet passou a afetar a cultura e comércio dos países de maneira drástica.

Anos 2000, o advento da banda-larga torna possível a transmissão de grandes volumes de dados com uma velocidade sem precedência, permitindo transferências de grandes arquivos e streaming. Com isso as mídias tradicionais começam um processo drástico de reformulação.

2004: surge o Mozilla Firefox ameaçando o reinado do IE, criado nas férias de verão por um adolescente americano que se juntou a outros colegas do projeto Mozilla, que inclusive era financiado pela Netscape.

2008: surge o Google Chrome, browser que viria a tirar o reinado do IE e que hoje representa mais de 50% dos navegadores conectados

à Internet mundialmente, seguido pelo IE com 22% e pelo Firefox com 19%.

2014: surge o Marco Civil da Internet no Brasil, onde pela primeira vez na história do país passam a existir leis específicas de atuação no ambiente digital da Internet, cuja legislação até então era vaga e praticamente inexistente (geralmente interpretavam-se disputas legais envolvendo a Internet com base em mídias como TV e rádio).

Mantenedores

Cada país possui suas próprias regras com relação ao uso e distribuição da Internet, ou seja, a regulamentação da Internet cabe ao governo de cada país. Entretanto, toda a parte de endereçamento da Internet compete ao ICANN (nomes de domínio e DNS) e a parte técnica ao IETF (protocolos e tecnologias), entidades globais sem fins lucrativos que visam a padronização da Internet mundial.

Como falado anteriormente, também temos a W3C, que define os padrões técnicos da web.

Como funciona um sistema web?

Em linhas gerais temos o seguinte fluxo do lado do usuário:

- O usuário abre seu computador com um navegador de Internet e uma conexão com um provedor de Internet (geralmente uma linha telefônica).
- O usuário digita uma URL contendo o domínio de Internet (também chamado de domínio público ou “endereço do site”) do sistema.
- Com este domínio em mãos, seu computador envia uma requisição para o servidor DNS do seu provedor de Internet (ISP) visando que ele lhe diga o endereço IP do servidor onde está aquele conteúdo. Os domínios públicos são para facilitar e organizar o acesso à Internet, mas na verdade, precisamos do endereço IP para acessar qualquer coisa na Internet.

- Caso o ISP já saiba qual o IP associado àquele endereço ele lhe retorna diretamente, caso contrário terá de perguntar a um servidor de DNS “mais alto” na hierarquia, sendo que as extensões dos domínios orientam os servidores DNS neste sentido.
- Com o IP em mãos, seu computador vai enviar uma requisição HTTP, o protocolo de transferência de hipertexto dizendo, entre outras informações, o endereço IP do servidor onde está o site e o arquivo que deseja acessar naquele servidor (geralmente uma tela do sistema).
- Quando a requisição chega ao servidor será avaliada o tipo de requisição realizado, pois podemos querer ler um arquivo, escrever informações, excluir etc. Além disso, dependendo do arquivo que estamos requisitando, pode ser necessário a presença de um manipulador (handler) específico para tratar aquela requisição, como é o caso de páginas Java e PHP, por exemplo. Este tratamento é feito pelos servidores web, como Apache e IIS.
- Após decidir o que fazer com a requisição, o servidor retorna uma resposta para o usuário. Aqui tem um ponto importante pois o navegador do usuário somente entende respostas HTTP com conteúdo em HTML (que veremos mais à frente). Ou seja, o servidor web terá que traduzir a resposta do sistema, escrito em Java por exemplo, para essa única linguagem que o browser entende.
- Aqui encerra-se o ciclo básico e genérico, com o usuário recebendo uma resposta em seu navegador após ter realizado uma ação no sistema web.

Como programo um sistema web?

Falando de tecnologias, a linguagem de marcação básica para criação de páginas web é o HTML, você precisará conhecê-la para construir as interfaces gráficas do seu sistema.

Visando tornar o HTML mais dinâmico você também precisará conhecer JavaScript, que já falamos bastante aqui anteriormente. Inicialmente a função do JavaScript era somente tornar o HTML dinâmico, mais tarde com o advento do Node.js, ele passou a desempenhar mais funções.

Visando tornar as interfaces HTML mais atraentes, você deverá conhecer CSS, um conjunto de estilos personalizáveis aplicados sobre o HTML.

Com esses três itens você tem o que chamamos de front-end ou apresentação do sistema web. Entretanto, com apenas estes três itens você não conseguirá fazer muito mais do que um site, que não salva dados, ou envia mensagens, ou possua controle de acessos, algumas características intrínsecas ao desenvolvimento de sistemas para Internet. Aí que entra o back-end, ou camada do servidor.

Todo o núcleo da sua aplicação fica longe do usuário, em um servidor web, que tratará as requisições de todos usuários em um único lugar, com um único código fonte. É como se fosse um software que não tem interface (a interface é feita em HTML, lembra?), 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 web:

- Java
- PHP
- ASP.NET
- Rails
- Django
- Node.js

Dentre todas essas tecnologias de back-end, escolhi para esse livro a tecnologia Node.js.

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.

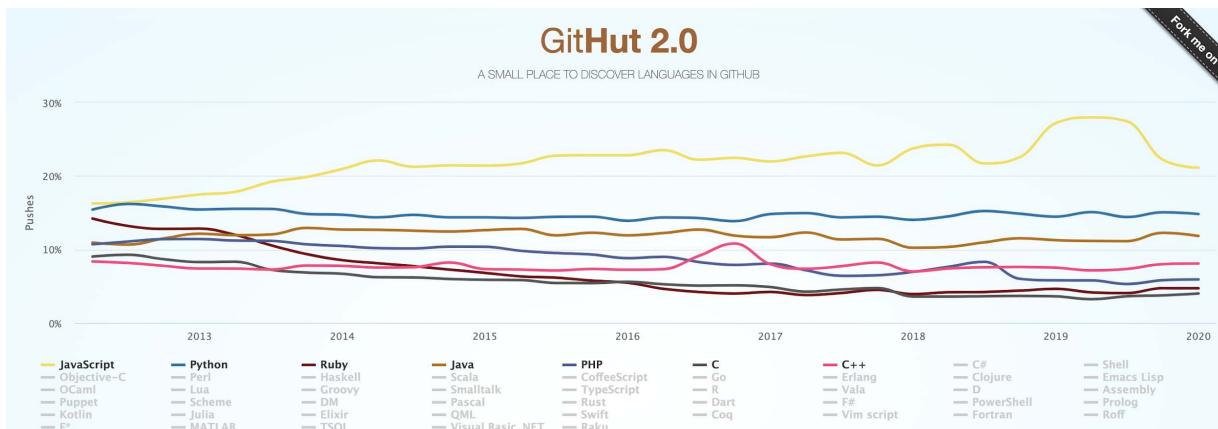
Ambientes de desenvolvimento modernos fornecem recursos de edição e debugging específicas para aplicações Node.js como Atom, Brackets, JetBrains WebStorm, Microsoft Visual Studio, NetBeans, Nodeclipse e Visual Studio Code. Também existem diversas ferramentas web como Codeanywhere, Codenvy, Cloud9 IDE, 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 dado interessante, de 2020, mostra a porcentagem de pushes, por linguagem, de projetos hospedados no GitHub. Ou seja, um a cada cinco projetos que recebem pushes no Github em 2020 são projetos JavaScript.



Fonte: <https://madnight.github.io/githut/#/pushes/2020/1>

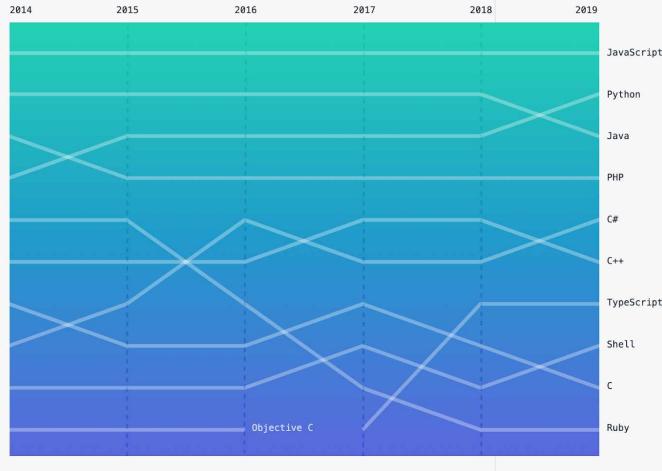
Outro gráfico, este de 2019, mostra a dominância do JavaScript como linguagem principal de projetos no GitHub, o que é consoante ao gráfico anterior, que mostrava o total de pushes na plataforma. Ou seja, desde 2014 JavaScript se mantém como a principal tecnologia para projetos open-source no mundo.

Top languages

Top languages over time

This year, C# and Shell climbed the list. And for the first time, Python outranked Java as the second most popular language on GitHub by repository contributors.*

In the last year, developers collaborated in more than 370 primary languages on GitHub.



Fonte: <https://octoverse.github.com/>

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

Eu realmente acredito que você deveria usar também.

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Os bastidores da Internet no Brasil

Excelente livro para a entender a história da Internet no país e as principais empresas que moldaram a mesma aqui e no mundo.

<https://www.luiztools.com.br/post/os-bastidores-da-internet-no-brasil-resenha/>

Palestra sobre Node.js e MongoDB

Palestra que ministrei durante o IX Telecomptech na UniLaSalle, em Canoas/RS.

<https://www.youtube.com/watch?v=4kYww7GLg0&t=1s>

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

2 Configurando o ambiente

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.

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 a versão recomendada é a estável (LTS), 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-fontes>

node -v

O resultado deve ser a versão do Node.js atualmente instalada na sua máquina, no meu caso, "v14.3.0". 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-fontes>

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-fontes>

```
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?

Código 2.4: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
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-fontes>

```
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-fontes>

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

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

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

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

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

Visual Studio Code

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

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

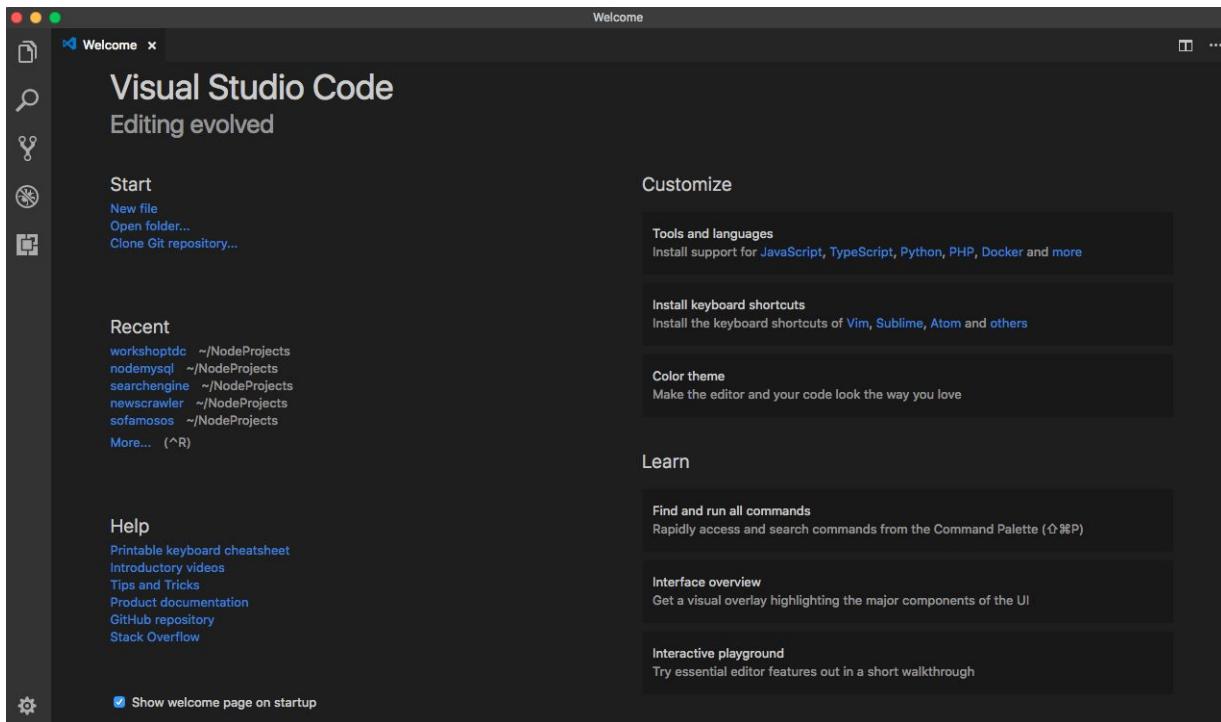
Outras excelentes ferramentas incluem o Visual Studio Community, Atom e o Sublime, sendo que o primeiro eu já utilizava quando era programador .NET e usei durante o início dos meus aprendizados com Node. No entanto, não faz sentido usar uma ferramenta tão pesada para uma plataforma tão leve, mesmo ela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

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

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

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

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



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

Código 2.7: disponível em <https://www.luiztools.com.br/livro-node-fontes>

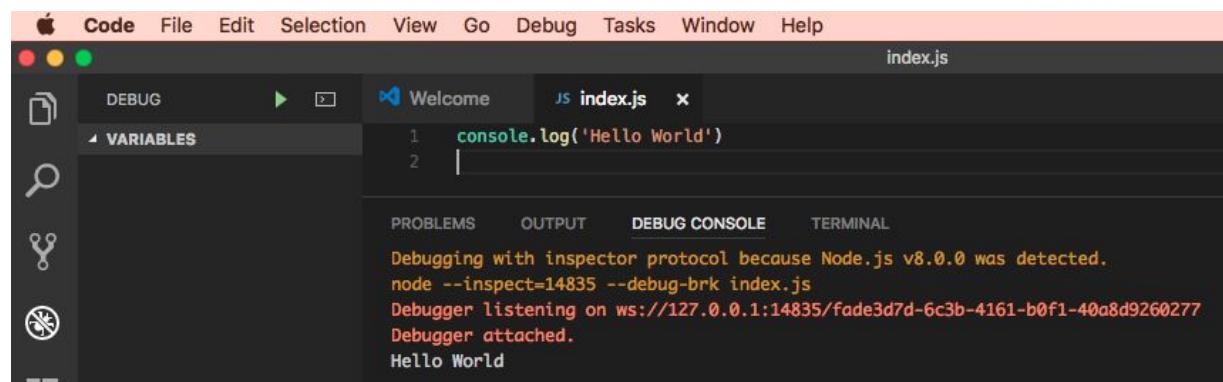
console.log('Hello World');

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.

A screenshot of the Visual Studio Code interface. The title bar shows 'index.js'. The left sidebar has icons for file, folder, search, and other development tools. The main area shows a code editor with the following content:

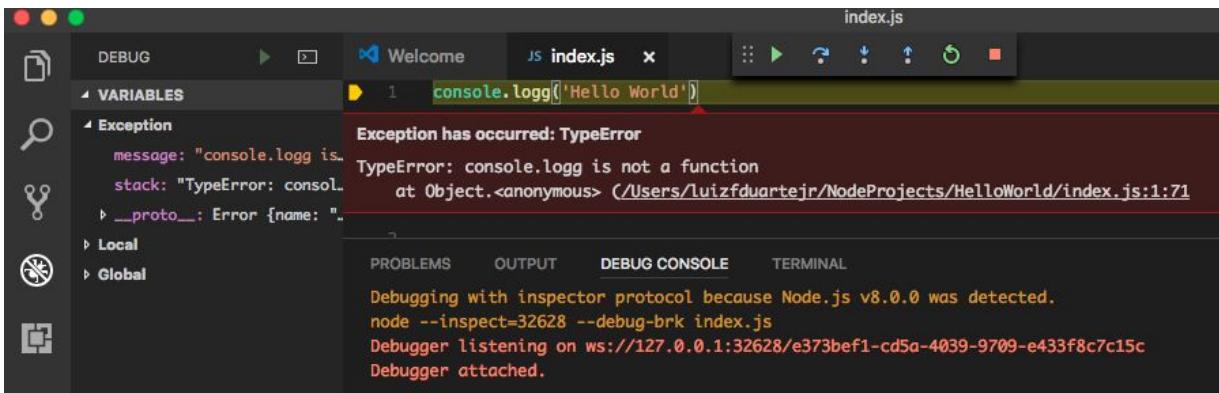
```
1 console.log('Hello World')
```

The status bar at the bottom shows the output of the debugger:

```
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.  
Hello World
```

Parabéns! Seu programa funciona!

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



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

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

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

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

Google Chrome

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

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

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

Durante muitos exemplos deste livro será necessária a utilização de um navegador de Internet. Todos os exemplos foram criados e testados usando o navegador Google Chrome, na versão mais recente disponível à época que era a versão 59. Caso não possuam o Google Chrome na sua máquina, baixe a versão mais recente no site oficial antes de avançar no livro:

<https://www.google.com.br/chrome/browser/desktop/index.html>

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

MongoDB

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

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

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

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

MongoDB Compass

Para gerenciar o MongoDB existem uma ferramenta de linha de comando disponível junto da própria pasta do banco, mas se quiser uma ferramenta visual, existe uma que eu recomendo, a MongoDB Compass.

Ela é gratuita, possui versão para os principais sistemas operacionais e é recomendada por ser a ferramenta oficial dos criadores do MongoDB:

<https://www.mongodb.com/products/compass>

No entanto, a sua instalação é perfeitamente opcional, eu sequer vou falar dela ao longo do livro, embora talvez você veja alguns prints/screenshots em alguns momentos.

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.

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Vídeo de ambiente

Vídeo sobre configuração de ambiente para desenvolver com Node.js e MongoDB.

<https://www.youtube.com/watch?v=6sS2m9HQRI8>

Vídeo de VS Code

Vídeo curto sobre uso de debug no VS Code, que gravei no meu canal.

https://www.youtube.com/watch?v=_5ql0zj7Yu0

Vídeo de MongoDB Compass

Vídeo curto sobre uso da ferramenta Compass, que gravei no meu canal.

<https://www.youtube.com/watch?v=bL5uH4V3O5o>

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

3 A Linguagem JavaScript

*Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.*

- Martin Fowler

JavaScript foi originalmente desenvolvido por Brendan Eich quando trabalhou na Netscape sob o nome de Mocha, posteriormente teve seu nome mudado para LiveScript e por fim JavaScript. LiveScript foi o nome oficial da linguagem quando foi lançada pela primeira vez na versão beta do navegador Netscape 2.0 em setembro de 1995, mas teve seu nome mudado em um anúncio conjunto com a Sun Microsystems em dezembro de 1995 quando foi implementado no navegador Netscape versão 2.0B3.

A mudança de nome de LiveScript para JavaScript coincidiu com a época em que a Netscape adicionou suporte à tecnologia Java em seu navegador (Applets). A escolha final do nome causou confusão dando a impressão de que a linguagem foi baseada em java, sendo que tal escolha foi caracterizada por muitos como uma estratégia de marketing da Netscape para aproveitar a popularidade do recém-lançado Java.

JavaScript rapidamente adquiriu ampla aceitação como linguagem de script client-side de páginas web. JScript, um dialeto compatível e intercambiável criado pela Microsoft foi incluído no Internet Explorer 3.0, liberado em Agosto de 1996.

Em novembro de 1996 a Netscape anunciou que tinha submetido JavaScript para Ecma International como candidato a padrão industrial e o trabalho subsequente resultou na versão padronizada chamada ECMAScript.

JavaScript hoje é a linguagem de programação mais popular da web. Inicialmente, no entanto, muitos profissionais denegriram a

linguagem pois ela tinha como alvo principal o público leigo. Com o advento do Ajax durante o boom da Web 2.0, JavaScript teve sua popularidade de volta e recebeu mais atenção profissional. O resultado foi a proliferação de frameworks e bibliotecas, práticas de programação melhoradas e o aumento no uso do JavaScript fora do ambiente de navegadores, bem como o uso de plataformas de JavaScript server-side.

Características do JavaScript

As seguintes características são comuns a todas as implementações conformantes com o ECMAScript.

Imperativa e Estruturada

JavaScript suporta os elementos de sintaxe de programação estruturada da linguagem C (por exemplo, if, while, switch). No entanto, características de escopo funcionam de modo ligeiramente diferente mas aproximado, desde a versão 1.7. Como C, JavaScript faz distinção entre expressões e comandos. Uma diferença sintática do C é que a quebra de linha termina automaticamente o comando, sendo o ponto-e-vírgula opcional ao fim do mesmo.

Tipagem Dinâmica

Como na maioria das linguagens de script, tipos são associados com valores, não com variáveis, sendo uma linguagem de tipagem dinâmica. Por exemplo, a variável x poderia ser associada a um número e mais tarde associada a uma string. JavaScript suporta várias formas de testar o tipo de um objeto, incluindo *duck typing*.

Baseada em Objetos

JavaScript é quase inteiramente baseada em objetos. Objetos JavaScript são arrays associativos, aumentados com protótipos. Os nomes da propriedade de um objeto são strings: obj.x = 10 e obj["x"] = 10 são equivalentes, o ponto neste exemplo é apenas sintático. Propriedades e seus valores podem ser adicionadas, mudadas, ou deletadas em tempo de execução. A maioria das propriedades de um objeto (e aqueles em sua cadeia de herança via protótipo) pode ser enumerada usando-se uma estrutura de repetição for...in. JavaScript possui um pequeno número de objetos padrão da linguagem como window e document.

Avaliação em tempo de execução

JavaScript inclui a função eval que consegue executar em tempo de execução comandos da linguagem que estejam escritos em uma string.

Funções de Primeira Classe

No JavaScript as funções são de primeira classe, isto é, são objetos que possuem propriedades e métodos, e podem ser passados como argumentos, serem atribuídos a variáveis ou retornados como qualquer outro objeto.

Funções Aninhadas

Funções 'internas' ou 'aninhadas' são funções definidas dentro de outras funções. São criadas cada vez que a função que as contém (externa) é invocada. Além disso, o escopo da função externa, incluindo constantes, variáveis locais e valores de argumento, se transforma parte do estado interno de cada objeto criado a partir da função interna, mesmo depois que a execução da função interna é concluída.

Closures

JavaScript permite que funções aninhadas sejam criadas com o escopo léxico no momento de sua definição e possui o operador () para invocá-las em outro momento. Essa combinação de código que pode ser executado fora do escopo no qual foi definido, com seu próprio escopo durante a execução, é denominada, dentro da ciência da computação, fechamento ou closure.

Protótipos

JavaScript usa protótipos em vez de classes para o mecanismo herança (mesmo em implementações modernas onde existe a keyword class). É possível simular muitas características de orientação a objetos baseada em classes com protótipos.

Funções e métodos

Diferente de muitas linguagens orientadas a objetos, não há distinção entre a definição de uma função e a definição de um método no JavaScript. A distinção ocorre durante a chamada da função; a função pode ser chamada como um método. Quando uma função é chamada como método de um objeto, a keyword this da função é associada àquele objeto via tal invocação.

A seguir apresentarei a sintaxe da linguagem JavaScript, abordando as estruturas de dados existentes, as regras para declaração de variáveis, as recomendações gerais para nomenclatura, os operadores e as estruturas de controle disponíveis.

Como será notado por quem já é programador de outras linguagens, a sintaxe da linguagem JavaScript é muito semelhante àquela usada pela linguagem C, porém muito mais dinâmica.

Declaração de Variáveis

Uma variável é um nome definido pelo programador ao qual pode ser associado um valor pertencente a um certo tipo de dados.

Para que muitas coisas aconteçam em nossos sistemas, nós precisaremos de variáveis, tal qual operações aritméticas, armazenamento de dados e mensagens para o usuário, só para citar alguns exemplos comuns.

Em outras palavras, uma variável é como uma memória, capaz de armazenar um valor de um certo tipo, para a qual se dá um nome que usualmente descreve seu significado ou propósito. Desta forma toda variável possui um nome, um tipo e um conteúdo.

O nome de uma variável em JavaScript pode ser uma sequência de um ou mais caracteres alfabéticos e numéricos, iniciados por uma letra ou ainda pelo caractere ‘_’ (underscore).

Os nomes não podem conter outros símbolos gráficos, operadores ou espaços em branco. É importante ressaltar que as letras minúsculas são consideradas diferentes das letras maiúsculas, ou seja, a linguagem JavaScript é sensível ao caixa empregado (*case sensitive*), assim temos como exemplos válidos:

- a
- total
- x2
- idade
- _especial
- TOT
- Maximo
- ExpData
- nome
- meuNumero

Segundo as mesmas regras temos abaixo exemplos inválidos de nomes de variáveis:

- 1x
- Total geral
- numero-minimo
- function

A razão destes nomes serem inválidos é simples: o primeiro começa com um algarismo numérico, o segundo possui um espaço em branco, o terceiro contém o operador menos, mas por que o quarto nome é inválido?

Porque além das regras de formação do nome em si, uma variável não pode utilizar como nome uma palavra reservada da linguagem.

Mas o que é uma palavra reservada?

As palavras reservadas são os comandos, especificadores e modificadores pertencentes a sintaxe de uma linguagem.

As palavras reservadas da linguagem JavaScript, que portanto não podem ser utilizadas como nome de variáveis ou outros elementos, são:

- abstract
- arguments
- await
- boolean
- break
- byte
- case
- catch
- char
- class
- const
- continue
- debugger

- default
- delete
- do
- double
- else
- enum
- eval
- export
- extends
- false
- final
- finally
- float
- for
- function
- goto
- if
- implements
- import
- in
- instanceof
- int
- interface
- let
- long
- native
- new
- null
- package
- private
- protected
- public
- return
- short
- static
- super

- switch
- synchronized
- this
- throw
- throws
- transient
- true
- try
- typeof
- var
- void
- volatile
- while
- with
- yield

Algumas destas palavras não são mais reservadas em versões recentes do JavaScript, mas ainda assim não são utilizadas em nomes de variáveis por boa prática. Muitas dessas palavras são reservadas do JavaScript mas não são utilizadas de fato e existem ainda uma série de outras palavras que são nomes de funções ou objetos globais existentes em diferentes ambientes como no navegador, por exemplo.

Desta forma para declararmos uma variável temos as seguintes opções:

- nomeDaVariavel
- var nomeDaVariavel
- const nomeDaVariavel
- let nomeDaVariavel

Na primeira opção, apenas escrevendo o nome da variável, declaramos a mesma como global na nossa aplicação. Isso não é exatamente verdade em Node.js, que possui algumas regras diferentes do JavaScript tradicional, e não é nem um pouco recomendado, mas existe.

Na segunda opção, a mais clássica, declaramos uma variável no seu sentido mais tradicional em JavaScript. Note que nós não declaramos o tipo da variável, 'var' simplesmente significa 'variable'. Em JavaScript a tipagem é fraca e dinâmica, o que quer dizer que ela a princípio é inferida a partir do valor do dado que guardarmos na variável, mas que o tipo pode ser alterado durante a execução do programa.

Sendo assim, quando declaramos:

Código 3.1: disponível em <https://www.luiztools.com.br/livro-node-fontes>

var x = 1;

a variável x passa a ser do tipo número inteiro, automaticamente.

Agora se declararmos na linha:

Código 3.2: disponível em <https://www.luiztools.com.br/livro-node-fontes>

x = 'teste';

a variável x passa a ser do tipo string (texto).

Já os modificadores `let` e `const` foram criados nas versões mais recentes do JavaScript para termos mais controle sobre o que acontece com nossas variáveis.

Quando declaramos uma variável com '`const`' na frente, estamos dizendo que ela é uma constante, e que seu valor somente poderá ser atribuído uma vez, caso contrário incorrerá em erro.

Curiosamente, por causa da tipagem dinâmica do JavaScript, recomenda-se declarar as variáveis sempre como `const` e, conforme a necessidade, muda-se para `let` (a seguir) ou `var`. Essa abordagem

diminui a chance de mudanças de tipo que podem estragar a sua aplicação.

Já quando declaramos uma variável com 'let' na frente, queremos dizer que ela existe apenas no escopo atual. Ao contrário do var, que após sua declaração faz com que a variável exista para todo o código seguinte, 'let' garante um controle maior da memória e um ciclo de vida mais curto para as variáveis.

Boa prática: tente declarar suas variáveis como const. Se não puder, pois elas precisam ser alteradas, tente usar let. Se não puder usar let, porque a variável precisa ser usada fora do seu escopo (o que não é recomendado), use var.

Por escopo entende-se o bloco (conjunto de comandos da linguagem) onde ocorreu a declaração da variável (geralmente delimitado por chaves).

Em JavaScript recomenda-se que a declaração de variáveis utilize nomes iniciados com letras minúsculas. Caso o nome seja composto de mais de uma palavras, as demais deveriam ser iniciadas com letras maiúsculas tal como nos exemplos:

- contador
- total
- sinal
- posicaoAbsoluta
- valorMinimoDesejado
- mediaGrupoTarefa2

E também é possível declarar mais de uma variável por vez:

Código 3.3: disponível em <https://www.luiztools.com.br/livro-node-fontes>

let x, y, z;

ou mesmo já atribuindo o seu valor, usando o operador de atribuição (sinal de igualdade):

Código 3.4: disponível em <https://www.luiztools.com.br/livro-node-fontes>

let x = 1;

Tipos de dados

Ok, falei anteriormente que JavaScript possui tipagem dinâmica, certo? Ainda assim, existe tipagem, mesmo que fraca e inferida. Os tipos existentes em JavaScript são:

- **Numbers**: podem ser com ou sem casas decimais, usando o ponto como separador. Ex: let x = 10.1
- **Strings**: podem ser definidas com aspas simples ou duplas. Ex: let nome = 'Luiz'
- **Booleans**: podem ter os valores literais true ou false, representando verdadeiro ou falso
- **Arrays**: podem ter vários valores em seu interior
- **Objects**: veremos mais pra frente, são tipos complexos que podem ter variáveis e funções em seu interior
- **Functions**: veremos mais pra frente, são tipos complexos que recebem parâmetros, executam instruções e geram saídas

Não existem diferenças entre tipos primitivos e derivados em JavaScript, como existem em outras linguagens de programação. Todas variáveis são objetos, incluindo até mesmo funções e booleanos. O impacto disso será conhecido mais tarde, mas dois efeitos colaterais é que isso torna a linguagem mais fácil de aprender e mais difícil de executar (mais lenta), em linhas gerais.

A tipagem dinâmica do JavaScript permite que você mude os tipos das variáveis em tempo de execução, embora isso não seja recomendado.

Comentários

Comentários são trechos de texto, usualmente explicativos, inseridos dentro do programa de forma que não sejam considerados como parte do código, ou seja, são informações deixadas juntamente com o código para informação de quem programa.

O JavaScript aceita dois tipos de comentários: de uma linha e de múltiplas linhas. O primeiro de uma linha utiliza duas barras (//) para marcar seu início:

Código 3.5: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
// comentário de uma linha  
// tudo após as duas barras é considerado comentário
```

O segundo usa a combinação /* e */ para delimitar uma ou mais linhas de comentários:

Código 3.6: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
/* comentário  
de múltiplas linhas */
```

Mas porque estou falando de comentários agora, logo após falar de declaração de variáveis?

Porque uma excelente maneira de melhorar os seus estudos é, a cada trecho de código escrito, comentar do que se trata a respectiva linha de código!

Conforme você for avançando e pegando experiência com JavaScript, não precisará usar mais tantos comentários, provavelmente apenas em blocos de código complexos, como forma

de documentação para uso futuro. Por ora, comente tudo o que achar conveniente para seus estudos.

Operadores

A linguagem JavaScript oferece um conjunto bastante amplo de operadores destinados a realização de operações aritméticas, lógicas, relacionais e de atribuição.

Operadores Aritméticos

Como na maioria das linguagens de programação, o JavaScript possui vários operadores aritméticos. Considere nos exemplos que `a` e `b` são variáveis numéricas.

Operador	Significado	Exemplo
<code>+</code>	Adição, soma dois valores	<code>a + b</code>
<code>-</code>	Subtração, subtrai um valor de outro	<code>a - b</code>
<code>*</code>	Multiplicação, multiplica dois valores	<code>a * b</code>
<code>/</code>	Divisão, divide um valor pelo outro e retorna o resultado da divisão	<code>a / b</code>
<code>%</code>	Módulo, divide um valor pelo outro e retorna o resto da divisão	<code>a % b</code>
<code>++</code>	Incremento unário, aumenta o valor em 1	<code>a++ ou ++a</code>
<code>--</code>	Decremento unário, diminui o valor em 1	<code>a-- ou --a</code>
<code>+=, -=, *=, /=</code>	Auto-adição, auto-subtração, etc. Calcula o valor atual com o valor à direita e armazena o resultado em si mesmo.	<code>a += b</code> (soma a e b e guarda o resultado em a,

equivalente a 'a = a
+ b')

Estes operadores aritméticos podem ser combinados para formar expressões, fazendo uso de parênteses para determinar a ordem específica de avaliação de cada expressão.

A seguir um exemplo de aplicação que declara algumas variáveis, atribui valores iniciais e efetua algumas operações imprimindo os resultados obtidos. Crie uma nova pasta para esse projeto chamada Aritmetica, com um arquivo index.js dentro dela. Abra essa pasta no VS Code (File > Open) e escreva o código abaixo dentro do referido arquivo.

Código 3.7: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
// Declaracao e inicializacao de duas variaveis, troque os valores se quiser
let a = 5;
let b = 2;
// Varios exemplos de operacoes sobre variaveis
console.log('a = ' + a);
console.log('b = ' + b);
console.log('a + b = ' + (a + b));
console.log('a - b = ' + (a - b));
console.log('a * b = ' + (a * b));
console.log('a / b = ' + (a / b));
console.log('a % b = ' + (a % b));
console.log('a++ = ' + (a++));
console.log('--b = ' + (--b));
console.log('a = ' + a);
console.log('b = ' + b);
```

Executando o código fornecido (basta apertar F5 no VS Code ou Debug > Start Debugging) teremos os resultados 5, 2, 7, 3, 10, 2.5,

1, 5, 1, 6 e 1.

Operadores Relacionais

Além dos operadores aritméticos o JavaScript possui operadores relacionais, isto é, operadores que permitem comparar valores literais, variáveis ou o resultado de expressões retornando um resultado do tipo lógico, ou seja, um resultado falso ou verdadeiro. Os operadores relacionais disponíveis são:

Operador	Significado	Exemplo
<code>==</code>	Similaridade, retorna true se os valores forem iguais, independente do seu tipo	<code>a == b</code>
<code>!=</code>	Não similaridade, retorna true se os valores não forem iguais, independente do seu tipo	<code>a != b</code>
<code>===</code>	Igualdade, retorna true se os valores e tipos forem iguais	<code>a === b</code>
<code>!==</code>	Desigualdade, retorna true se os valores ou os tipos forem diferentes	<code>a !== b</code>
<code>></code>	Maior que, retorna true se o valor da esquerda for maior que o da direita	<code>a > b</code>
<code>>=</code>	Maior ou igual que, retorna true se o valor da esquerda for maior ou igual ao da direita	<code>a >= b</code>
<code><</code>	Menor que, retorna true se o valor da esquerda for menor que o da direita	<code>a < b</code>
<code><=</code>	Menor ou igual que, retorna true se o valor da esquerda for menor ou igual que o da direita	<code>a <= b</code>

Note que temos diferentes usos para o sinal de igualdade (`=`). Se usarmos ele em duplas (`==`), comparamos valores de variáveis,

independente do seu tipo. Assim '1' == 1 é verdadeiro, pois ambos são o número 1. Por causa dessa interpretação do JavaScript, recomenda-se sempre utilizar a igualdade em trios (===), que além de comparar o valor entre as variáveis, também compara o seu tipo, para ver se coincidem. Em ambos os casos, não confunda com o uso isolado do sinal de igualdade (=), que é o operador de atribuição, que falarei mais tarde. Esse erro é um dos mais comuns por programadores JS desatentos.

Boa Prática: use sempre === e !== para comparações. Diminui a chance de 'falsos positivos' e se tem um pequeno ganho de performance entre algumas comparações.

O operador de desigualdade é semelhante ao existente na linguagem C, ou seja, é representado por "!=" ou "!==". Na versão mais 'completa'. Os demais são idênticos a grande maioria das linguagens de programação em uso.

É importante ressaltar também que os operadores relacionais duplos, isto é, aqueles definidos através de dois caracteres (>= por exemplo), não podem conter espaços em branco.

A seguir um outro exemplo simples de aplicação envolvendo os operadores relacionais. Como para o exemplo anterior, sugere-se que esta aplicação seja testada como forma de se observar seu comportamento e os resultados obtidos. Assim como sugerido anteriormente, crie uma nova pasta chamada Relacional com uma index.js dentro, mandando abrir e executar usando o VS Code.

Código 3.8: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let a = 15;
let b = 12;
let c = '15';
console.log('a = ' + a);
console.log('b = ' + b);
```

```

// aqui a igualdade é apenas sobre o valor, mas são valores
// diferentes
console.log('a == b : ' + (a == b));
// aqui, o interpretador acha que são iguais, pois o valor é o mesmo,
// mas com tipos diferentes
console.log('a == c : ' + (a == c));
// agora sim, valida-se o tipo primeiro e depois o valor
console.log('a === c : ' + (a === c));
// ou seja, use sempre === e !===
console.log('a !== b : ' + (a !== b));
console.log('a < b : ' + (a < b));
console.log('a > b : ' + (a > b));
console.log('a <= b : ' + (a <= b));
console.log('a >= b : ' + (a >= b));

```

Operadores Lógicos

Como seria esperado o JavaScript também possui operadores lógicos, isto é, operadores que permitem conectar logicamente o resultado de diferentes expressões aritméticas ou relacionais construindo assim uma expressão resultante composta de várias partes e portanto mais complexa.

Nos exemplos abaixo, a e b não são necessariamente variáveis, mas sim condições booleanas (true ou false), ou seja, podem ser qualquer tipo de expressões que retornem true ou false, incluindo variáveis booleanas simples.

Operador	Significado	Exemplo
&&	E lógico (AND), retorna true se ambas expressões retornarem true	a && b
	Ou lógico (OR), retorna true se ao menos uma expressão retornar true	a b
!	Negação lógica (NOT), retorna o valor	!a

| oposto ao da expressão |

Os operadores lógicos duplos, isto é, definidos por dois caracteres, também não podem conter espaços em branco.

Operador de Atribuição

Atribuição é a operação que permite definir o valor de uma variável através de um literal ou através do resultado de uma expressão envolvendo operações diversas. Geralmente lemos a expressão ‘`a = b`’ como ‘`a` recebe `b`’ e ‘`a = 1`’ como ‘`a` recebe `1`’. Note que isto é diferente de ‘`a == b`’ e ‘`a == 1`’, que lemos como ‘`a` é igual a `b`’ e ‘`a` é igual a `1`’ (o mesmo para `a === b` e `a === 1` como vimos anteriormente).

Exemplos de atribuições válidas são mostradas no trecho de código abaixo.

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

```
var result = false;  
i = 0;  
y = a*x + b;
```

Note como podemos fazer uma atribuição no momento que declaramos a variável, para inicializá-la com algum valor. Também podemos atribuir valores mais tarde, com valores literais ou como resultado de expressões. No entanto, se apenas declaramos uma variável e não assumimos um valor à ela, ela será considerada `undefined` (não definida) pelo interpretador JS.

Boa Prática: sempre initialize suas variáveis no momento da declaração, o que costuma melhorar a performance dos otimizadores de código JS presentes no V8, que é o motor de JS

usado pelo Node.js e também evita erros futuros por usar variáveis não inicializadas.

Que tal exercitar o aprendizado de operadores? Abra o terminal de linha de comando e inicialize o Node REPL usando o comando 'node'. Escreva as linhas abaixo, pressionando Enter a cada linha para ver o resultado final, podendo modificar este teste à sua vontade:

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

```
const i = 0;  
let y = i + 2;  
console.log('Valor do i: ' + i);
```

Realize testes ao menos uma vez com cada operador, para entender na prática como eles funcionam. Se não conseguir realizar testes com os operadores lógicos, não tem problema, revisitaremos eles no próximo tópico: Estruturas de Controle.

Functions

Outro tipo de variável existente em JavaScript são as functions ou funções. São objetos especiais que, quando invocados, executam uma série de comandos a partir de parâmetros, produzindo saídas. Functions são extremamente úteis para reaproveitamento de código e organização de projetos. Se uma mesma lógica de programação precisa ser usada em mais de um ponto do seu código, ao invés de repetir os mesmos comandos JavaScript, você pode criar uma função com eles e chamar a função as duas vezes.

A própria linguagem JavaScript possui diversas funções globais que estão à disposição do programador, bem como outras funções que estão atreladas a tipos específicos de dados como Strings e Arrays, que veremos mais adiante.

Em JS as funções são tratadas como objetos de primeiro nível, diferentemente de outras linguagens. Uma vez definidas em um arquivo JS, esta função torna-se pública para aquele arquivo (chamado de módulo em Node.js) e pode-se com as configurações adequadas torná-la disponível em outros arquivos. Sendo o JS uma linguagem procedural, uma function somente existe depois que o interpretador JS leu ela em algum arquivo ou bloco de código. Chamar uma função JS que ainda não foi carregada para a memória do navegador retornará undefined.

A sintaxe não é muito complicada: basicamente temos a palavra reservada `function` que define que o trecho de código a seguir refere-se à uma função. Damos um nome à função (seguindo as mesmas regras de nomenclatura de variáveis) e depois definimos os seus parâmetros entre parênteses, sendo que precisamos apenas dos nomes dos mesmos. Como abaixo:

Código 3.11: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function somar(num1, num2) { return num1 + num2; }
```

Note que neste exemplo usamos a instrução return para retornar o valor da soma dos dois números, mas isso é opcional. Funções JS não necessariamente precisam retornar algum valor, assim como também não precisam ter parâmetros.

Functions JS são tratadas como objetos, assim como as variáveis. Dessa forma, podemos guardar functions em variáveis para serem usadas depois ou até mesmo passá-las por parâmetro. Como abaixo:

Código 3.12: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const somar = function(num1, num2) { return num1 + num2; }
```

Mais tarde, podemos usar esta função chamando:

Código 3.13: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
somar(1,2);
```

Ou passá-la por parâmetro como abaixo:

Código 3.14: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
outraFuncao(somar);
```

Como a função foi passada para o método, pode-se chamar ela dentro do mesmo, da mesma maneira que foi mostrada no exemplo anterior à este. Esse comportamento em JavaScript geralmente é chamado de callback, enquanto que em outras linguagens costumamos chamar isso de injeção de dependência ou inversão de controle.

Em versões mais recentes de JavaScript, adicionou-se a notação "arrow functions", que permitem escrever as mesmas funções mas de maneira mais sucinta e direta, como no método somar abaixo:

Código 3.15: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const somar = (num1,num2) => num1 + num2;
```

É bem útil para agilizar a leitura e a escrita de funções pequenas, principalmente quando precisamos definir elas de maneira anônima na passagem de parâmetro para outras funções.

Diferente, não?!

Note que nos exemplos que declarei a function como uma variável, eu usei const na declaração. Isso porque essa function não será alterada, é uma constante.

O tipo String

Existem alguns tipos de dados em JavaScript que são mais complexos que outros. Na curva de aprendizado, logo após os numéricos e booleanos temos as Strings. O tipo String é um dos mais utilizados, isso porque ele é utilizado para criar variáveis que guardam texto dentro de si, algo muito recorrente em desenvolvimento de software.

O tipo String permite armazenar um número virtualmente infinito de caracteres, formando palavras, frases, textos, etc. Apesar de todo esse “poder”, sua declaração é tão simples quanto a de qualquer outra variável:

Código 3.16: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const nome = 'Luiz';
```

Note que a palavra ‘Luiz’ está entre aspas (simples) e que poderia estar entre aspas-duplas, sem qualquer diferença. No entanto, se você começou a string com aspas simples, deve terminar da mesma forma e vice-versa.

O ideal é sempre inicializarmos nossas Strings, nem que seja com aspas vazias ("") para não corrermos o risco de termos erros de execução mais pra frente, pois por padrão o conteúdo das variáveis é ‘undefined’.

Assim como vimos anteriormente na prática, quando usamos o operador ‘+’ com textos (que agora chamaremos de Strings), ele não soma os seus valores, mas sim concatena (junta) eles. Isso vale tanto para Strings literais (textos entre aspas) quanto para variáveis Strings.

Código 3.17: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const nome = 'Luiz';
const sobrenome = 'Duarte';
console.log('Prof. ' + nome + sobrenome);
```

O código acima, se executado pelo terminal interativo ou dentro de um index.js deve imprimir o texto Prof. LuizDuarte na janela do console. Isso porque ele concatenou (juntou) a String literal “Prof. ” com as variáveis nome e sobrenome, resultando em uma nova String, maior e mais completa “Prof. LuizDuarte”, que foi impressa pelo console.log.

Outra característica muito interessante das Strings é que qualquer variável que a gente concatene com uma String, acaba gerando outra String, como abaixo.

Código 3.18: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const nome = 'Luiz';
const idade = 32;
console.log('Prof. ' + nome + ' tem ' + idade + ' anos!');
```

Isso vai imprimir o texto “Prof. Luiz tem 32 anos！”, copiando o valor da variável idade pra dentro da nova String completa. Note que isso não altera o tipo da variável idade ou seu valor, apenas faz uma cópia dela em forma de texto, automaticamente.

Legal, não é mesmo?!

Mas as Strings são bem mais complexas e interessantes que isso, existindo uma série de características e funções que podemos chamar a partir delas para obter variados efeitos, como nos exemplos abaixo (considere str uma variável inicializada com uma String):

- **str.length**: retorna a quantidade de caracteres da String
- **str.toUpperCase()**: retorna a String toda em maiúsculas
- **str.toLowerCase()**: retorna a String toda em minúsculas
- **str.endsWith('x')**: retorna true se a String termina com o caracter (ou palavra) passado por parâmetro
- **str.startsWith('x')**: retorna true se a String começa com o caracter (ou palavra) passado por parâmetro
- **str.replace('x', 'y')**: retorna uma String cópia da original, mas com os caracteres x substituídos por y
- **str.trim()**: retorna uma String cópia da original, mas sem espaços no início e no fim
- **str.concat(str2)**: faz a fusão dos caracteres de str2 no final de str
- **parseInt(str)**: retorna um número a partir da String (se for possível)
- **parseFloat(str)**: retorna um número decimal a partir da String (se for possível)
- **x.toString()**: retorna uma String criada a partir de outra variável qualquer (x)

Note que nenhuma dessas funções muda a String em si, todas retornam um valor a partir dela. Strings são imutáveis na maioria das linguagens de programação, e JavaScript não é exceção. Quando colocamos um texto dentro de uma variável que já possuía um texto, não mudamos o texto original, mas criamos um novo.

Para exemplificar estes conceitos, escreva e execute o código abaixo:

Código 3.19 disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let str = 'Teste';
console.log(str.length);
console.log(str.toUpperCase());
console.log(str.toLowerCase());
```

```
str = '1';
console.log(parseInt(str));
str = '1.5';
console.log(parseFloat(str));
```

E as Strings possuem muitas outras características incríveis devido à sua natureza textual, que é interpretada internamente como um array (vetor) de caracteres. Exploraremos essas características mais pra frente, quando entrarmos em estruturas de dados mais complexas.

Está começando a ficar mais interessante!

Estruturas de Controle de Fluxo

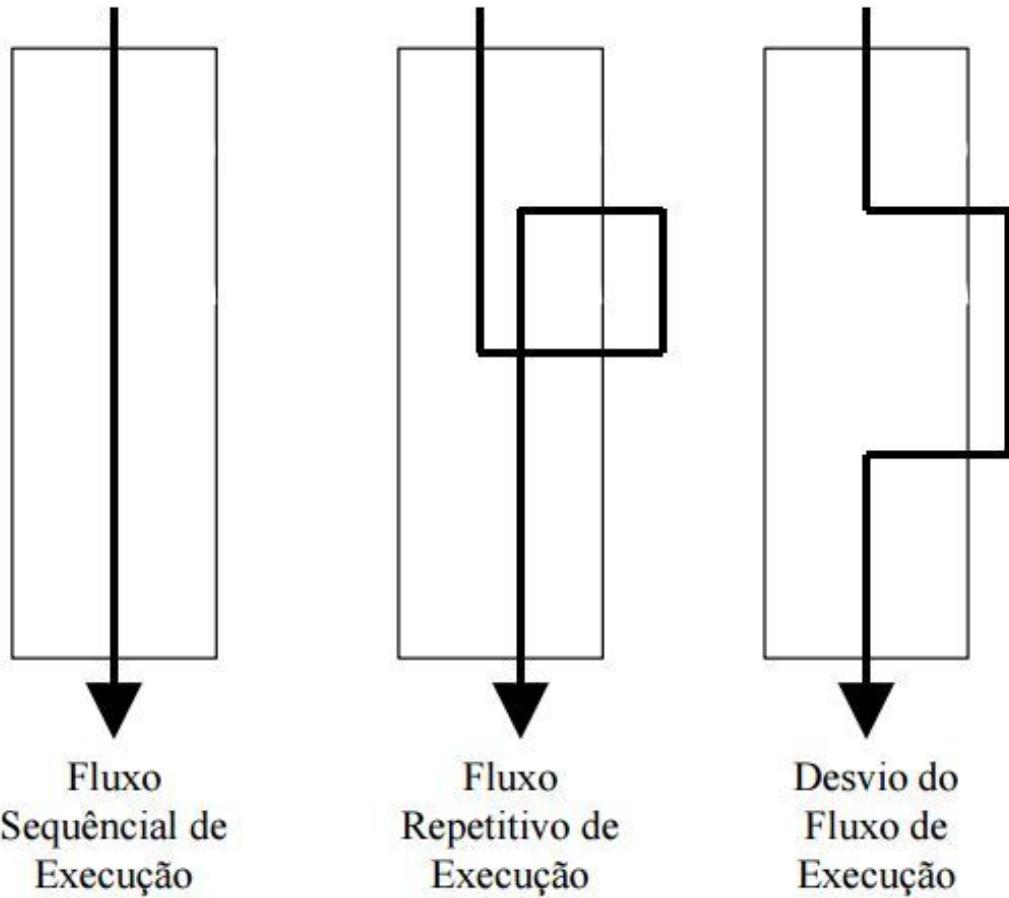
Um programa de computador é uma sequência de instruções organizadas de forma tal a produzir a solução de um determinado problema. Naturalmente tais instruções são executadas em sequência, o que se denomina fluxo sequencial de execução. Em inúmeras circunstâncias é necessário executar as instruções de um programa em uma ordem diferente da estritamente sequencial. Tais situações são caracterizadas pela necessidade da repetição de instruções individuais ou de grupos de instruções e também pelo desvio do fluxo de execução.

As linguagens de programação tipicamente possuem diversas estruturas de programação destinadas ao controle do fluxo de execução, isto é, estruturas que permitem a repetição e o desvio do fluxo de execução. Geralmente as estruturas de controle de execução são divididas em:

Estruturas de repetição: destinadas a repetição de um ou mais comandos, criando o que se denomina laços. O número de repetições é definido por uma condição, que pode ser simples como uma comparação numérica, ou complicado como uma expressão lógica. No JavaScript dispõe-se das diretivas **for**, **while** e **do/while**.

Estruturas de desvio de fluxo: destinadas a desviar a execução do programa para uma outra parte, quebrando o fluxo sequencial de execução. O desvio do fluxo pode ocorrer condicionalmente, quando associado a avaliação de uma expressão, ou incondicionalmente. No JavaScript dispomos das diretivas **if/else** e **switch/case**.

A imagem abaixo ilustra os três tipos possíveis de fluxos de execução em um software: o sequencial, que seria o fluxo padrão; um fluxo repetitivo, onde temos um laço de repetição em determinado momento; e um fluxo com desvio, cuja “direção” é decidida com base em uma condição lógica.



Agora vamos estudar cada um dos tipos de estruturas de controle básicas que possuímos no JavaScript.

Estruturas de repetição

Estas estruturas podem ser divididas entre simples e condicionais. Independente da classificação, elas repetem um ou mais comandos JavaScript durante um número de vezes definido na sua declaração ou baseado em uma expressão lógica, como veremos a seguir.

Como repetição simples consideramos um trecho de código, isto é, um conjunto de diretivas que deve ser repetido um número conhecido e fixo de vezes. A repetição é uma das tarefas mais comuns da programação utilizada para efetuarmos contagens, para obtenção de dados, para impressão etc. Em JavaScript dispomos da diretiva **for** cuja sintaxe é dada a seguir:

```
for (inicialização; condição de execução; incremento/decremento)
diretiva
```

O **for** (PARA) possui três campos ou seções, delimitados por um par de parênteses que efetuam o controle de repetição de uma diretiva individual ou de um bloco de diretivas (neste caso, circundado por chaves). Cada campo é separado do outro por um ponto e vírgula.

O primeiro campo é usado para dar valor inicial a uma variável de controle (um contador). Nele declaramos nossa variável de controle, geralmente um inteiro, e inicializamos ele, geralmente com 0, como no exemplo abaixo.

```
for (var i=0; condição de execução; incremento/decremento)
diretiva
```

O segundo campo é uma expressão lógica que determina a execução da diretiva associada ao for, geralmente utilizando a variável de controle e outros valores. Resumindo: o segundo campo vai determinar quantas vezes a diretiva do for será executada.
Atente ao exemplo abaixo:

```
for (var i=0; i < 10; incremento/decremento)
diretiva
```

O que você consegue abstrair a partir do segundo campo do **for**?

Considerando que **i** é a nossa variável de controle, usamos ela em uma expressão lógica para determinar quantas vezes a diretiva irá

ser executada. Neste caso, enquanto **i** for menor que 10, executaremos a diretiva (um comando JavaScript qualquer).

Mas se **i = 0**, quando que ele será maior ou igual à 10 para o **for** encerrar a repetição?

Aí que entra o terceiro campo do **for**: o incremento/decremento. Neste último campo nós mudamos o valor da variável de comando, para mais ou para menos. Esse incremento/decremento acontece uma vez a cada repetição do laço, logo após a execução da diretiva e antes da condição (segundo campo do **for**) ser analisada novamente. Isto faz com que, em determinado momento, o laço pare de se repetir, como no exemplo abaixo.

```
for (var i=0; i < 10; i++)
    diretiva
```

Neste caso, a diretiva será executada 10 vezes e sequência, uma vez que a cada execução o valor de **i** aumentará em uma unidade até alcançar o valor 10, o que fará com que a condição '**i < 10**' se torne falsa e o **for** acabe sua execução.

Não acredita? Copie o código abaixo, onde substituí a diretiva por um comando de impressão do JavaScript e execute no VS Code (ou no terminal interativo):

Código 3.20: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
for (var i=0; i < 10; i++)
    console.log('i=' + i);
```

A saída esperada é uma sequência de impressões iniciadas em '**i=0**' e terminando em '**i=9**', pois quando o **i** for igual a 10, o **for** será

encerrado.

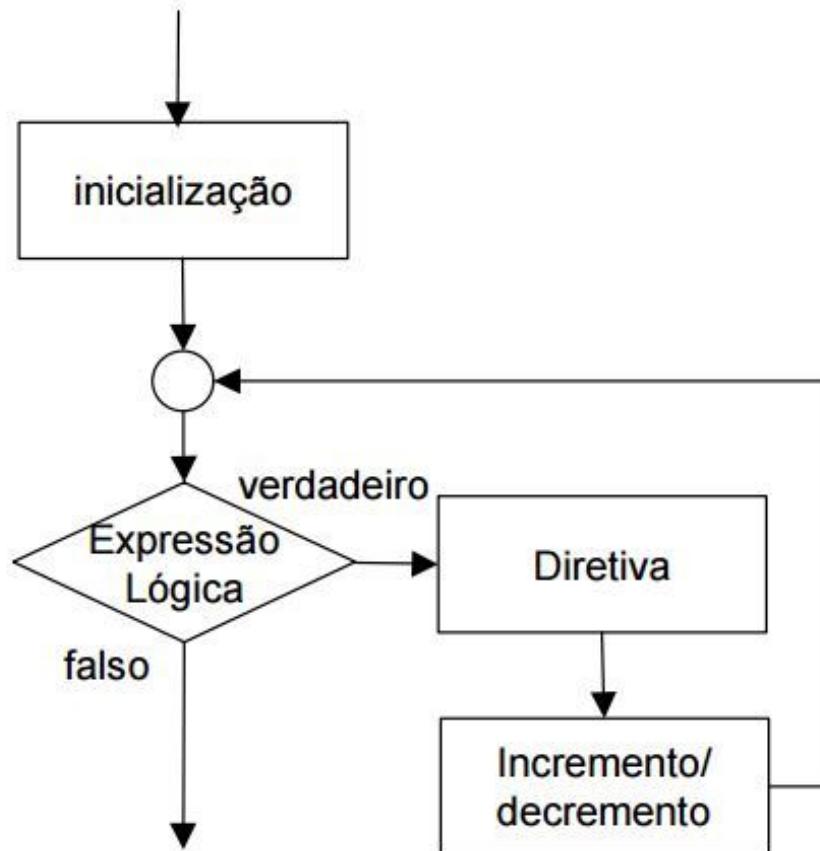
A tradução literal do **for** é PARA, e lemos o código anterior como “para uma dada variável i, inicializada com 0, até ela se tornar 10, imprima o valor de i e incremente-o logo após”.

Caso quiséssemos executar mais de uma instrução dentro do for, repetidas vezes, basta apenas circundarmos elas com chaves, como no exemplo abaixo:

Código 3.21: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
for (var i=0; i < 10; i++){  
    console.log("i=" + i);  
    //outra instrução qualquer, quantas quiser  
}
```

Podemos resumir o funcionamento da estrutura for através do fluxograma abaixo:



Mas e as estruturas de repetição mais complexas, baseadas em condições?

As estruturas de repetição condicionais são estruturas de repetição cujo controle de execução é feito pela avaliação de expressões condicionais. Estas estruturas são adequadas para permitir a execução repetida de um conjunto de diretivas por um número indeterminado de vezes, isto é, um número que não é conhecido durante a fase de programação mas que pode ser determinado durante a execução do programa tal como um valor a ser fornecido pelo usuário, obtido de um arquivo ou ainda de cálculos realizados com dados alimentados pelo usuário ou lido de arquivos.

Existem duas estruturas de repetição condicionais: **while** e **do/while**.

O **while** (ENQUANTO) é o que chamamos de laço condicional, isto é, um conjunto de instruções que é repetido enquanto o resultado de uma expressão lógica (uma condição) é avaliado como verdadeiro. Abaixo segue a sintaxe desta diretiva:

```
while (expressão_lógica)
    diretiva
```

Note que a diretiva **while** avalia o resultado da expressão antes de executar a diretiva associada, assim é possível que a diretiva nunca seja executada caso a condição seja inicialmente falsa.

Mas que tipo de expressão lógica usamos geralmente como condição do **while**?

Abaixo temos um exemplo bem simples:

```
let chave = true
while(chave === true)
    diretiva
```

Neste exemplo inicializamos uma variável chave como true (verdadeiro) e depois usamos a própria variável como condição de parada do **while**. Enquanto chave for true, o **while** continuará sendo executado (i.e. a diretiva será executada). No entanto, note que um problema típico relacionado a avaliação da condição da diretiva **while** é o seguinte: se a condição nunca se tornar falsa o laço será repetido indefinidamente.

Para que isso não aconteça, devemos nos certificar de que exista algo na diretiva que, em dado momento, modifique o valor da condição, como no exemplo abaixo:

Código 3.22: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let chave = true;
while(chave === true)
    chave = false;
```

Nesse caso, o **while** executará apenas uma vez e depois será encerrado, pois na segunda vez que a condição for analisada, a variável `chave` terá valor `false` e ele não executará sua diretiva novamente.

Lemos o trecho de código anterior como “enquanto a variável `chave` for verdadeira, troque o valor dela para `false`”.

Claro, esse é um exemplo pouco útil, uma vez que a única instrução existente é para encerrar o **while**. O que acontece geralmente é que o **while** possua mais de uma diretiva e, assim como for, elas devem ser circundadas por chaves:

Código 3.23: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let chave = true;
while(chave === true){
    //outra instrução qualquer
    console.log('Imprime!');
    chave = false;
}
```

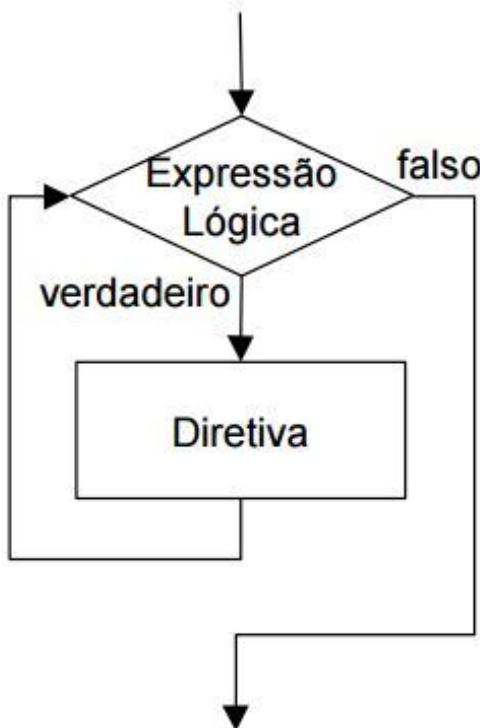
É possível ainda, usar o **while** de maneira semelhante ao **for**, definindo como condição uma comparação lógico-aritmética:

Código 3.24: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let i = 0
while(i < 10){
    //outra instrução qualquer
    console.log('Imprime!');
    i++;
}
```

Nesse caso, o texto “Imprime!” será impresso 10 vezes, assim como seria possível fazer com um **for**.

O funcionamento do while pode ser resumido através do fluxograma abaixo:



O **do/while** também é um laço condicional, isto é, tal como o **while** é um conjunto de instruções que são repetidas enquanto o resultado da condição é avaliada como verdadeira mas, diferentemente do **while**, a diretiva associada é executada antes da avaliação da expressão lógica e assim temos que esta diretiva é executada pelo menos uma vez.

Por exemplo:

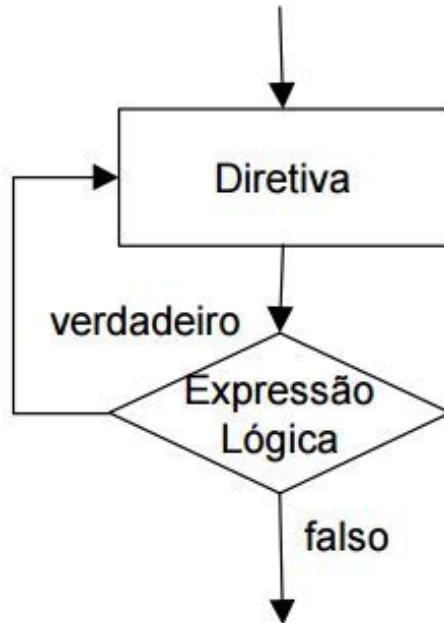
Código 3.25: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let chave = false;  
do{  
    //outra instrução qualquer  
    console.log('Imprime!');  
}while(chave === true)
```

Note que a variável chave foi declarada como false logo na sua inicialização, ou seja, quando ela for comparada na condição do **while** ela ainda será false e consequentemente sairá do **while**. Porém, como o bloco **do** é executado primeiro, teremos a impressão da palavra “Imprime!” na saída do console.

Lemos o código anterior como: “faça (DO) a impressão da palavra imprime, enquanto (WHILE) a variável chave for verdadeira”.

Basicamente esta é a única diferença entre os laços **while** e **do/while**. O **do/while** sempre executa o laço ao menos uma vez, antes de fazer a comparação, como ilustrado pelo fluxograma abaixo:



Um exercício que você pode fazer para testar o **for**, o **while** e o **do/while** é o seguinte: antes de conhecer os laços de repetição, como você faria para declarar uma variável e imprimi-la de 0 a 9?

O código abaixo exemplifica como você poderia fazer:

Código 3.26: disponível em <https://www.luiztools.com.br/livro-node-fontes>

Copie e cole o código acima dentro de um index.js, execute e você verá, que graças ao operador de incremento unário a variável x vai do valor 0 ao 9, sendo impressa durante o processo.

Repetitivo, não?!

Seria tão bom se tivéssemos uma maneira de repetir comandos JavaScript sem repetir código... É nessas horas, quando você precisa repetir comandos, que os laços de repetição são úteis!

Pare um minuto e pense como esse mesmo exercício (imprimir um número de 0 a 9) poderia ser feito usando laços de repetição...

Aqui vai uma solução, agora usando um laço **for**:

Código 3.27: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
for(let i=0; i < 10; i++)
    console.log(i);
```

Copie e cole esse código e verá que o resultado no console é o mesmo.

Muito menor, não?!

Como é uma tarefa repetitiva cuja condição de parada é numérica e os valores vão incrementando unitariamente, o **for** é a melhor opção. Mesmo assim, a título de exercício, vou mostrar implementações usando **while**:

Código 3.28: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let x=0;
while(x < 10)
    console.log(x++);
```

E outra usando **do/while**:

Código 3.29: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let x=0;  
do{  
    console.log(x);  
} while(x++ < 10)
```

Ambos exemplos fazem exatamente a mesma coisa. Estudar algoritmos de programação é exatamente isso: entender que diversos caminhos levam à mesma solução. Cabe ao programador identificar qual o melhor, neste caso, o **for**.

Um último ponto digno de nota é a forma como usei o incremento unário nestes exemplos. Note que incrementei o valor de x dentro de um `console.log` e dentro de uma estrutura **while**. Isso é perfeitamente possível e lhe permite estratégias interessantes em seus algoritmos. Temos dois operadores de incremento unário, o `++x` e o `x++`. O primeiro aumenta o valor de x em 1 e depois retorna o valor do mesmo, enquanto que o segundo retorna o valor de x e depois o aumenta em 1. Pode parecer uma pequena diferença, mas analise esse código:

Código 3.30: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let x=0;  
console.log(x++);
```

O que será impresso?

A resposta é 0.

Já neste exemplo...

Código 3.31: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let x=0;  
console.log(++x);
```

A resposta é 1. Capicce?

Então vamos passar à próxima estrutura, temos muito pra ver ainda dentro do básico de JavaScript!

Estruturas de desvio de fluxo

Existem várias estruturas de desvio de fluxo que podem provocar a modificação da maneira com que as diretivas de um programa são executadas conforme a avaliação de uma condição. O JavaScript dispõe de duas destas estruturas: **if/else** e **switch/case**.

O **if/else** é uma estrutura simples de desvio de fluxo de execução, isto é, é uma diretiva que permite a seleção entre dois caminhos distintos para execução dependendo do resultado falso ou verdadeiro resultante de uma expressão lógica.

Vamos começar do **if** básico:

```
if (expressão_lógica)  
    diretiva1
```

A tradução literal de um **if** é SE. SE a expressão entre parênteses for verdade (true), então a diretiva1 será executada. Caso contrário, ela é ignorada. Exemplo:

Código 3.32: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let x = 0;
if(x === 0)
    console.log('X é zero!');
```

O que acontece quando executamos esse código?

E se alterarmos o valor inicial de x (na primeira linha), fazendo ele receber 1 ao invés de 0?

A frase “X é zero” somente será impressa se a variável x for zero. Em qualquer outra circunstância, não.

Note que, assim como acontece nos laços de repetição, se vamos executar mais de uma diretiva, devemos agrupá-las dentro de chaves, como no exemplo abaixo, levemente modificado do anterior.

Código 3.33: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let x = 0;
if(x === 0){
    console.log('X é zero!');
    x++;
}
```

Neste exemplo, como queremos executar dois comandos no caso do x ser zero, devemos circundar os comandos com chaves. Podemos ler esta estrutura como “se a variável x for igual a zero, imprimimos que X é zero e depois incrementamos o mesmo”.

Note também que a expressão lógica que vai dentro dos parênteses do if aceitam qualquer expressão que retorne true ou false, usando os operadores lógicos (<, >, <=, >=, ==, != e !). Sendo assim, podemos fazer coisas muito mais complexas com um if do que

apenas uma comparação de igualdade, como no exemplo abaixo, onde vemos se um número é PAR (isto é, divisível por 2):

Código 3.34: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let x = 0;  
if(x % 2 === 0){  
    console.log('X é PAR!');  
}
```

Você conhece o operador de módulo (%)?

Falamos dele brevemente no tópico sobre operadores, mas só agora que estou mostrando uma utilidade dele: descobrir se um número é par ou não. O funcionamento do **mod** (apelido popular do % entre programadores) é fazer a divisão de um número pelo outro, mas ao invés de retornar o resultado, ele retorna o resto.

Ora, se dividirmos um número por 2 ($x / 2$) e o resto for zero, é porquê foi uma divisão exata e ele é PAR, certo?

E agora, unindo o conceito do **if** e do **for**, como você faria para imprimir todos os números pares de 1 a 20?

Tente fazer, e depois compare a sua solução com a apresentada abaixo:

Código 3.35: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
for(let i=1; i <= 20; i++){  
    if(i % 2 === 0){  
        console.log(i + ' é PAR!');  
    }  
}
```

Este **for** que criei no exemplo anterior está ligeiramente diferente. Primeiro, eu initializei a variável de controle com 1 ao invés do 0 tradicional. Além disso, coloquei na expressão de controle o operador ‘menor ou igual que’, comparando com o número 20, para que o 20 em si seja impresso também. Já o **if** eu aproveitei do exemplo anterior, então nenhuma novidade aqui.

Copie e cole este código e veja como ele imprime corretamente somente os números pares de 1 a 20.

Mas e se quiséssemos imprimir a mensagem “é ÍMPAR” para os demais números?

Como você faria?

Código 3.36: disponível em <https://www.luiztools.com.br/livro-node-fontes>

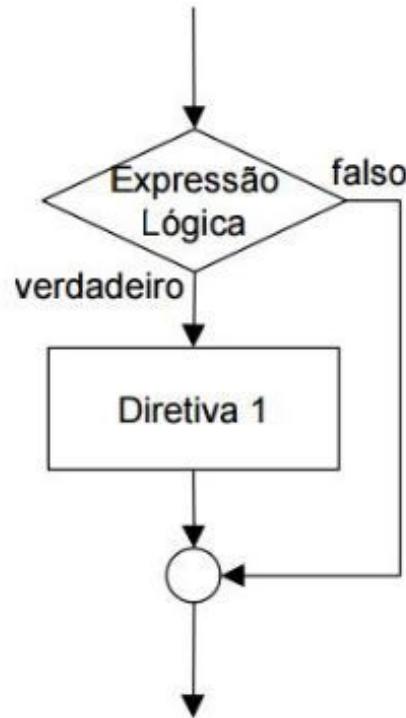
```
for(let i=1; i <= 20; i++){
    if(i % 2 == 0){
        console.log(i + ' é PAR!');
    }
    console.log(i + ' é ÍMPAR!');}
```

Essa talvez tenha sido uma solução pensada por você, mas está errada. Execute ela dentro de um arquivo index.js e notará que ela sempre imprime que todos os números são ÍMPARES, além de imprimir quando algum é PAR também.

Porquê?

O **if** é um desvio de execução, mas ao término da execução da sua diretiva, ele volta ao fluxo tradicional, como o fluxograma abaixo nos

mostra.



Uma vez que a impressão de que o número é ÍMPAR está no fluxo normal, ela sempre será executada após o **if**.

Mas então, como resolvemos isso?

Com a segunda parte da construção **if/else**, o **else**.

A tradução literal de **else** é SENÃO, e ele define basicamente o caminho alternativo que seu programa tomará caso a condição do **if** não seja verdadeira, como abaixo.

```
if (expressão_lógica)
    diretiva1
else
    diretiva2
```

Lemos essa estrutura como: “se a expressão lógica for verdadeira, executamos a diretiva 1, senão, executamos a diretiva 2”.

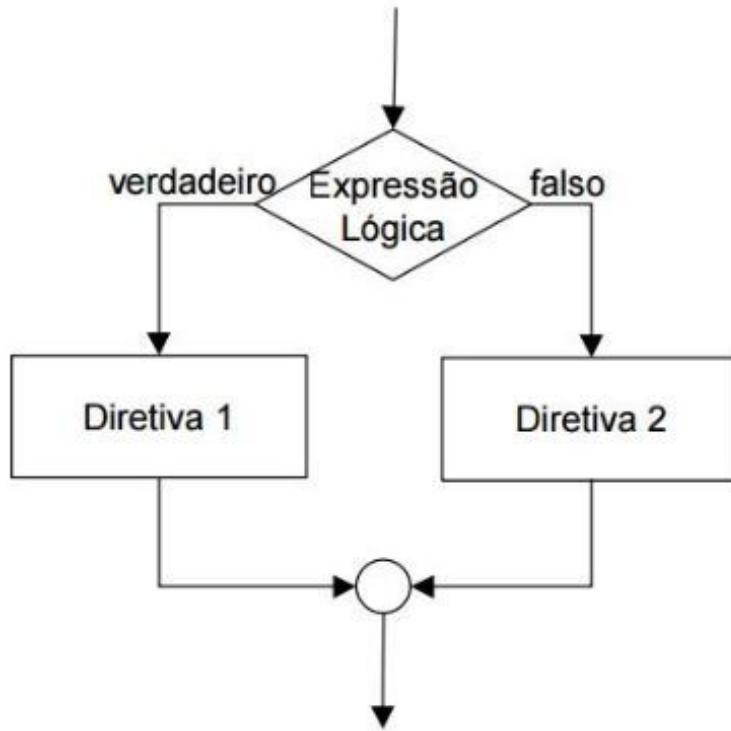
Dessa forma, conseguimos modificar nosso programa anterior para imprimir que o número é ÍMPAR, quando ele não entrar no **if** que diz que ele é PAR, como abaixo, usando **else**.

Código 3.37: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
for(let i=1; i <= 20; i++){
  if(i % 2 === 0){
    console.log(i + ' é PAR!');
  } else {
    console.log(i + ' é ÍMPAR!');
  }
}
```

Lemos o trecho mais interno do algoritmo acima como: “se *i* for divisível por 2, então imprima que ele é PAR, senão, imprime que ele é ímpar”.

O fluxograma que melhor define o comportamento de uma estrutura **if/else** como essa é visto abaixo.



Neste modelo simples de **if/else** e no anterior, com apenas **if**, consideramos que há sempre um ou no máximo dois caminhos alternativos para um dado fluxo de execução. No entanto, existem fluxos ainda mais complexos, em que podemos precisar de um encadeamento de execuções. Como no caso abaixo, em que queremos que a palavra “FUNCIONOU” seja impressa somente quando o número estiver entre 5 e 7.

Código 3.38: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```

for(let i=0; i < 10; i++){
  if(i >= 5){
    if(i <= 7){
      console.log('FUNCIONOU');
    }
  }
}

```

Alguns problemas lógicos como esse são facilmente resolvidos apenas estudando e combinando os operadores lógico e relacionais, como no caso abaixo, que faz exatamente a mesma coisa que o algoritmo anterior, mas de maneira mais eficiente.

Código 3.39: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
for(let i=0; i < 10; i++){  
    if(i >= 5 && i <= 7){  
        console.log('FUNCIONOU');  
    }  
}
```

Ou seja, somente entrará no **if** se ambas condições forem verdadeiras, uma vez que usei o operador AND (E lógico). O estudo dos operadores lhe abre um leque enorme de opções para otimizar o seu código e conseguir ‘mais’ com ‘menos’.

Para finalizar o estudo do **if/else**, cabe ressaltar que muitas vezes também precisamos que nossos **else**’s também possuam condições, que o algoritmo não entre dentro do **else** simplesmente porque não atendeu à condição do **if**. Neste caso, podemos adicionar um **if** ao **else**, o que chamamos de **else if**.

```
if(x === 1){  
    diretiva1  
} else if(x === 2){  
    diretiva2  
}
```

Lemos esta estrutura como: “se x é igual a 1, executamos a diretiva 1, senão, se x for igual a 2, executamos a diretiva 2”.

Neste caso, se o x for igual a 1, executaremos a primeira diretiva. Senão, caso o x seja igual a 2, executaremos a segunda diretiva. Agora, se x não for 1 ou 2, ele passará ‘reto’ pelo if, apenas usando o fluxo principal da execução.

Podemos ainda criar um fluxo default (padrão), colocando um (e apenas um) **else** no final da cadeia de **ifs** e **else ifs**.

```
if(x === 1){  
    diretiva1  
} else if(x === 2){  
    diretiva2  
} else {  
    diretiva3  
}
```

Neste caso, se o x não for 1, nem 2, executaremos a diretiva3.

A construção **if/else if/else** é muito poderosa e permite múltiplos fluxos de execução alternativos, uma vez que podemos combinar inúmeras condições diferentes.

Boa Prática: tentar manipular variáveis 'undefined' provoca erros em tempo de execução. Sempre que quiser saber se uma variável está undefined ou não, basta fazer um if sobre ela, pois variáveis undefined retorna false em ifs. Como no exemplo a seguir:
`if (!x) { console.log('x is undefined') }`

No entanto, se suas comparações são sempre de uma variável em relação a um valor, existe uma estrutura mais eficiente para isso: a **switch/case**.

O **switch/case** é uma diretiva de desvio múltiplo de fluxo, isto é, baseado na avaliação de uma variável é escolhido um caminho de execução dentre vários possíveis. O **switch/case** equivale

logicamente a um conjunto de diretivas **if** encadeadas, embora seja usualmente mais eficiente durante a execução.

A sintaxe desta diretiva é a seguinte:

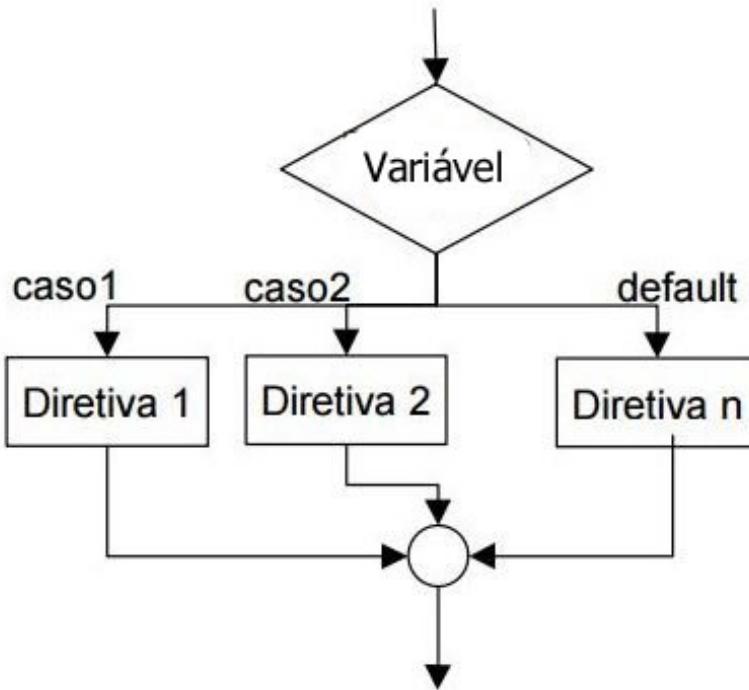
```
switch (variavel) {
    case valor1: diretiva1
        break
    case valor2: diretiva2
        break
    default: diretiva_default
}
```

As diretivas encontradas a partir do caso (**case**) escolhido são executadas até o final da diretiva **switch** ou até uma diretiva **break** que encerra o **switch/case**. Se o valor resultante não possuir um caso específico, é executado a diretiva default colocada, opcionalmente, ao final da diretiva **switch/case**.

O mesmo efeito obtido pela construção switch/case mostrada antes também pode ser realizado (para fins didáticos) apenas com ifs e elses:

```
if(variavel === valor1)
    diretiva1
else if(variavel === valor2)
    diretiva2
else
    diretiva_default
```

Em um fluxograma, temos a estrutura switch/case representada da seguinte maneira:



Note que o ponto de início de execução é um caso (**case**) cujo valor é aquele da variável avaliada. Após iniciada a execução do conjunto de diretivas identificadas por um certo caso, tais ações só são interrompidas com a execução de uma diretiva **break** ou com o final da diretiva **switch**.

A seguir temos uma aplicação simples que exemplifica a utilização das diretivas **switch/case** e **break**. Como de costume, sugiro criar um novo arquivo **index.js** dentro de uma pasta **ExemploSwitch** e abri-la no VS Code para trabalhar melhor:

Código 3.40: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```

let letra = 'A';
switch (letra) {
  case 'A': console.log('Vogal A'); break;
  case 'E': console.log('Vogal E'); break;
  case 'I': console.log('Vogal I'); break;
  case 'O': console.log('Vogal O'); break;
  case 'U': console.log('Vogal U'); break;
  
```

```
default: console.log('Não é uma vogal');  
}
```

Troque o valor da variável letra e veja os resultados no console. A saída esperada depende do valor que você colocar na variável 'letra', declarada logo no início do algoritmo.

Podemos ler a estrutura **switch/case** anterior da seguinte forma: "considerando a variável letra, caso seja A, imprima 'Vogal A', caso seja B, imprima 'Vogal B'...etc".

Arrays

Aprendemos nos tópicos anteriores como declarar variáveis e manipular os valores delas, certo?

Mas e quando o que precisamos armazenar em memória não é apenas um valor, mas diversos deles?

E se eu lhe pedir para guardar 10 números, como você faria?

Seguindo uma lógica recente que utilizamos, você declararia 10 variáveis, certo?

Mas e se começássemos com 10 números e esse valor tivesse de aumentar em tempo de execução? Como faria neste caso?

É aí que entram os arrays!

Um array (também chamado de vetor em algumas linguagens de programação) é um tipo especial de objeto que permite que a gente armazene diversos valores dentro dele, cada um em uma posição indexada dentro do array. Declarar um array é muito simples, basta inicializarmos uma variável qualquer como um par de colchetes:

Código 3.41: disponível em <https://www.luiztools.com.br/livro-node-fontes>

const numeros = [];

Aqui declaramos uma variável que é na verdade um array, demonstrado pelo uso de colchetes ([]). Essa declaração não indica qual o tipo de dado que armazenaremos nesse array, embora recomende-se que você armazene sempre dados de mesmo tipo em um mesmo array. Ao contrário de outras linguagens de programação, em JS os arrays não possuem tamanho fixo e nem mesmo precisam ser pré-dimensionados.

Calma, eu vou explicar.

Cada elemento que quisermos guardar em nosso array deve ficar em uma posição, começando na posição 0, depois 1, 2 e assim por diante. Sendo assim, se quisermos guardar o número 5 na primeira posição (0) faremos:

Código 3.42: disponível em <https://www.luiztools.com.br/livro-node-fontes>

numeros[0] = 5;

Agora se quisermos guardar o número 3 na segunda posição (1), fazemos:

Código 3.43: disponível em <https://www.luiztools.com.br/livro-node-fontes>

numeros[1] = 3;

Se por algum motivo quisermos substituir o valor contido em alguma das posições, basta definirmos o novo valor sobre ela, como neste caso, em que estamos substituindo o valor guardado na posição 1 do array por um novo valor:

Código 3.44: disponível em <https://www.luiztools.com.br/livro-node-fontes>

numeros[1] = 6;

Os elementos do array são numerados iniciando com zero (*zero-based*), e índices válidos variam de zero ao número de elementos menos um. O elemento do array com índice 1, por exemplo, é o segundo elemento no array. O número de elementos em um array é seu comprimento (*length*).

Resumindo: para guardar valores dentro do array, usamos o nome que demos a ele, seguido de colchetes e o número da posição onde queremos guardar o valor, como abaixo:

Código 3.45: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const numeros = [];
numeros[0] = 8;
numeros[1] = 4;
numeros[2] = 29;
```

Aqui temos um array de comprimento 3, onde guardamos valores nas posições 0 (a primeira), 1 (a segunda) e 2 (a terceira e última).

Agora outro exemplo, onde crio um array e populo com nomes de pessoas:

Código 3.46: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const inventores= [];
inventores[0] = 'Einstein';
inventores[1] = 'Edson';
inventores[2] = 'Galileu';
inventores[3] = 'Da Vinci';
```

Atenção: note que apesar de declarar a variável inventores com 'const' (constante) eu consigo adicionar elementos nela. Isso porque os elementos não são a variável inventores em si, mas estão sendo referenciados por ela. Sem entrar em muitos detalhes de baixo nível aqui (baixo nível = nível de máquina), basta entender que adicionar um item no array não muda o que o array é e por isso podemos declará-lo como const sem problemas. O que geraria um erro?

Tentar reinicializar inventores com um novo valor, pois daí estariamos mudando quem/o quê ele é.

Mais tarde, quando quisermos acessar os elementos do array, basta utilizarmos a sua posição (índice) para retornarmos o valor contido dentro dela:

Código 3.47: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const inventor = inventores[3];
console.log(inventor);
```

Considerando os exemplos anteriores de código, o que o trecho acima irá imprimir no console quando executado?

Da Vinci! Pois esse foi o nome do inventor armazenado na quarta posição do array (índice 3).

Os Arrays possuem algumas características e funções especiais que podemos utilizar para manipulá-los mais facilmente, como as abaixo (considere que arr é um variável Array):

- **arr.length**: retorna a quantidade atual de elementos do array
- **arr.push(x)**: adiciona o valor x ao final do Array
- **arr.splice(i)**: remove o elemento na posição i do array
- **arr.splice(i, y)**: remove y elementos a partir da posição i do array
- **arr.concat(arr2)**: faz a fusão dos elementos de arr2 (outro array) no final de arr
- **arr.indexOf(x)**: retorna a posição do elemento x dentro de arr, ou -1 caso ele não exista dentro de arr
- **arr.forEach(function(x){})**: percorre todas as posições do array, da primeira à última, e à cada iteração executa a função passada por parâmetro onde x representa o item da iteração atual.

Mas voltando ao desafio inicial deste tópico, como faríamos para armazenar um número arbitrário de valores?

Você lembra dos laços de repetição?

Uma maneira muito inteligente de popular arrays é com laços de repetição, como um **for**, por exemplo.

Código 3.48: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const numeros = [];
for(let i=0; i < 10; i++)
    numeros[i] = 1;
```

Note que usamos a própria variável de controle ‘i’ para acessar a posição do array de números onde queremos guardar os valores.

Um código equivalente, mas usando a função 'push' seria:

Código 3.49: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const numeros = [];
for(let i=0; i < 10; i++)
    numeros.push(1);
```

Atenção: reparou que usei let para a variável de controle do for?
Isso porque não posso usar const, uma vez que o valor dela altera a cada iteração. Podem parecer coisas bobas e você até pode se sentir tentado a só usar var, mas recomendo fazer o esforço de usar corretamente const/let/var nessa ordem pois vale a pena a longo prazo.

Notou algo estranho? Sim, eu guardei o número 1 em todas posições do array.

Não acredita?

Use o seguinte código para imprimir todas posições do array:

Código 3.50: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
for(let i=0; i < numeros.length; i++)  
  console.log(numeros[i]);
```

Note que aqui eu usei a propriedade length acessada a partir do array de números. Eu citei ela nas características e funções úteis do array e essa é a mesma propriedade existente no tipo String que vimos anteriormente. Hmm, por que você acha que Strings e Arrays possuem uma propriedade length em comum?

Porque ambos são Arrays!

Sim, a diferença é que Strings apenas guardam caracteres dentro de si, mas ambos possuem muitas características em comum, muito mais do que o length em si. Ou seja, considerando que str é uma variável String:

- **str.length**: retorna a quantidade atual de caracteres da String
- **str.split('x')**: retorna um array de Strings quebrando a original pelo caracter (ou palavra) informado na função;
- **str.charAt(i)**: retorna o caracter na posição informada na String (zero-based)
- **str.slice(start,end)**: retorna uma substring da original começando na posição start e terminando na posição end
- **str.indexOf('x')**: retorna a posição do caracter (ou palavra) x dentro de str, ou -1 caso ele não exista dentro de str

Muito legal, não?!

O tipo Object

Tudo é tratado como objeto em JavaScript, mesmo os números e textos literais. Além disso, os objetos em JavaScript são extremamente dinâmicos como você já deve ter percebido, podendo alterar o seu conteúdo livremente.

No entanto, existe uma forma de criar seus próprios objetos em Javascript, embora ela não seja uma linguagem orientada à objetos da mesma forma que Java e C#, por exemplo. Objetos JS são declarados como qualquer outra variável (embora exista a possibilidade de criá-los a partir de classes), os atributos e métodos podem ser definidos no mesmo momento em que são lidos e/ou escritos.

Sendo assim, para criar um objeto considere a seguinte declaração de variável:

Código 3.51: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const cliente = {};
```

Nada de especial, certo? Mas o que acontece se na linha seguinte fizermos o seguinte:

Código 3.52: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
cliente.nome = 'Luiz';
```

No Java e outras linguagens compiladas e tipadas isso daria errado, certo?

Mas no JS não.

Simplesmente será criado um atributo nome dentro do objeto cliente contendo o valor da string 'Luiz'. Prático, não?! Quando digo que cliente recebe as chaves, estou dizendo que cliente é um object (objeto complexo), e poderei colocar dentro dele atributos e funções.

Note que se eu mandasse imprimir no console o nome deste cliente, antes do mesmo ser definido, o atributo nome não existiria e com isso teríamos undefined como resultado.

Assim como definimos atributos conforme vamos precisando deles, também o fazemos com funções. Assim, podemos ter instruções "estranhas" como esta:

Código 3.53: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
cliente.exibirMensagem = function(msg){  
    console.log(msg);  
}
```

Ou seja, a partir da linha seguinte podemos chamar cliente.exibirMensagem('teste') que irá disparar um console.log com a mensagem teste dentro.

Quando queremos criar um objeto já com seus atributos e funções pré-definidos podemos utilizar a notação JSON. JSON é uma sigla para JavaScript Object Notation e registra um padrão de escrita e leitura de objetos. Um objeto cliente pode ser definido em JSON da seguinte maneira:

Código 3.54: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const cliente = {  
    nome: 'Luiz',  
    saldo: 100.0,  
    idade: 29,  
    gaucho: true  
}
```

Assim, a partir da linha seguinte à este trecho JS, se mandarmos um console.log com cliente.saldo, irá aparecer o valor '100.0' no navegador.

E para criar arrays de objects (algo perfeitamente possível), podemos fazer como segue:

Código 3.55: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const clientes = [cliente1, cliente2, { nome:"cliente3", saldo:5.0,  
idade:21, gaúcho:false }]
```

Nota: o padrão JSON (<http://json.org>) define que o nome dos atributos deve estar entre aspas duplas. Entretanto, o JS entende os atributos que não possuam aspas, desde que os mesmos não possuam espaços ou caracteres especiais.

No próximo capítulo vamos incrementar nossos conhecimentos de programação para web!

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

O que é JavaScript?

Neste vídeo eu explico um pouco da história da linguagem de programação mais popular do mundo atualmente.

<https://www.youtube.com/watch?v=jLU1p0vu4QA>

Revisão de JavaScript

Neste vídeo eu passo pelos principais aspectos da linguagem JavaScript que você deve conhecer, na prática.

https://www.youtube.com/watch?v=s5YAGR_kS2I

Classes em JavaScript

Aprenda a criar classes em JavaScript para ajudar a organizar melhor os seus códigos.

<https://www.luiztools.com.br/post/como-criar-classes-em-javascript-es6-e-node-js/>

Dicas e truques do JavaScript

15 dicas e truques sensacionais da linguagem de programação mais popular do mundo.

<https://www.luiztools.com.br/post/15-dicas-e-truques-da-linguagem-javascript/>

Operador spread

4 segredos do operador spread em JavaScript

<https://www.luiztools.com.br/post/4-segredos-do-operador-spread-em-javascript/>

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

4 A plataforma Node.js

The most important property of a program is whether it accomplishes the intention of its user.
- C.A.R. Hoare

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.

História do Node.js

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

Suas pesquisas nessa área levaram-no a criticar as possibilidades limitadas do servidor web Apache de lidar (em 2009) com conexões concorrentes e a forma como se criava código web server-side na época que bloqueava os recursos do servidor web a todo momento o que fazia com que eles tivessem de criar diversas stacks de tarefas em caso de concorrência para não ficarem travados, gerando um grande overhead.

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

Em 2011 a Microsoft ajudou o projeto criando a versão Windows de Node.js, lançando-a em julho deste ano e daí em diante nunca mais

parou de crescer, atualmente sendo mantido pela Node.js Foundation, uma organização independente e sem fins lucrativos que mantém a tecnologia com o apoio da comunidade mundial de desenvolvedores.

Características do Node.js

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

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

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

Isso é diferente do funcionamento tradicional da maioria das linguagens de programação, que trabalham com o conceito de multi-threading, onde, para cada requisição recebida, cria-se uma nova thread para atender à mesma. Isso porque a maioria das linguagens tem comportamento bloqueante na thread em que estão, ou seja, se uma thread faz uma consulta pesada no banco de dados, a thread fica travada até essa consulta terminar.

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

Node.js não é uma linguagem de programação.

Você programa utilizando a linguagem Javascript, a mesma usada há décadas no client-side das aplicações web. Javascript é uma linguagem de scripting interpretada, embora seu uso com Node.js guarde semelhanças com linguagens compiladas, uma vez que

máquina virtual V8 (veja mais adiante) faz etapas de pré-compilação e otimização antes do código entrar em operação.

Node.js não é um framework Javascript.

Ele está mais para uma plataforma de aplicação (como um Nginx?), na qual você escreve seus programas com Javascript que serão compilados, otimizados e interpretados pela máquina virtual V8.

Essa VM é a mesma que o Google utiliza para executar Javascript no browser Chrome, e foi a partir dela que o criador do Node.js, Ryan Dahl, criou o projeto. O resultado desse processo híbrido é entregue como código de máquina server-side, tornando o Node.js muito eficiente na sua execução e consumo de recursos.

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

Node.js serve para fazer APIs.

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

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

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

Node.js serve para fazer aplicações de tempo real.

Usando algumas extensões de web socket com Socket.io, Comet.io, etc é possível criar aplicações de tempo real facilmente sem onerar

demais o seu servidor como acontecia antigamente com Java RMI, Microsoft WCF, etc.

Express

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

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

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

```
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 4.2: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
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 4.3: disponível em <https://www.luiztools.com.br/livro-node-fontes>

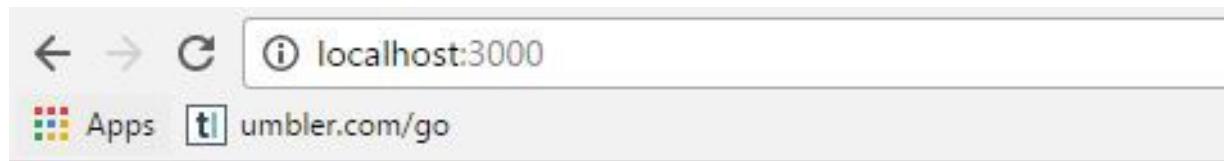
```
cd workshop
npm install
```

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

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

npm start

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



Express

Welcome to Express

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 4.5: disponível em <https://www.luiztools.com.br/livro-node-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 4.6: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
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:

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

```
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 4.8: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
// 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\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 4.9: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
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 4.10: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
// 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.

Código 4.11: disponível em <https://www.luiztools.com.br/livro-node-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 4.12: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
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 4.13: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
// 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:

Código 4.14: disponível em <https://www.luiztools.com.br/livro-node-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>
```

Ainda não aprendemos HTML, coisa que veremos no próximo capítulo, mas 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 4.15: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

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

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

- crie o arquivo da nova rota em 'routes';
- para programar a rota, use como exemplo o routes/index.js;

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

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

Código 4.16: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
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:

Código 4.17: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
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. A explicação do

porque isso funciona encontra-se no último bloco **Atenção**, dá uma lida lá se você ainda não leu.

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

Event Loop

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

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

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

O Problema

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

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

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

nesse tempo uma CPU de 2GHz consegue executar 28.000 instruções.

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

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

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

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

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

É muita diferença!

A solução

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

Esse tipo de comportamento pode ser alcançado de diversas maneiras. Atualmente a forma mais comum de fazer isso é através do uso de threads o que geralmente torna nosso código muito mais complexo. Por exemplo, ler um arquivo em Java é uma operação

bloqueante, ou seja, seu programa não pode fazer mais exceto esperar a comunicação com a rede ou disco terminar. O que você pode fazer é iniciar uma thread diferente para fazer essa leitura e mandar ela avisar sua thread principal quando a leitura terminar.

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

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

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

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

Vamos dar uma olhada em uma aplicação Node.js comum - um servidor escutando em localhost:3000. Após receber a requisição, o servidor vai ler um arquivo para obter um trecho de texto e imprimir algumas mensagens no console e depois retorna a resposta HTTP.

Código 4.18: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

Task/Event/Message Queue

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

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

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

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

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

E é assim que tudo funciona!

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

Microtasks e Macrotasks

Além disso, como se não fosse o bastante, nós temos duas task queues, não apenas uma. Uma task queue para microtasks e outra para macrotasks, que sem entrar em detalhes, vou deixar alguns exemplos.

Exemplos de microtasks

- process.nextTick
- promises
- Object.observe

Exemplos de macrotasks

- setTimeout
- setInterval
- setImmediate
- I/O

O conceito do event loop pode ser um tanto complicado no início mas uma vez que você entender seu funcionamento na prática você não conseguirá mais imaginar sua vida sem ele. Obviamente o uso inicial intenso de callbacks é muito chato de gerenciar mas já é possível usar Promises em nossos códigos Javascript que permitem deixar as tarefas assíncronas mais legíveis e também o recurso `async-await` do ES7.

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

O que é Node.js?

Neste vídeo, parte do meu curso online, eu lhe explico o que é Node.js

<https://www.youtube.com/watch?v=irrkH6VII78>

Programação assíncrona em Node.js

Aprenda as diferenças entre callbacks e promises e como usar este segundo padrão de codificação em Node.js.

<https://www.luiztools.com.br/post/programacao-assincrona-em-nodejs-callbacks-e-promises/>

package.json

Aprenda tudo que você deve saber sobre o package.json em Node.js.

<https://www.luiztools.com.br/post/o-guia-completo-do-package-json-do-node-js/>

express-generator

Vídeo bem curto que gravei sobre o express-generator para o meu canal.

<https://www.youtube.com/watch?v=BQAljFOTlvw>

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

5 Front-end: HTML

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

— Alan J. Perlis

Em Ciência da Computação, front-end e back-end são termos generalizados que se referem às etapas inicial e final de um processo. O front-end é responsável por coletar a entrada do usuário em várias formas e processá-la para adequá-la a uma especificação em que o back-end possa utilizar.

Resumindo, o front-end é a camada de apresentação, que o usuário consegue ver e interage, muitas vezes sendo considerada a principal camada para ele que não costuma distinguir o que vê do que é o "sistema web de verdade". Cabe ao profissional de front-end projetar a experiência do usuário, a identidade visual do sistema e muitas vezes acaba fazendo papel de web-designer e de programador ao mesmo tempo.

Este não é um livro de web design, de user experience ou sequer de front-end, mas existem uma série de conceitos básicos que todo programador web deve saber (mesmo que ele se especialize no back-end), e é isso que exploraremos aqui. Caso já possua conhecimentos básicos de HTML, dê uma olhada rápida neste capítulo, mas evite ignorá-lo.

Quando falamos de programação web, existem três principais tecnologias que são utilizadas e que devemos aprender, acima de todas as outras: HTML, CSS e JavaScript. Falamos de JavaScript de maneira genérica anteriormente e voltaremos a abordar ele mais pra frente. CSS também será visto mais adiante, enquanto neste capítulo falaremos de HTML.

Nota: deixaremos o Node.js um pouco de lado neste início de capítulo para explorar somente o HTML de maneira isolada. Mais pra frente farei o gancho com o capítulo anterior novamente, então não se preocupe. Para os exemplos de código HTML, recomendo continuar usando o Visual Studio Code.

Introdução ao HTML

HTML (abreviação para a expressão inglesa HyperText Markup Language, que significa Linguagem de Marcação de Hipertexto) é uma linguagem de marcação utilizada para produzir páginas na Web. Documentos HTML podem ser interpretados por navegadores, como o Google Chrome, que pedi que você instalasse no capítulo 2.

Documentos HTML podem ser qualquer coisa que queiramos que seja exibida no browser de nossos usuários: a timeline do Facebook, a tela de pesquisa do Google ou a home-page da Microsoft. Todos são documentos HTML. Com exceção de telas criadas através de plug-ins, como o Flash (obsoleto), os documentos HTML são a cara da web e é de suma importância que entendamos a sua estrutura básica antes de começar a programar pra ela com Node.js.

Todo documento HTML é um arquivo de texto com a extensão .html (ou .htm, tanto faz). Pode ser criado e editado em qualquer editor de texto, desde que salvo com a extensão correta. Mas quando aberto no navegador, é que vemos o que o documento representa de verdade.

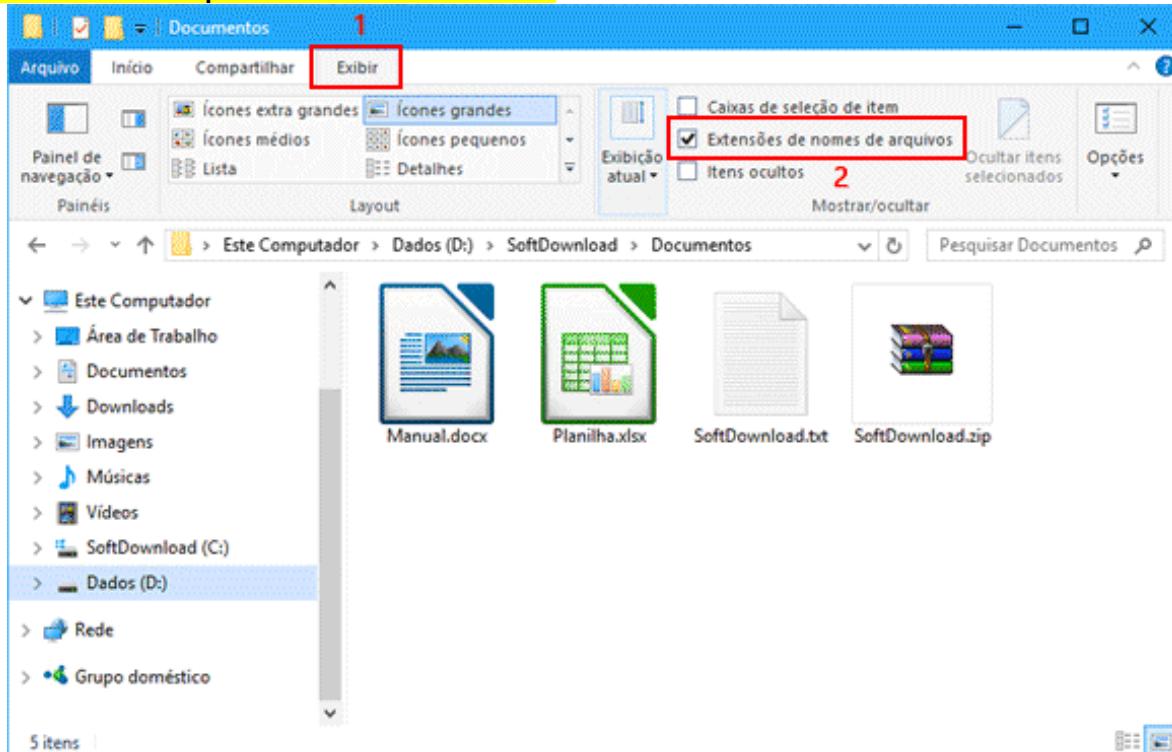
Isso porque todo documento HTML é composto de informações e de meta-informações. Estas últimas são organizadas em tags (rótulos) e cada tag possui uma propriedade especial no documento, indo desde alterar o tamanho ou cor de algo, até o seu posicionamento e comportamento.

Vamos começar simples, abra um editor de texto qualquer (podendo inclusive ser o VS Code) e digite o seguinte texto:

Olá Mundo

Depois salve com o nome index e a extensão '.html'.

Dica: No Windows, caso não consiga alterar a extensão (o notepad teima em salvar como .txt por padrão) pode ser necessário alterar a opção de 'Extensões de nomes de arquivos no menu Exibir, do Windows Explorer como abaixo.



Agora dê um duplo-clique neste arquivo ou mande abrir com o Google Chrome. Você verá o seu arquivo de texto no browser.

Note que não haverá qualquer diferença em relação ao texto originalmente escrito, talvez apenas a tipografia ligeiramente diferente. Isso porque colocamos apenas informações, mas nenhuma meta informação no arquivo, que a partir de agora chamaremos simplesmente de tags.

Cada tag é escrita usando colchetes angulares (<>), também chamados de símbolo "menor-maior". Dentro dos símbolos de menor e maior temos o nome da tag. Ao escrever uma tag no documento HTML, temos de seguir o seguinte formato abre-efecha:

```
<NOME_DA_TAG>conteúdo</NOME_DA_TAG>
```

Ou nesse outro formato auto-fecha:

```
<NOME_DA_TAG />
```

Note que a barra ‘/’ serve para demonstrar o fechamento da tag. No primeiro formato (abre-e-fecha) temos uma tag de container, que pode ter conteúdo dentro do seu interior. Este conteúdo pode ser um texto, ou uma outra tag por exemplo.

No segundo formato (auto-fecha), a tag não permite conteúdo, sendo apenas uma marcação isolada, podendo definir que naquele local do documento teremos a renderização de algo especial e não apenas informação pura.

Veremos tags de ambos tipos com detalhes, mas vale salientar desde já que toda tag que abre, tem de fechar. Ou ela fecha com uma tag de mesmo nome precedida de barra ‘/’ (caso abre-fecha) ou ela fecha em si mesmo, com uma barra ‘/’ antes de ‘>’

E por fim, algumas tags possuem atributos, ou seja, características personalizáveis em seu interior.

Atenção: por questões de implementação de cada um dos browsers de Internet, algumas tags podem ter seu efeito visual ligeiramente diferente, o que é normal. Além disso, alguns browsers são mais displicentes com relação à formação do documento HTML, ou seja, alguns aceitam formatações errôneas como tags que não fecham ou tags dentro de outras tags que não são containers. Isso não deve ser usado como uma desculpa para escrever HTML errado. Sempre que possível, siga os padrões de formatação de documentos HTML para garantir o máximo de compatibilidade entre os browsers. Todos estes padrões podem ser encontrados no site da W3C.

As tags HTML

A primeira e mais importante tag que você deve conhecer é a tag HTML. Ela inicia e termina o documento HTML, ou seja, é a primeira e a última presente em um documento, uma tag contâiner que engloba todas as demais tags do documento, como segue:

Código 5.1: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<html>
... o conteúdo vai aqui ...
</html>
```

Sendo uma tag contâiner, ela permite dentro dela tanto conteúdo textual quanto outras tags, no entanto, como manda o padrão W3C, a tag HTML deve conter em seu interior essencialmente apenas outras duas tags: head e body, como veremos a seguir.

Além da tag HTML, sugere-se atualmente começar seu documento com a tag DOCTYPE, indicando que usaremos HTML5, a especificação mais recente da linguagem na data de escrita deste livro.

Código 5.2: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
... o conteúdo vai aqui ...
</html>
```

A tag HEAD

Todo documento HTML possui um cabeçalho, com informações e propriedades gerais do documento e essa é a função da tag contâiner HEAD. Ela deve vir imediatamente após a tag <html> e

deve conter em seu interior apenas algumas tags especiais mas nenhum conteúdo solto. Como segue:

Nota: tabulações e quebras de linha não surtem qualquer efeito na apresentação do seu documento HTML no browser do usuário, então use-as para se organizar melhor, como faço abaixo quebrando a linha entre as tags e tabulando as que estão dentro de outras.

Código 5.3: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head> ... cabeçalho aqui ... </head>
  ... conteúdo aqui ...
</html>
```

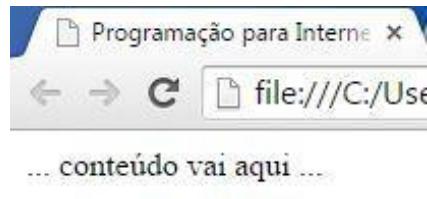
Dentro da tag HEAD podemos colocar, por exemplo, as tags TITLE, SCRIPT e STYLE além de tags para ajudar os mecanismos de busca, chamadas de meta-tags.

A tag TITLE denota o título deste documento e aparece na aba do seu navegador, indicando o que o usuário está vendo no momento. Já as tags SCRIPT e STYLE serão vistas mais pra frente para inserir Javascript e CSS no documento, respectivamente. A tag TITLE é um contâiner que permite colocar uma linha de texto entre seu abre-efecha com o título da página, veja abaixo e compare com o resultado:

Código 5.4: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
  </head>
```

... o conteúdo vai aqui ...
</html>



Note na primeira imagem o código HTML e na segunda imagem o resultado no navegador. Agora entendeu a diferença entre informação e meta-information (tag)? A tag HEAD define que o conteúdo em seu interior é propriedade do documento, como seu título, que é colocado na aba do navegador. Já o que está fora da HEAD é interpretado como texto plano e é impresso diretamente na página.

Atenção: caso tenha problemas com acentos na sua página você pode utilizar uma tag no HEAD para dizer ao navegador qual a codificação de caracteres que está utilizando, da seguinte forma: <meta charset="UTF-8" /> neste caso, UTF-8. Outra codificação é ISO-8859-1.

A tag BODY

Assim como temos um cabeçalho no documento HTML, também precisamos ter um corpo, função da tag BODY, que deve vir logo após a tag HEAD, mas fechar antes da tag fecha-HTML como segue:

Código 5.5: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
```

```
</head>
<body>
... o conteúdo vai aqui ...
</body>
</html>
```

A tag BODY é uma tag container e aceita a maior parte das tags HTML existentes, pois ela representa o documento em si, o que o usuário vê e interage na tela. É dentro da tag body que construiremos as interfaces de nossos sistemas web.

A tag BODY possui alguns atributos que permitem sua personalização. É incentivado que procure conhecer estes atributos, embora desnecessário no momento para o aprendizado de desenvolvimento de sistemas para Internet. Entenda apenas que as tags apresentadas a seguir devem ser inseridas dentro do BODY do documento HTML, mesmo que não seja citado isso explicitamente.

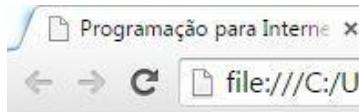
As tags H1, H2, ...

Um conjunto de tags iniciados com H são as tags de títulos e subtítulos de textos, containers que recebem texto plano em seu interior. Não confunda com a tag TITLE, que não aparece visivelmente na página web, as tags H aparecem sim e de forma bem chamativa, sendo a H1 a maior delas e quanto maior o número depois do H, menor seu tamanho, como mostra a imagem abaixo:

Código 5.6: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Teste</h1>
```

```
<h2>Teste</h2>
<h3>Teste</h3>
<h4>Teste</h4>
</body>
</html>
```



Teste

Teste

Teste

Teste

Note que houve quebra de linha entre os testes. Isso NÃO é devido à ter quebra de linha entre as tags H no HTML, mas sim porque as tags H possuem comportamento de bloco, ou seja, não permitem elementos ao seu lado, na mesma linha.

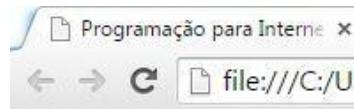
As tags P, BR e HR

Eu falei anteriormente sobre o comportamento de bloco das tags H. Mas não é só elas que “quebram linha” após seu conteúdo. Existem algumas tags especiais que também permitem fazer isso de uma maneira, digamos, mais controlada.

Sempre que queremos definir um parágrafo de texto em um documento HTML, com espaçamento antes e depois do parágrafo, usamos a tag P, que é um contâiner onde devemos colocar o texto a ser formatado em seu interior.

Código 5.7: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Teste</h1>
    <p>Parágrafo 1</p>
    <p>Parágrafo 2</p>
    <p>Parágrafo 3</p>
  </body>
</html>
```



Teste

Parágrafo 1

Parágrafo 2

Parágrafo 3

Note que a tag **P** não influencia no estilo do texto, apenas dá uma aparência de parágrafo, com espaçamentos antes e depois, para facilitar a leitura.

Atenção: a tag P deve ser usada para definir blocos de texto e somente texto. Para outros elementos veremos as tags DIV e SPAN mais à frente.

Já a tag **BR** apenas quebra a linha do documento exatamente naquele ponto, sem praticamente qualquer espaçamento. Note também no exemplo abaixo que a tag BR não é um contâiner, ou seja, ela não aceita conteúdo em seu interior. O browser apenas entende que quando encontrar uma tag BR ele deve mostrar um

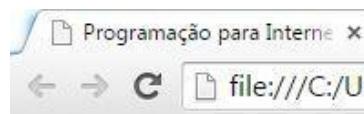
break-line no lugar (o nosso \n das linguagens de programação desktop).

Talvez você fique na dúvida de quando usar P e quando usar BR. A regra é simples: blocos de texto devem ser organizados usando tags P. Caso dentro de um bloco de texto haja a necessidade de quebrar linha, aí sim usamos a tag BR.

Atenção: a altura da quebra de linha feita com BR não é padronizado, diferenças podem ocorrer entre os browsers e sistemas operacionais.

Código 5.8: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Tópico 1</h1>
    <p>Introdução ao HTML</p>
    Linha 1<br />
    Linha 2<br />
  </body>
</html>
```



Tópico 1

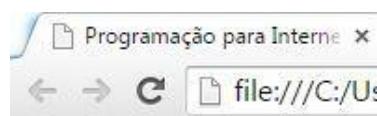
Introdução ao HTML

Linha 1
Linha 2

E por fim, a tag **HR** é como se fosse uma BR, mas além de realizar a quebra ainda coloca uma linha contínua naquele ponto, o que pode ser útil em alguns casos.

Código 5.9: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
<head>
  <title>Programação para Internet</title>
  <meta charset="utf-8" />
</head>
<body>
  <h1>Tópico 1</h1>
  <p>Introdução ao HTML</p>
  Linha 1
  <hr />
  Linha 2
</body>
</html>
```



Tópico 1

Introdução ao HTML

Linha 1

—
Linha 2

As tags B, STRONG, I e U

Assim como a tag P que formata um bloco de texto, existem outras tags que alteram a aparência do mesmo. As tags **B** e **STRONG** por exemplo, tornam o texto negrito (bold), sendo a segunda mais

utilizada quando queremos dizer aos mecanismos de busca que essa palavra é importante para nosso site.

Já as tags I e U fazem o mesmo para itálico (italic) e sublinhado (underline), todas contâiners, que devem receber em seu interior o texto a ser formatado, como segue:

Código 5.10: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Tópico 1</h1>
    <p>Introdução ao HTML</p>
    <b>Texto negrito</b>
    <i>Texto itálico</i>
    <u>Texto sublinhado</u>
  </body>
</html>
```



Tópico 1

Introdução ao HTML

Texto negrito *Texto itálico* Texto sublinhado

Note que os textos ficarão lado-a-lado. Isso porque as tags B, STRONG, I e U não são blocos de texto, mas apenas estilizações. Afinal, você não iria querer que no mesmo do seu texto a sua palavra sublinhada quebrasse todo o parágrafo, certo?

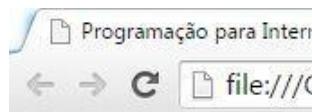
As tags UL, OL e LI

Quando queremos listar elementos, seja de texto ou de qualquer outra coisa (imagens, por exemplo), usamos as tags de lista **UL** (unordered list, lista desordenada) e **OL** (ordered list). Ambas listas são containers que aceitam somente tags **LI** dentro do seu interior, estas por sua vez também são containers que aceitam praticamente qualquer coisa no seu interior.

A diferença principal entre a lista desordenada (UL) e a ordenada (OL) não tem a ver com a ordem dos elementos, mas sim com o bullet, a marcação que vai antes de cada item, que no primeiro caso é um ponto preto e no segundo caso uma numeração crescente, como segue:

Código 5.11: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Tópico 1</h1>
    <p>Introdução ao HTML</p>
    <ul>
      <li><b>Texto negrito</b></li>
      <li><i>Texto itálico</i></li>
      <li><u>Texto sublinhado</u></li>
    </ul>
  </body>
</html>
```



Tópico 1

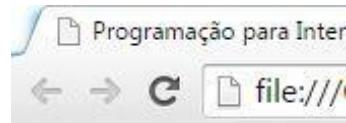
Introdução ao HTML

- **Texto negrito**
- *Texto itálico*
- Texto sublinhado

No exemplo acima usei uma UL, note que dentro dela somente existem tags LI, mas que dentro das tags LI coloquei uma combinação de texto e tags de formatação. O mesmo exemplo trocando a tag UL por OL pode ser visto abaixo, com seu efeito visível:

Código 5.12: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Tópico 1</h1>
    <p>Introdução ao HTML</p>
    <ol>
      <li><b>Texto negrito</b></li>
      <li><i>Texto itálico</i></li>
      <li><u>Texto sublinhado</u></li>
    </ol>
  </body>
</html>
```



Tópico 1

Introdução ao HTML

1. **Texto negrito**
2. *Texto itálico*
3. Texto sublinhado

Note também que cada LI (list item) é colocado em uma linha separada, pois é um novo item da lista, o que nos leva a entender que as tags UL, OL e LI são tags de bloco, ou seja, quebram linha e garantem espaçamento automático antes e depois.

A tag A

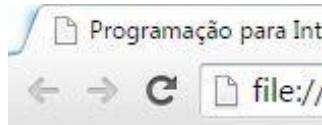
A tag A é o que chamamos de âncora (anchor no original), mais conhecida popularmente como “link” (ligação), isso porque utilizamos tags A quando queremos levar o usuário de um ponto da web a outro, o que chamamos de hyperlink (em analogia ao hypertexto).

Falando de maneira prática, cada tag A é composta principalmente de duas partes: a URL de destino e o conteúdo do link, que pode ser textual ou gráfico. A URL deve ser colocada dentro do atributo href da âncora (hyperreference), enquanto que o conteúdo deve ficar entre as tags de abre-e-fecha da âncora (sim, ela é um contâiner). Como segue:

Código 5.13: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
```

```
</head>
<body>
  <h1>Tópico 1</h1>
  <p>Introdução ao HTML</p>
  <a href="http://www.google.com">Clique aqui</a>
</body>
</html>
```



Tópico 1

Introdução ao HTML

[Clique aqui](#)

Note que o conteúdo da tag A (a frase “Clique aqui”) é o texto que aparece na página, enquanto que o atributo href define para onde o usuário será levado no caso dele clicar no link, o que neste caso é o site do Google. Isto é o básico que você precisa saber sobre âncoras HTML.

Além do básico, existem ainda dois atributos que podem ser úteis às suas âncoras: o atributo title, que define a dica da âncora (quando você fica com o mouse parado sobre ela, a dica aparece) e o atributo target, que permite a você definir se ao ser clicado o link levará o usuário para outra aba ou permanecerá na mesma. Os valores possíveis para o atributo target são: _blank e _self

Código 5.14: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
```

```
</head>
<body>
  <h1>Tópico 1</h1>
  <p>Introdução ao HTML</p>
  <a href="http://www.google.com" title="Ir ao Google"
target="_blank">Clique aqui</a>
</body>
</html>
```

No exemplo acima, levemente modificado da versão anterior, colocamos um title “Ir ao Google” e um target “_blank”, que fará com que uma nova aba se abra no navegador com o endereço do Google quando o usuário clicar neste link. Visualmente nenhum desses atributos mudará a aparência da âncora original, que mantém-se com o texto “Clique aqui” e a cor azul escuro sublinhado, denotando que é um hyperlink.

Embora o mais comum seja encontrarmos links textuais pela Internet, sendo a tag A um contâiner, ela permite agregar outras tags em seu interior, como tags de imagens, por exemplo, ou determinadas áreas do site como DIVs (a seguir).

Atenção: o atributo href aceita links absolutos e links relativos. Links absolutos são aqueles que começam com um protocolo (geralmente <http://>) e geralmente levam a outros sites da Internet. Links relativos começam com um nome de arquivo, pasta ou apenas uma ‘/’ e fazem referência a um recurso existente no mesmo domínio/site. Ou seja, se tivermos um href="<http://www.google.com>" quer dizer que o link lhe levará ao site do Google. Agora se tivermos um href="[index.html](#)" o link nos levará ao arquivo index.html no site atual.

A tag IMG

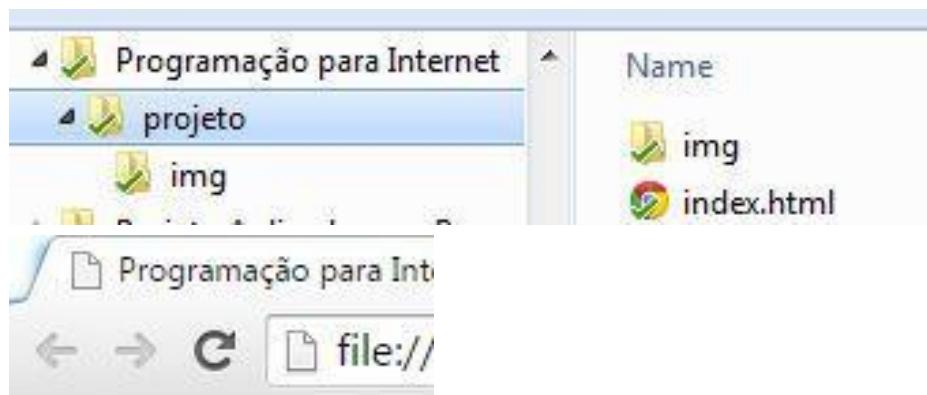
O nome IMG vem de IMAGE e denota a tag que representa uma imagem na página web. Esta tag NÃO é um contâiner, ou seja, ela abre-efecha a tag nela mesma, e representando uma imagem seu

atributo mais importante é o SRC (source, origem em inglês) que deve conter um endereço (absoluto ou relativo) de um arquivo de imagem (geralmente JPEG, GIF ou PNG).

Código 5.15: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Tópico 1</h1>
    <p>Introdução ao HTML</p>
    
    <a href="http://www.google.com" title="Ir ao Google"
target="_blank">Clique aqui</a>
  </body>
</html>
```

Por questões de organização costumamos guardar as imagens em uma subpasta do projeto, para separá-las das páginas HTML e demais arquivos do projeto. Nomes comuns para este subpasta incluem IMG (tããão criativo), imagens, images e content. Crie uma pasta com o nome img ao lado do seu arquivo HTML e dentro coloque uma imagem qualquer com o nome de imagem.jpg, que no meu exemplo abaixo, é um carro.



Tópico 1

Introdução ao HTML



Podemos também criar imagens com links. Para isso, basta colocar uma tag IMG contendo a sua imagem dentro de uma tag A, com o endereço do link. Assim, teremos uma imagem que quando clicada leva o usuário à outra página web.

Código 5.16: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Tópico 1</h1>
    <p>Introdução ao HTML</p>
    <a href="http://www.mitsubishimotors.com.br" title="Ir à
Mitsubishi" target="_blank">
```

```

</a>
</body>
</html>
```

Outros atributos da tag IMG incluem ALT, que é uma legenda para a imagem (não visível), WIDTH (que representa sua largura, sendo por padrão a largura original da imagem) e HEIGHT (que representa sua altura. Caso apenas WIDTH seja especificado, a altura será redimensionada proporcionalmente e vice-versa. Note que isso não afeta o tamanho real da imagem, apenas o visual.

Atenção: procure usar caminhos absolutos apenas de imagens existentes na Internet e ainda assim, considere a hipótese de salvar as imagens localmente para poder utilizar caminhos relativos, mais confiáveis.

As tags DIV e SPAN

As tags DIV e SPAN são tags contâiner (ou seja, permitem que sejam colocadas outras tags em seu interior) utilizadas para organizar e formatar as áreas da sua página HTML, sendo que a tag DIV organiza uma área em bloco (in-block, ou seja, quebra a linha automaticamente) e a tag SPAN organiza uma área em linha (in-line, ou seja, não quebra a linha automaticamente). Por ora é o que precisamos saber.

Por exemplo, o seguinte layout abaixo, com 3 seções verticais: na primeira temos um título e parágrafo de texto, na segunda uma imagem e na terceira um rodapé.

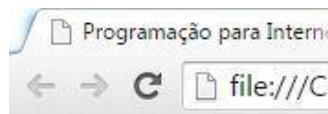
Código 5.17: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
```

```

</head>
<body>
  <div>
    <h1>Tópico 1</h1>
    <p>Introdução ao HTML</p>
  </div>
  <div>
    <a href="http://www.mitsubishimotors.com.br" title="Ir à
Mitsubishi" target="_blank">
      
    </a>
  </div>
  <div><i>Apenas um rodapé</i></div>
</body>
</html>

```



Tópico 1

Introdução ao HTML



Apenas um rodapé.

A especificação HTML diz que as tags SPAN e DIV somente devem ser usadas quando nenhuma outra tag HTML possui a função que desejamos.

As tags TABLE, TR e TD

Vimos na seção anterior que usamos DIVs e SPANS para organizar as seções da nossa página HTML. Entretanto, às vezes as informações de uma seção está organizada em linhas e colunas, como uma tabela Excel, de uma maneira que ficaria extremamente

diffícil (e trabalhoso) de fazer tudo usando DIVs (imagine uma DIV por célula da tabela). Nestes casos onde temos dados tabulares devemos usar a tag TABLE.

A tag TABLE é uma tag container que somente aceita em seu interior tags TR (table row ou linha da tabela). As tags TR, por sua vez, também são containers, mas que aceitam somente tags TD (table data ou dado da tabela, entenda como célula) em seu interior. Já as tags TD são containers que aceitam qualquer coisa em seu interior.

Assim, seguindo esta hierarquia de containers, podemos recriar a aparência de qualquer tabela que necessitarmos, como a abaixo:

Tópico 1

Introdução ao HTML

	A	B
1	Nome	RA
2	Luiz	123
3	João	456
4	Maria	789
-		

Nome RA	
Luiz	123
João	456
Maria	789

Usando o código abaixo. Atente ao uso da tag B para tornar o texto do cabeçalho negrito.

Código 5.18: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
```

```
<body>
  <div>
    <h1>Tópico 1</h1>
    <p>Introdução ao HTML</p>
  </div>
  <table>
    <tr>
      <td><b>Nome</b></td>
    <td><b>RA</b></td>
    </tr>
    <tr>
      <td><b>Luiz</b></td>
      <td><b>123</b></td>
    </tr>
    <tr>
      <td><b>João</b></td>
      <td><b>456</b></td>
    </tr>
    <tr>
      <td><b>Maria</b></td>
      <td><b>789</b></td>
    </tr>
  </table>
</body>
</html>
```

Atenção: apesar de parecer uma boa ideia, evite utilizar tabelas para organizar as seções da sua página HTML, pois elas não foram criadas com este propósito. O ideal é sempre utilizarmos DIVs e SPANS para essa finalidade, deixando as tabelas para a apresentação de dados tabulares.

Formulários HTML

Quando falamos de programação para a web não temos como fugir da criação de formulários, tal qual na programação tradicional (desktop). A especificação HTML possui capítulos específicos para tratar disso e abaixo você confere um resumo das regras de criação e tags específicas de formulários HTML.

A tag FORM

Quando queremos criar um formulário na nossa página HTML devemos utilizar a tag FORM. A tag FORM é um contâiner que indica que todas as demais tags e conteúdos no seu interior representam um único formulário HTML, por exemplo, um formulário de cadastro, de contato ou login.

Dentro de um form, além das tags HTML comuns, que ajustam a aparência e organizam o conteúdo da página, usaremos as tags INPUT, que serão vistas a seguir.

O atributo mais importante de um form é sua ACTION (ação) que é a URL da página web para onde irão os dados preenchidos neste form, sendo que geralmente esta página é escrita em uma linguagem de programação como PHP e JAVA, ao invés de HTML. Nos exemplos que faremos neste livro, nossos formulários enviarão os dados para arquivos JavaScript que serão interpretados pelo Node.js.

Além deste atributo temos o METHOD, que define o verbo HTTP que será utilizado para transmissão dos dados (geralmente GET ou POST). HTTP é o protocolo de transferência de hipertexto utilizado na Internet mundial. Ele define os dados e metadados que devem ser transmitidos a cada requisição e resposta web.

Usamos HTTP GET quando queremos obter dados de uma página. Usamos HTTP POST quando queremos enviar dados para uma página. Em ambos os casos, a informação a ser enviada é incluída junto à requisição, em pares de chave-valor como verá a seguir.

Atenção: existem outros verbos HTTP como DELETE, PUT, HEAD e DEBUG, sendo que GET e POST são os únicos suportados por HTML Forms e também os mais utilizados no geral em qualquer contexto web.

Código 5.19: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <div>
      <h1>Tópico 1</h1>
      <p>Introdução ao HTML</p>
    </div>
    <form action="/teste"></form>
  </body>
</html>
```

Visualmente, o uso da tag FORM sozinha não renderiza nada no navegador do usuário, então passaremos às demais tags.

A tag LABEL

Uma tag que serve para criar um rótulo em seu FORM HTML geralmente acompanhada de outro componente. A LABEL é uma tag contâiner que permite texto ou um INPUT (veja a seguir) em seu interior. Geralmente a LABEL possui o mesmo comportamento visual de um texto escrito diretamente na página, e seu uso mais comum é como rótulo de campos de texto, sendo que segundo a especificação os pares LABEL + INPUT devem estar dentro de tags P (parágrafos) dentro do FORM.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <div>
      <h1>Tópico 1</h1>
      <p>Introdução ao HTML</p>
    </div>
    <form action="/teste">
      texto comum <br />
      <label>texto em label</label>
    </form>
  </body>
</html>
```

Tópico 1

Introdução ao HTML

texto comum
texto em label

Note no exemplo acima o uso da tag BR para quebrar a linha após o primeiro texto. Isso porque a tag LABEL tem comportamento inline (em-linha) ou seja, não quebra linha automaticamente. Essa seria outra alternativa em relação ao uso da tag P.

A tag INPUT

Na verdade INPUT não representa uma única coisa dentro de FORMs HTML, mas uma família de componentes utilizados para

criação de formulários. A tag INPUT pode ou não ser um contêiner, dependendo de seu tipo, definido no atributo TYPE.

Além do TYPE, as tags INPUT costumam ter o atributo VALUE, que representa o seu valor, e o atributo NAME, que representa o seu nome (invisível ao usuário final). Também temos o atributo DISABLED, que quando definido como TRUE torna o INPUT desabilitado, o que pode ser útil em algumas situações.

Os principais valores para o atributo TYPE e seus respectivos usos são:

TYPE="TEXT" e TYPE="PASSWORD"

Utilizados para criar campos de texto e de senha, respectivamente, semelhantes ao JTextField do Java Swing e TextBox do ASP.NET. O atributo NAME define o nome do seu campo (não confundir com seu rótulo) e o atributo VALUE define o seu conteúdo. Este INPUT TYPE não aceita outros elementos em seu interior e seu comportamento visual é inline (ou seja, não quebra linha automaticamente).

Código 5.21: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <div>
      <h1>Tópico 1</h1>
      <p>Introdução ao HTML</p>
    </div>
    <form action="/teste">
      <label>Usuário:
        <input type="text" name="txtUsuario" value="Luiz" /></label>
      <br />
```

```
<label>Senha:  
<input type="password" name="txtSenha" value="123" />  
</label>  
</form>  
</body>  
</html>
```

Nota: repare como coloquei o input dentro da tag label. Isso está perfeitamente correta e ajuda a dizer ao browser que "essa label é daquele input" e inclusive se você clicar no texto da label no navegador verá que o input fica selecionado pois o browser entende que deseja preenchê-lo.

Resultado do código acima (note a diferença entre o text e o password, além do fato de que o atributo NAME não aparece ao usuário):

Tópico 1

Introdução ao HTML

Usuário:

Senha:

TYPE="RADIO"

Em algumas ocasiões queremos que o usuário apenas uma dentre um número pequeno de opções. Assim, podemos definir estas opções como sendo botões de rádio HTML, os INPUT TYPE="RADIO", em analogia aos JRadioButtons e RadioButtons de outras linguagens.

Cada INPUT TYPE="RADIO" deve ser sucedido por um texto plano (pois ele não possui conteúdo próprio) e deve conter dois atributos: VALUE, que indica o valor da opção selecionada (não visível), e

NAME, que indica o nome deste grupo de botões de rádio. Ou seja, todos os botões de rádio com o mesmo name representam o mesmo grupo de informação, e com isso o usuário poderá selecionar apenas um deles (bem mais fácil que o JButtonGroup do Java Swing!).

Código 5.22: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <div>
      <h1>Tópico 1</h1>
      <p>Introdução ao HTML</p>
    </div>
    <form action="/teste">
      <label>Usuário:<br />
      <input type="text" name="txtusuario" value="Luiz" /></label>
      <label>Senha:<br />
      <input type="password" name="txtsenha" value="123" />
    </label>
      <br />
      <label>Sexo:</label>
      <input type="radio" name="sexo" value="M" checked
/>Masculino
      <input type="radio" name="sexo" value="F" />Feminino
    </form>
  </body>
</html>
```

Abaixo temos o resultado.

Tópico 1

Introdução ao HTML

Usuário:

Senha:

Sexo: Masculino Feminino

Note que o fato de ambos INPUT TYPE="RADIO" possuírem o mesmo valor de atributo NAME fará com que o usuário apenas possa selecionar um, bem como quando este formulário for submetido ao servidor, somente um deles será enviado sob a variável "sexo".

Note também que o atributo VALUE não é mostrado ao usuário, ele somente será conhecido quando o FORM for enviado ao servidor (e somente o VALUE do RADIO selecionado).

E por fim, note que foi utilizado um terceiro atributo, chamado CHECKED no RADIO masculino, mostrando que o mesmo deve vir selecionado por padrão. Este atributo não possui qualquer valor, a sua presença por si só no interior de uma tag INPUT RADIO já indica que o mesmo está selecionado.

INPUT TYPE="CHECKBOX"

Vimos antes que o INPUT RADIO serve para o usuário selecionar uma dentre algumas opções. Mas e quando ele pode selecionar mais de uma opção? Neste caso podemos usar o INPUT CHECKBOX.

Neste INPUT nós temos o atributo NAME para definir o nome da variável que será enviado ao servidor e fora do INPUT (ele não é um contâiner) temos o texto descritivo para o usuário entender do que se trata o controle, como mostra a última linha de INPUT do código abaixo.

Código 5.23: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <div>
      <h1>Tópico 1</h1>
      <p>Introdução ao HTML</p>
    </div>
    <form action="/teste">
      <label>Usuário:<input type="text" name="txtusuario" value="Luiz" /></label>
      <br />
      <label>Senha:<input type="password" name="txtsenha" value="123" /></label>
      <br />
      <label>Sexo:</label>
      <input type="radio" name="sexo" value="M" checked />Masculino
      <input type="radio" name="sexo" value="F" />Feminino
      <input type="checkbox" name="chkSpam" />Quero receber spam por e-mail
    </form>
  </body>
</html>
```

Como resultado temos:

Tópico 1

Introdução ao HTML

Usuário:

Senha:

Sexo: Masculino Feminino

Quero receber spam por e-mail

Note que poderíamos ter definido o atributo CHECKED neste INPUT CHECKBOX caso quiséssemos que ele viesse marcado por padrão, assim como fizemos com o INPUT RADIO anteriormente.

INPUT TYPE="SUBMIT" e TYPE="BUTTON"

Estes últimos INPUTs são tipos diferentes de botões. Ambos são representados visualmente da mesma forma, como botões com um texto dentro, definido pelo atributo VALUE desse input. Entretanto, suas funcionalidades variam: o INPUT BUTTON dispara um código Javascript quando clicado (que não veremos agora) e o INPUT SUBMIT submete todos os dados do formulário para o servidor, sendo este último o mais importante a ser estudado no momento.

Anteriormente, quando vimos a tag FORM foi falado que ela tinha dois atributos: ACTION e METHOD, sendo este último opcional. Quando um INPUT SUBMIT é clicado, o navegador coleta todas as informações contida nos campos do FORM e envia eles para o endereço fornecido no atributo ACTION, no formato NAME1=VALUE1&NAME2=VALUE2, ou seja, ele pega o name de cada componente HTML e seu valor e junta-os usando sinais de '=', e depois junta todos os pares de chave-valor usando sinais de '&'. Se o atributo METHOD não for definido, será enviado para o servidor via HTTP GET, caso contrário será enviado usando o verbo HTTP definido neste atributo (o que é irrelevante no momento).

Assim, quando esta informação chega em nosso backend, o nosso código programado em alguma linguagem de servidor irá dar cabo da mesma fazendo o que for necessário, como salvar dados no

banco, por exemplo, inclusive retornando uma mensagem de sucesso ou fracasso na maioria das vezes.

Código 5.24: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <div>
      <h1>Tópico 1</h1>
      <p>Introdução ao HTML</p>
    </div>
    <form action="/teste">
      <label>Usuário:<input type="text" name="txtusuario" value="Luiz" /></label>
      <br />
      <label>Senha:<input type="password" name="txtsenha" value="123" /></label>
      <br />
      <label>Sexo:</label>
      <input type="radio" name="sexo" value="M" checked />Masculino
      <input type="radio" name="sexo" value="F" />Feminino
      <input type="checkbox" name="chkSpam" />Quero receber spam por email
      <br />
      <input type="submit" value="Salvar" />
    </form>
  </body>
</html>
```

Como resultado, temos a imagem abaixo. Note que se clicarmos no INPUT SUBMIT seremos enviados para a página /teste que não existe, exibindo um erro no navegador. Isso é normal, uma vez que ainda não vimos como fazer a programação desta página em Node.js.

Tópico 1

Introdução ao HTML

Usuário:

Senha:

Sexo: Masculino Feminino

Quero receber spam por e-mail

Outros TYPES: não existem somente os TYPES mostrados acima. Existem muitos outros principalmente a partir da especificação 5 da linguagem HTML. Alguns TYPES especialmente interessantes vêm para substituir o INPUT TEXT quando queremos limitar o tipo de dados que pode ser informado pelo usuário, como o INPUT NUMBER (para somente números). Veja a lista completa em http://www.w3schools.com/tags/tag_input.asp

A tag TEXTAREA

Funciona de maneira idêntica ao INPUT TYPE="TEXT" porém permitindo múltiplas linhas, como em um JTextArea do Java Swing. O atributo VALUE do TEXTAREA define o seu conteúdo textual e seu NAME o nome da variável a ser enviada ao servidor.

As tags SELECT e OPTION

Quando queremos listar diversas opções ao usuário sem poluir demais o visual da tela podemos usar a tag SELECT, que é o equivalente ao JComboBox e DropDownList de outras linguagens. A tag SELECT é um contâiner que aceita tags OPTION em seu interior

e possui um atributo name para representá-la quando o FORM é submetido para o servidor.

Já as tags OPTION por sua vez possuem um conteúdo textual (são containers que aceitam somente texto plano em seu interior) e um conteúdo oculto, seu atributo VALUE, que pode ser utilizado para guardar o código correspondente ao texto escolhido pelo usuário. Quando o FORM é submetido ao servidor, é enviado uma variável com o NAME do SELECT e o VALUE do OPTION selecionado pelo usuário.

Código 5.25: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <div>
      <h1>Tópico 1</h1>
      <p>Introdução ao HTML</p>
    </div>
    <form action="/teste">
      <label>Usuário:<input type="text" name="txtusuario" value="Luiz" /></label>
      <br />
      <label>Senha:<input type="password" name="txtsenha" value="123" /></label>
      <br />
      <label>UF:</label>
      <select name="cmbUF">
        <option value="RS">Rio Grande do Sul</option>
        <option value="SC">Santa Catarina</option>
        <option value="PR">Paraná</option>
      </select>
```

```
</form>
</body>
</html>
```

Visualmente temos (onde o primeiro OPTION se torna o valor selecionado por padrão):

Tópico 1

Introdução ao HTML

Usuário:

Senha:

UF:

Outras tags

Estas não são as únicas tags existentes. Além disso, a descrição de cada tag vista aqui é um resumo bem superficial e o estudo mais aprofundado é vital para todo profissional que deseje trabalhar com programação web.

Mais informações podem ser obtidas no site oficial da W3.org.

Node.js + HTML

Como vimos no capítulo anterior, usaremos o framework Express para criar aplicações web usando Node.js. O Express é um framework bem extensível o que lhe permite plugar diversos módulos nele para alterar seus comportamentos. Dentre esses comportamentos está a renderização de páginas web, que aqui vimos que é feita usando a linguagem HTML.

Durante todo este livro usaremos o módulo de renderização (view engine) Embedded JavaScript ou EJS, que permite usar JavaScript server-side juntamente com HTML para construir nossas telas. Essa junção de linguagens é permitida através do uso de server-tags como no exemplo abaixo, dentro de arquivos com extensão .ejs que são processados usando o Node.js:

```
<%= title %>
```

Como assim?

Vimos neste capítulo que arquivos HTML são interpretados diretamente no navegador, sem necessidade de qualquer software adicional. No entanto, arquivos .ejs são processados pelo módulo EJS em conjunto com o Node.js. Esse módulo lê o arquivo .ejs, da primeira à última linha, e quando encontra uma server-tag (aqueles tags com %), delega ao Node.js que faça alguma coisa com ela, pois é código JS server-side.

Nota: se o EJS encontrar código JavaScript dentro de uma tag <SCRIPT> (que veremos mais pra frente), ele ignora, pois nesses casos é JavaScript client-side e deve ser processado pelo browser.

O que o Node.js vai fazer com essa server-tag depende de diversos fatores de como a server-tag está escrita e qual seu conteúdo.

A server-tag pode ser descrita como abaixo:

```
<%= title %>
```

quando queremos apenas "imprimir" naquele local do HTML, o resultado de uma expressão JavaScript, que no exemplo acima, é apenas uma variável simples que deve ter vindo no model da requisição (lá na sua rota, lembra?).

Atenção: se você tentar imprimir ou usar uma variável que não existe no model dessa requisição, dará erro no Node.js. Esse erro estará impresso no terminal.

Essa impressão é semelhante ao 'echo' do PHP ou ao 'Response.Write' do ASP.NET e permite inclusive que você escreva HTML, CSS ou JS client-side com ela. Ou seja, se a variável 'title' contiver texto HTML dentro dela, isso irá parar no HTML final que será renderizado no navegador, porque essa server-tag é processada no back-end, antes da resposta ser enviada pro usuário.

Mas não é apenas para embutir texto no HTML que servem as server-tags.

Se você usar apenas <%, você pode colocar qualquer código JavaScript que você quiser, incluindo condicionais, laços, declaração de variáveis e functions, etc. Tudo que for declarado é válido para todo o contexto da página e permite, por exemplo, declarar uma variável em um ponto e usá-la em outro ponto:

Código 5.26: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <% var teste = 0; %>
```

```
<p>Apenas um texto</p>
<%= teste %>
</body>
</html>
```

No exemplo acima, eu declarei a variável teste no início da página e apenas no final dela quis imprimir o seu valor. Se você salvar esse código dentro de um arquivo .ejs, na sua pasta views, e chamá-lo através de alguma rota em uma aplicação Express você verá o resultado final no seu navegador, que exibirá o texto e o número 0.

Se você inspecionar o código-fonte da página (Ctrl+U no Chrome ou clique-direito do mouse > Inspecionar), não verá sinal das server-tags, pois elas são processadas no servidor antes de serem enviadas ao browser.

Mais do que apenas lidar com variáveis, as server-tags permitem condicionais que podem mudar completamente como seu HTML será renderizado, como abaixo, onde decido se vou exibir uma div ou outra baseado em uma variável que veio no model da requisição:

Código 5.27: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Programação para Internet</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <% if(exibirDiv) { %>
      <div>Apenas uma div</div>
    <% } else { %>
      <div>Apenas outra div</div>
    <% } %>
```

```
</body>
</html>
```

Note como quebrei a estrutura if/else usando server-tags enquanto que colocava HTML como diretivas do if/else. Se a variável exibirDiv vier do model como true, a primeira div será renderizada para o usuário, senão, a segunda será renderizada.

Para testar esse código, salve dentro do arquivo views/index.ejs (ou crie uma nova rota completa, assim como fizemos no exercício final do capítulo anterior) e modifique o seu routes/index.js para passar esse model como abaixo:

Código 5.28: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
router.get('/', function(req, res, next){
  res.render('index', { exibirDiv: true });//ou false, você escolhe
})
```

O mesmo vale para um laço de repetição, que podemos usar para criar repetições dinâmicas de HTML em nossas páginas.

Exercitando

Agora que aprendemos como criar e rotear páginas HTML usando Node.js com o framework Express, o que acha de criarmos um projeto que nos permita compreender como fazer uma aplicação CRUD (create, retrieve, update e delete) em Node.js?

Obviamente não vimos persistência de dados ainda, o que veremos no capítulo seguinte, mas o que acha de deixarmos as páginas semi-prontas e com o fluxo de roteamento ok?

Crie o seu projeto Express usando o express-generator, assim como fizemos anteriormente (chamei aqui o projeto de crud):

Código 5.29: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
express -e --git crud
```

Depois entre na pasta do projeto e instale as dependências:

Código 5.30: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
cd crud  
npm install
```

Se você tem o Visual Studio Code instalado, abra-o e vá no menu File > Open e selecione a pasta do projeto que acabamos de criar, para ser mais fácil programar ele.

A nossa index será a página de listagem. Ela precisará ter uma tabela de clientes com nome, idade e UF, mas sem nenhum cliente por enquanto, incluindo nessa tabela links para editar e excluir cada uma das linhas. Ela também deve ter um link para ir à tela de cadastro, no final da tela.

A imagem abaixo mostra como a sua views/index.ejs deve se parecer neste momento e sinta-se livre para codificar o HTML conforme o seu gosto:



Listagem de Clientes

Clientes já cadastrados no sistema.

Nome	Idade	UF	Ações
Nenhum cliente cadastrado.			
Cadastrar Novo			

Caso esteja problema com o HTML, use o código abaixo como exemplo (note que usei um atributo STYLE e uma tag LINK referenciando um arquivo css que ainda não estudamos):

Código 5.31: disponível em <https://www.luiztools.com.br/livro-node-fontes>

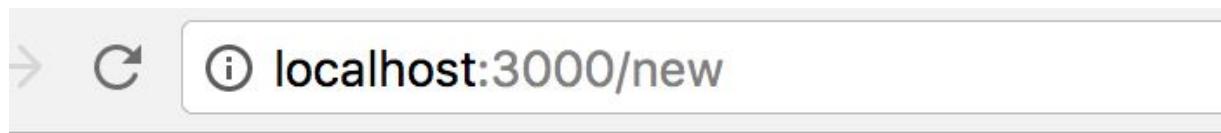
```
<!DOCTYPE html>
<html>
  <head>
    <title>CRUD de Clientes</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Listagem de Clientes</h1>
    <p>Clientes já cadastrados no sistema.</p>
    <table style="width:50%">
      <thead>
        <tr style="background-color: #CCC">
          <td style="width:50%">Nome</td>
          <td style="width:15%">Idade</td>
          <td style="width:15%">UF</td>
          <td>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>
```

```
</tfoot>
</table>
</body>
</html>
```

Essa tela será modificada depois para dinamicamente listar os clientes cadastrados no banco de dados...quando descobrirmos como fazer isso!

Agora, note que o link de Cadastrar Novo está levando para uma URL /new que ainda não existe.

Primeiro, crie uma nova view chamada new.ejs com a seguinte aparência:



Cadastro de Cliente

Preencha os dados abaixo para salvar o cliente.

Nome:

Idade:

UF: RS

[Cancelar](#) |

Se estiver com problemas no HTML, pode usar o código abaixo.

Código 5.32: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>CRUD de Clientes</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <meta charset="utf-8" />
  </head>
  <body>
```

```

<h1><%= title %></h1>
<p>Preencha os dados abaixo para salvar o cliente.</p>
<form action="<%= action %>" 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>

```

Agora, para que essa tela new.ejs possa ser acessada quando o usuário clicar no link da homepage, devemos criar uma rota GET /new no nosso routes/index.js, como abaixo:

Código 5.33: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```

/* GET new page. */
router.get('/new', function(req, res, next) {
  res.render('new', { title: "Cadastro de Cliente", action: "/new" })
})

```

Você deve ter notado que o action do form HTML é dinâmica, ela vem do model (isso fará toda diferença mais tarde). Por ora o model está retornando a própria página /new, mas o method do FORM é POST, ou seja, precisamos criar outra rota para receber um POST /new. Futuramente esta rota vai salvar os dados no banco, mas por enquanto apenas está redirecionando o usuário de volta para a página inicial com uma informação na URL:

Código 5.34: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
/* POST new page. */
router.post('/new', function(req, res, next) {
  //futuramente vamos salvar o cliente aqui
  res.redirect('?new=true')
})
```

Agora você pode executar seu projeto usando o comando abaixo:

Código 5.35: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
npm start
```

E poderá testar o fluxo dessa aplicação, porque funcionalidade ela não possui ainda.

Deixe este projeto aberto, continuaremos ele no final do próximo capítulo, para fazê-lo funcionar!

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Tutorial de HTML5

Excelente tutorial de HTML5 disponível gratuitamente na W3Schools..

<https://www.w3schools.com/html/>

Upload de arquivos

Vimos formulários com inputs comuns, mas e se quisermos usar inputs de upload de arquivo?

<https://www.luiztools.com.br/post/como-fazer-upload-de-arquivos-em-node-js/>

Validação de formulários

Aprenda como validar os inputs de formulários usando o pacote Joi.

<https://www.luiztools.com.br/post/tutorial-de-validacao-de-input-de-dados-em-node-js/>

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

6 Back-end: MongoDB

Truth can only be found in one place: the code.
— Robert C. Martin

Em 2007, mais ou menos, eu estava fazendo as cadeiras de Banco de Dados I e Banco de Dados II na faculdade de Ciência da Computação. Eu via como modelar um banco de dados relacional, como criar consultas e executar comandos SQL, além de álgebra relacional e um pouco de administração de banco de dados Oracle.

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

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

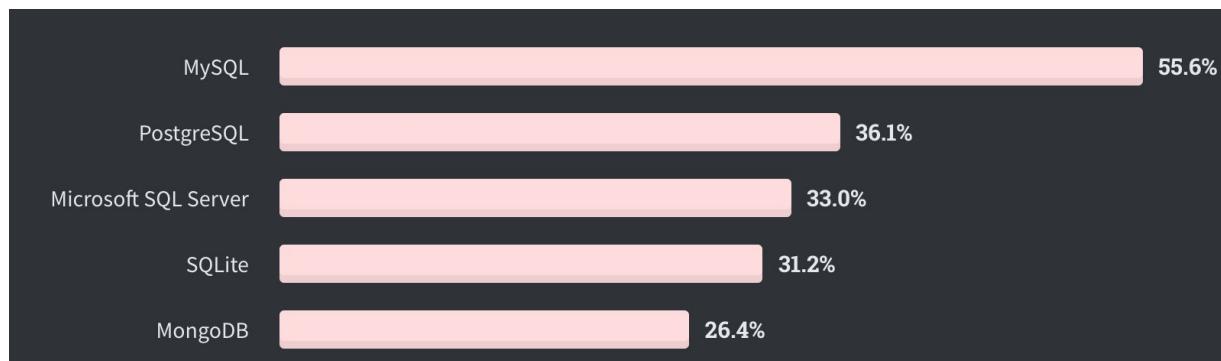
Introdução ao MongoDB

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 1/4 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 (que nem aparece ali na imagem).

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",           ← field: value  
    age: 26,              ← field: value  
    status: "A",           ← field: value  
    groups: [ "news", "sports" ] ← field: value  
}
```

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

Além do formato de documentos utilizado pelo MongoDB ser perfeitamente intercambiável com o JSON serializado do JS,

MongoDB opera basicamente de maneira assíncrona em suas operações, assim como o próprio Node.js, o que nos permite ter uma persistência extremamente veloz aliado a uma plataforma de programação igualmente rápida.

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

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

Quando não devo usar MongoDB?

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

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

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

Além disso, o MongoDB possui alguns concorrentes que possuem variações interessantes que de repente se encaixam melhor como solução ao seu problema. Outro mecanismo bem interessante desta categoria de banco de dados é o RethinkDB, que foca em consultas-push real-time de dados do banco, ao invés de polling como geralmente se faz para atualizar a tela.

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

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

Instalação e Testes

Diversos players de cloud computing fornecem versões de Mongo hospedadas e prontas para uso como [Umbler](#) e [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 (esta pasta já deve estar criada).

Código 6.1: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
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 6.2: disponível em <https://www.luiztools.com.br/livro-node-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 6.3: disponível em <https://www.luiztools.com.br/livro-node-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 6.4: disponível em <https://www.luiztools.com.br/livro-node-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 6.5: disponível em <https://www.luiztools.com.br/livro-node-fontes>

show collections

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

Código 6.6: disponível em <https://www.luiztools.com.br/livro-node-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 6.7: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

Como sabemos se funcionou?

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

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!

Comandos elementares

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

Array Insert

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

Código 6.8: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
custArray = [{ nome : "Monica", idade : 31 }, { nome : "Teste", "uf" :  
"RS" }]  
db.customers.insert(custArray)
```

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

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

Nesse exemplo passei um array com vários documentos para nossa função `insert` inserir na coleção `customers` e isso nos trás algumas coisas interessantes para serem debatidas. Primeiro, sim, você pode passar um array de documentos por parâmetro para o `insert`.

Segundo, você notou que o segundo documento não possui "idade"? E que ele possui um campo "uf"?

Find avançado

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

Código 6.9: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

Mas voltando à questão do "uf", ao contrário dos bancos relacionais, o MongoDB possui schema variável, ou seja, se somente um customer tiver "uf", somente ele terá esse campo, não existe um schema pré-definido compartilhado entre todos os documentos, cada um é independente. Obviamente considerando que eles compartilham a mesma coleção, é interessante que eles possuam coisas em comum, caso contrário não faz sentido guardar eles em uma mesma coleção.

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

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

Código 6.10: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

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

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

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

Código 6.11: disponível em <https://www.luiztools.com.br/livro-node-fontes>

db.customers.find({nome: { \$regex: /a/ }})

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

Uma forma abreviada de fazer a mesma consulta acima seria como abaixo. O MongoDB interpreta textos entre // (barras) como sendo expressões regulares, automaticamente:

Código 6.12: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
db.customers.find({nome: /a/})
```

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

Código 6.13: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

Código 6.14: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

O que a expressão acima irá retornar?

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

E a expressão abaixo:

Código 6.15: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

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

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

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

Código 6.16: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

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

Código 6.17: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

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

Update

Além do `insert` que vimos antes, também podemos atualizar documentos já existentes, por exemplo usando o comando `update` e derivados. O jeito mais simples de atualizar um documento é chamando a função `update` na coleção. Esta função possui dois parâmetros obrigatórios e um opcional:

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

Como em:

Código 6.18: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
db.customers.update({nome: "Luiz"}, {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 quantos documentos foram atualizados.

Se você deseja que o update afete mais de um documento que atenda ao filtro, deve passar o terceiro parâmetro, como abaixo:

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

E aqui vão alguns cuidados que você deve ter...

Primeiro, 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!

E segundo, que se você passar o documento errado e ainda usar a opção `{multi: true}`, muita coisa pode ser substituída erroneamente, certo?

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

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

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

Mas que operadores são esses?

Update Operators

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

Código 6.19: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

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

Nota: você pode usar null se quiser "limpar" um campo.

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

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

Não importa o valor que ela tenha antes, o operador \$set vai sobrescrevê-lo. Agora, se o valor anterior importa, como quando queremos incrementar o valor de um campo, não se usa o operador

`$set`, mas sim outros operadores. A lista dos mais úteis operadores de update estão abaixo:

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

Atenção: os operadores `$rename`, `$unset` e potencialmente o operador `$set`, podem ser usados para alterar a estrutura original de um documento (renomeando, removendo ou adicionando campos, respectivamente). No entanto, como não há um schema compartilhando entre todos os documentos de uma coleção, eles alteram apenas o(s) documento(s) englobados no filtro de update utilizado.

Update Options

Existe um terceiro parâmetro opcional no `update` e `updateOne` que são as opções de update. Dentre elas, existe uma muito interessante do MongoDB: **`upsert`**, como abaixo:

Código 25 ou 3.18: disponível em <http://www.luiztools.com.br/livro-mongo-fontes>

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

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

`upsert = "update ou insert"`

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

Delete

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

Existe uma função `deleteOne` e uma `deleteMany`, o que a essa altura do campeonato você já deve imaginar a diferença. Também existe uma função `remove`, mas não aconselho utilizá-la.

Além disso, assim como o `find` e o `update`, o primeiro parâmetro dos deletes é o filtro que vai definir quais documentos serão deletados e todos os filtros normais do `find` são aplicáveis.

Sendo assim, de maneira bem direta:

Código 6.21: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
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.

Node.js + MongoDB

Agora que entendemos, em capítulos anteriores, 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 uma continuação do projeto chamado 'crud' que começamos no capítulo sobre HTML. Basicamente temos duas telas: uma de listagem e outra de cadastro e embora isso pareça pouco, você verá que com um pouco de criatividade isso será o bastante para fazer a listagem, cadastro, edição e exclusão de clientes a partir do nosso banco MongoDB.

Certifique-se de que o seu banco MongoDB está rodando e que ele contém alguns customers/clientes de exemplo, conforme os exercícios que fizemos anteriormente. Abra também 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 (que chamamos de crud no último tutorial, lembra?), rode o seguinte comando que além de baixar a dependência também salva a mesma no seu `packages.json`:

Código 6.22: disponível em <https://www.luiztools.com.br/livro-node-fontes>

`npm install mongodb`

Para deixar nossa aplicação minimamente organizada não vamos sair por aí escrevendo lógica de acesso à dados. Vamos centralizar tudo o que for responsabilidade do MongoDB dentro de um novo módulo chamado db.js, que nada mais é do que um arquivo db.js na raiz do nosso projeto.

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

Código 6.23: disponível em <https://www.luiztools.com.br/livro-node-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;
}

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

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.

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

Mas e se a conexão não for estabelecida com sucesso?

Nesse caso disparamos um erro que deverá ser tratado por quem chamar esta função.

A última linha do nosso db.js contém o module.exports que é a instrução que permite compartilhar objetos com o restante da aplicação.

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):

Código 6.26: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
async function findAll(){
  const db = await connect();
  return db.collection("customers").find().toArray();
}
```

Nesta função 'findAll', nós realizamos a conexão com o banco de dados, usamos ela para nos conectar na coleção de customers para fazer um find sem filtro. O resultado desse find é um cursor, então usamos o toArray para convertê-lo para um array e quando terminar, retornaremos ele. Note o uso do await para que o retorno só seja realizado quando a conexão terminar, o que obriga também o uso da palavra reservada async antes da function.

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:

Código 6.27: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
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:

Código 6.28: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const db = require('../db');
/* GET home page. */
router.get('/', async function(req, res) {
  res.render('index', {docs: await db.findAll()});
})
```

A primeira linha carrega o nosso módulo db.js, que usaremos para acessar o banco de dados nos trechos de código a seguir.

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 './'.

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. O retorno é usado como model da função de renderização da página.

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:

Código 6.29: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

E agora, logo abaixo do bloco anterior, adicione um else para que execute um laço sobre o array de documentos do model e, dessa maneira, construa a nossa tabela:

Código 6.30: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<% } 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><!-- 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:

Código 6.31: disponível em <https://www.luiztools.com.br/livro-node-fontes>

npm start

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



Listagem de Clientes

Clientes já cadastrados no sistema.

Nome	Idade	UF	Ações
Luiz	29	RS	
Fernando	29		
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 sobre HTML nós deixamos uma rota GET /new que renderiza a view new.ejs e uma rota POST / criada para receber os POSTs vindos do HTML FORM da página 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:

Código 6.32: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
async function insert(customer){  
  const db = await connect();  
  return db.collection("customers").insertOne(customer);  
}
```

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

Código 6.33: disponível em <https://www.luiztools.com.br/livro-node-fontes>

`module.exports = { findAll, insert }`

Agora abra o seu routes/index.js e vamos editar a rota POST /new. Nós chamaremos o objeto db para salvar os dados no Mongo adicionando o seguinte bloco de código:

Código 6.34: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
/* POST new page. */
router.post('/new', async function(req, res) {
  const nome = req.body.nome
  const idade = parseInt(req.body.idade)
  const uf = req.body.uf
  await db.insert({nome, idade, uf});
  res.redirect('?new=true')
})
```

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 sequência, apenas coloquei um redirecionamento da resposta para outra página da aplicação. Como usei um await na linha anterior, esse redirecionamento só vai acontecer depois que o insert for finalizado.

Atenção: note que usei `parseInt` para converter o `req.body.idade`. Isso porque todos os dados vêm 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 de quatro etapas de uma aplicação CRUD completa.

Atualizando clientes

Para atualizar clientes (o U do CRUD) não é preciso muito esforço diferente do que estamos fazendo até agora. No entanto, são várias coisas menores que precisam ser feitas e é bom tomar cuidado para não deixar nada pra trás.

Primeiro, vamos editar nossa `views/index.ejs` para que quando o usuário clicar em um link de edição na coluna Ações, seja redirecionado para a tela de cadastro, mas com o formulário preenchido. Quando estiver na tela de cadastro em 'modo edição', se ele clicar no botão de salvar o cliente será atualizado ao invés de cadastrado.

Fazemos isso com um pequeno ajuste no `index.ejs`, dentro do `forEach`, na coluna Ações (onde antes estava o comentário 'em breve'):

Código 6.35: disponível em <https://www.luiztools.com.br/livro-node-fontes>

`<td><a href="/edit/<%= customer._id %>">Editar</td>`

Note que este link aponta para uma rota /edit que ainda não possuímos, e que após a rota ele adiciona o _id do customer no caminho da URL, que servirá para identificá-lo na página seguinte. Com esse _id em mãos, teremos de fazer uma consulta no banco para carregar seus dados no formulário permitindo um posterior update. Por ora, apenas mudou a aparência da tela de listagem:



Listagem de Clientes

Clientes já cadastrados no sistema.

Nome	Idade	UF	Ações
Luiz	29	RS	Editar
Fernando	29		Editar
Teste	28	RS	Editar
LuizTools	28	RS	Editar
Cadastrar Novo			

Sendo assim, vamos começar criando uma nova função no db.js que retorna apenas um cliente, baseado em seu _id:

Código 6.36: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const {MongoClient, ObjectId} = require("mongodb");
//...
async function findOne(id){
  const db = await connect();
  const objId = new ObjectId(id);
  return db.collection("customers").findOne(objId);
}
```

Logo no topo do arquivo db.js, edite a linha que faz o require na biblioteca do mongodb para que tenhamos também um objeto ObjectId. Como nosso filtro do find será o id, ele deve ser convertido para ObjectId, pois virá como string na URL e o Mongo não entende Strings como _id. Não esqueça de incluir esta nova função no module.exports, que está crescendo:

Código 6.37: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

Agora vamos criar a respectiva rota GET em nosso routes/index.js que carregará os dados do cliente para edição no mesmo formulário de cadastro. Coloque antes das demais rotas, para evitar conflito com o outro GET:

Código 6.38: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
/* GET edit page. */
router.get('/edit/:id', async function(req, res, next) {
  const id = req.params.id
  const doc = await db.findOne(id);
  res.render('new', { title: 'Edição de Cliente', doc, action: '/edit/' +
    doc._id })
})
```

Esta rota está um pouco mais complexa que o normal. Isso porque a notação :id diz que a informação após /edit deve ser considerada um parâmetro de URL e esse parâmetro deve se chamar id.

Assim, nós pedimos ao db que encontre o cliente cujo id veio como parâmetro da requisição (req.params.id). Após ele encontrar o dito cujo, mandamos renderizar a mesma view de cadastro, porém com

um model inteiramente novo contendo apenas um documento (o cliente a ser editado) e a action do form da view 'new.ejs'.

Nota: no capítulo sobre HTML nós já havíamos deixado o action do HTML FORM de cadastro como dinâmico lembra? Ele pega o model action que vier para que tenhamos uma diferenciação entre o POST de cadastro e o POST de edição.

Antes de editar nossa view para que receba as informações do cliente a ser editado, vamos alterar a rota GET /new para que retorne um cliente vazio, evitando erros depois (caso você tente chamar um model que não existe na view, dá erro):

Código 6.39: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
/* GET new page. */
router.get('/new', function(req, res, next) {
  res.render('new', { title: 'Cadastro de Cliente', doc: {} })
})
```

Agora vamos na views/new.ejs e vamos editá-la para preencher os campos do formulário com o model recebido, bem como configurar o form com o mesmo model:

Código 6.40: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<form action="<%=\naction %>" method="POST">
  <p>
    <label>Nome: <input type="text" name="nome" value="<%=\n      doc.nome %>" /></label>
  </p>
  <p>
    <label>Idade: <input type="number" name="idade" value="<%=\n      doc.idade %>" /></label>
  </p>
```

```

<p>
    <label>UF: <select name="uf">
        <% const s = "selected" %>
        <option <% if(doc.uf === "RS") { %><%= s %><% } %>>RS</option>
        <option <% if(doc.uf === "SC") { %><%= s %><% } %>>SC</option>
        <option <% if(doc.uf === "PR") { %><%= s %><% } %>>PR</option>
        <!-- coloque os estados que quiser --&gt;
    &lt;/select&gt;&lt;/label&gt;
&lt;/p&gt;
&lt;p&gt;
    &lt;a href="/"&gt;Cancelar&lt;/a&gt; | &lt;input type="submit"
value="Salvar" /&gt;
&lt;/p&gt;
&lt;/form&gt;
</pre>


---



```

Note que bastou associar o model correspondente a cada campo no value de cada input e quase tudo ficou pronto. A única coisa mais complicada aqui foi fazer a seleção correta da UF. Isso porque o SELECT não possui um atributo VALUE, mas o OPTION que deva estar selecionado deve ter o atributo SELECTED. Assim, tive de testar um-a-um (com IFs) para ver qual option devia ser selecionada.

Nota: existem maneiras mais elegantes de resolver este problema usando JavaScript client-side. Como ainda não estudamos isso, deixaremos assim por enquanto.

Agora, se você mandar rodar e clicar em um link de edição, deve ir para a tela de cadastro mas com os campos preenchidos com os dados daquele cliente em questão:

Edição de Cliente

Preencha os dados abaixo para salvar o cliente.

Nome: Luiz

Idade: 29

UF: RS

[Cancelar](#) | [Salvar](#)

Agora estamos muito perto de terminar a edição. Primeiro precisamos criar uma nova função no db.js para fazer update, como abaixo:

Código 6.41: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
async function update(id, customer){  
  const filter = { _id: new ObjectId(id) };  
  const db = await connect();  
  return db.collection("customers").update(filter, customer);  
  
}  
  
module.exports = { findAll, insert, findOne, update }
```

O processo não é muito diferente do insert, apenas temos de passar o filtro do update para saber qual documento será afetado (neste caso somente aquele que possui o id específico). Também já inclui o module.exports atualizado na última linha para que você não esqueça.

Para encerrar, apenas precisamos configurar uma rota para receber o POST em /edit com o id do cliente que está sendo editado, chamando a função que acabamos de criar:

Código 6.42: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
/* POST edit page. */
router.post('/edit/:id', async function(req, res) {
  const id = req.params.id
  const nome = req.body.nome
  const idade = parseInt(req.body.idade)
  const uf = req.body.uf
  await db.update(id, {nome, idade, uf});
  res.redirect('/?edit=true');
})
```

Aqui carreguei o id que veio como parâmetro na URL, e os dados de nome, idade e uf no body da requisição (pois foram postados via formulário). Apenas passei esses dados nas posições corretas da função de update e logo depois fiz um redirect de volta à tela de listagem (com uma informação na URL que usaremos no futuro).

E com isso finalizamos o update!

Excluindo um cliente

Agora em nossa última parte do tutorial, faremos o D do CRUD, D de Delete!

Vamos voltar à index.ejs (tela de listagem) e adicionar um link específico para exclusão, na coluna Ações, assim como fizemos com o link de editar anteriormente, como abaixo:

Código 6.43: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<td>
  <a href="/edit/<%= customer._id %>">Editar</a>
  <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 pouco de JavaScript client-side que veremos mais pra frente, 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.

The screenshot shows a web browser window with the URL 'localhost:3000' in the address bar. A modal dialog box is open, displaying the message 'localhost:3000 diz:' followed by 'Tem certeza?' (Are you sure?). There are two buttons at the bottom right of the dialog: 'Cancelar' (Cancel) and 'OK'. Below the browser window, there is a table titled 'Listagem de Clientes' with the following data:

Nome	Idade	UF	Ações
Luiz	29	RS	Editar Excluir
Fernando	29		Editar Excluir
Teste	28	RS	Editar Excluir
LuizTools	28	RS	Editar Excluir

At the bottom left of the table, there is a link '[Cadastrar Novo](#)'.

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

Código 6.44: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
async function deleteOne(id){  
    const db = await connect();  
    const filter = { _id: new ObjectId(id) };  
    return db.collection("customers").deleteOne(filter);  
}  
  
module.exports = { findAll, insert, findOne, update, 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 (vamos excluir através de um link, então tem de ser GET):

Código 6.45: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
/* GET delete page. */  
router.get('/delete/:id', async function(req, res) {  
    const id = req.params.id  
    await db.deleteOne(id);  
    res.redirect('/?delete=true');  
})
```

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

E com isso finalizamos nosso CRUD!

Mensagens ao Usuário

Quando a gente fala de construção de aplicações para usuários existe uma disciplina chamada usabilidade que estuda as melhores práticas para construção de interfaces humano-computador. Uma dessas boas práticas é informar ao usuário o que está acontecendo na aplicação e o jeito mais fácil de fazer isso é através de mensagens.

Em todas as nossas rotas, estamos sempre considerando apenas o caminho de sucesso, certo? Mas e se der erro?

Em JavaScript, fazemos o tratamento de erro usando um bloco try/catch. Tudo que estiver dentro do try está "monitorado" em relação a erros. Se algum erro acontecer, o fluxo será desviado para o conteúdo do catch.

Vamos alterar o código de nossas rotas para tratar o erro e enviar essa informação para a tela inicial da aplicação. Aplique a alteração abaixo em todas rotas da aplicação que executam alguma função do db.js:

Código 6.46: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
/* GET home page. */
router.get('/', async function(req, res) {
  try{
    res.render('index', {docs: await db.findAll()});
  }catch(ex){
    res.redirect(`/erro=${ex}`);
  }
})
```

Acredito que tenha usado Template Literals algumas vezes e não tenha lhe explicado o que é. Quando queremos juntar strings literais com variáveis no código, podemos fazê-lo concatenando do jeito tradicional (com o sinal de +) ou com Template Literals, que são as

strings entre crases (`string`). Para adicionar uma variável dentro dessa string, basta usar \${nomeVariavel}.

Note que nesta rota e nas demais que salvam ou excluem dados no banco, eu passei parâmetros na URL. O que faremos com esses parâmetros?

Com um pouco de JavaScript client-side (o JavaScript do Node.js é server-side, lembra?), podemos exibir mensagens personalizadas para o usuário ter a certeza de que a operação que ele realizou foi bem sucedida.

Não quero entrar em detalhes antes da hora, mas adicione o seguinte bloco SCRIPT na sua index.ejs, logo antes do </body>, para verificar a existência desses parâmetros e tomarmos ações:

Código 6.47: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<script>
  if(location.href.indexOf('delete=true') != -1){
    alert('Cliente excluído com sucesso!');
  }
  else if(location.href.indexOf('edit=true') != -1){
    alert('Cliente editado com sucesso!');
  }
  else if(location.href.indexOf('new=true') != -1){
    alert('Cliente cadastrado com sucesso!');
  }
  else if(location.href.indexOf('erro') != -1){
    alert('Ocorreu um erro!');
  }
</script>
```

O resultado você confere abaixo, com uma exclusão de exemplo.

localhost:3000/?delete=true

localhost:3000 diz:

Cliente excluído com sucesso!

OK

Listagem de Clientes

Clientes já cadastrados no sistema.

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

[Cadastrar Novo](#)

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

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Palestra Node.js + MongoDB

Palestra que realizei no IX Telecompact, na Unilasalle em Canoas/RS.

<https://www.youtube.com/watch?v=4kYww7GLg0>

ExpressJS

Vídeo no meu canal onde introduzo o ExpressJS na prática com Node.js.

<https://www.youtube.com/watch?v=QQFKneLazng>

MongoDB para Iniciantes

Meu livro focado em MongoDB, com muito material para enriquecer o seu conhecimento neste banco de dados.

<https://www.luiztools.com.br/livro-mongodb>

Persistência Poliglota

Entenda o movimento NoSQL e os principais players nesta palestra que dei durante a TechParty da Faccat, em Taquara/RS.

<https://www.youtube.com/watch?v=MznBxzSgV6w>

Mongoose

Aprenda a usar o Mongoose, principal ODM para MongoDB (o equivalente não-relacional aos ORMs)

<https://www.luiztools.com.br/post/tutorial-nodejs-com-mongodb/>

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

7 Back-end: Web APIs

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

— Max Kanat-Alexander

Uma Web API é uma interface de programação de aplicações (API) tanto para um servidor quanto um navegador. É utilizada para se conseguir recuperar somente o valor necessário 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.

Introdução ao HTTP

HTTP ou Hyper-Text Transfer Protocol (Protocolo de Transferência de Hipertexto) é o protocolo de comunicação padrão da Internet. É seguindo as regras deste protocolo que os clientes (como os browsers) e servidores conseguem conversar e se entender.

O seu entendimento é vital para a construção de web APIs.

REQUEST

As requisições HTTP são os comandos enviados por um cliente a um servidor. Requisições HTTP contém um cabeçalho (header) e um corpo (body).

Dentro do header temos diversas meta-informações, dentre elas um verbo, que veremos mais à frente do que se trata e uma URL (endpoint) no servidor que estamos fazendo a requisição (com ou sem parâmetros). Já no corpo, temos opcionalmente dados que queremos enviar ao servidor, em formatos pré-acordados para facilitar o entendimento.

RESPONSE

Em resposta à cada requisição o servidor envia uma resposta. Essa resposta também possui cabeçalho (header) e corpo (body).

No cabeçalho, dentre outras informações, existe um código de resposta, como os descritos mais adiante, enquanto que no corpo podem vir dados do servidor. O mais comum é quando as requisições enviam dados no corpo, a resposta não retorna nada no seu corpo e vice-versa. Daqui a pouco você vai entender o porquê.

Códigos de retorno comuns incluem:

- **200:** Ok, requisição processada com sucesso
- **403:** Forbidden, não foi possível processar a requisição pois o cliente não possui permissão para efetuar aquela ação.

- **404:** Not found, não foi possível processar a requisição pois não foi encontrado o objeto que se estava procurando.
- **500:** Internal server error, não foi possível processar a requisição devido a uma falha no servidor.

Verbos HTTP e o protocolo REST

Todas as requisições HTTP são organizadas em verbos, que indicam o que a requisição deseja fazer no servidor, enquanto que o endpoint indica onde ela deseja fazer. São os comportamentos do protocolo. O efeito final de cada verbo depende muito do servidor e o que ele faz, no entanto, existe um comportamento básico considerado padrão para cada um deles, sendo os principais listados abaixo.

A cada requisição HTTP usamos apenas um desses verbos, sendo necessários mais de uma requisição caso queiramos executar mais de um. Cada requisição contém no mínimo um verbo e uma URL (endpoint) na qual esse verbo será executado. Opcionalmente pode ter um corpo, com dados a serem enviados junto da requisição.

O uso correto dos verbos HTTP é imprescindível para a criação de web APIs usando o protocolo REST, o mais comum nos dias de hoje e o mais veloz também. O Representational State Transfer ou Transferência de Estado Representacional é uma abstração da arquitetura da World Wide Web, um estilo arquitetural que consiste de um conjunto coordenado de restrições arquiteturais aplicadas a componentes, conectores e elementos de dados dentro de um sistema de hipermídia distribuído.

Resumidamente o REST é um protocolo de comunicação sem estado, ou seja, cada requisição é atômica e independente. O servidor não combina diferentes requisições. Além disso, o verbo HTTP utilizado diz exatamente o que o servidor deve fazer, por isso vamos estudá-los agora.

GET

Verbo HTTP para obter conteúdo do servidor. Como o próprio nome sugere, é análogo aos métodos getters de diversas linguagens de programação como Java e C#, como `getNome` por exemplo. URLs de consulta e de listagem de uma web API, por exemplo, devem ser acessadas somente via verbo GET.

Uma requisição GET em uma URL geralmente traz todos os elementos que correspondem àquela URL, mas caso o usuário passe mais informações, elas funcionarão como filtro, seja um ID para trazer apenas um objeto ou uma consulta por nome, por exemplo

Exemplos comuns de requisições GET a web APIs podem ser como as abaixo:

```
GET /clientes  
GET /clientes/1  
GET /clientes?id=1  
GET /clientes?busca=luiz
```

Além disso, quando acessamos URLs da Internet no navegador do seu computador, o browser sempre faz uma requisição GET na URL que digitamos. A menos que aconteça um erro durante a consulta, um GET retorna dados no corpo da resposta (HTML no caso do browser, mas pode ser um objeto JSON ou qualquer outra informação) e um "200 OK" no cabeçalho.

POST

Verbo HTTP para enviar um conteúdo novo ao servidor ou criar um novo conteúdo. É análogo aos métodos construtores de diversas linguagens de programação como Java e C#, servindo para criar novos objetos, que devem ser enviados no corpo da requisição.

Exemplos comuns de requisições POST a web APIs podem ser como as abaixo (a segunda linha é o corpo da requisição), onde estamos salvando um novo cliente no servidor:

```
POST /clientes  
Nome=Luiz&Idade=29&Cidade=Gravataí&UF=RS
```

```
POST /clientes  
{nome:"Luiz", idade: 29, cidade: "Gravataí", uf: "RS"}
```

É esperado que o retorno de um POST seja um "200 OK" no cabeçalho e opcionalmente podem ter dados no corpo também. Algumas web APIs retornam o dado que acabaram de salvar, para mostrar ao requisitante que entendeu completamente a requisição e que o dado foi completamente salvo.

Junto ao GET, são os únicos dois verbos HTTP que podemos usar como atributo METHOD de um HTML FORM. Para disparar requisições com outros verbos temos de fazer via programação ou através de ferramentas como POSTMAN, Fiddler, etc.

PUT

Esse é o verbo HTTP para substituir um conteúdo existente no servidor (update completo). O conteúdo que substituirá o antigo deve ser enviado no corpo da requisição em formato pré-acordado. Para que o servidor saiba qual registro deve ser substituído, é passado junto à URL da requisição o ID do objeto ou chave primária análoga.

Exemplos de requisições PUT a web APIs podem ser como abaixo, onde requisito a substituição do cliente de ID 1 pelos dados do corpo da requisição (segunda linha):

```
PUT /clientes/1  
Nome=Luiz Fernando&Idade=29&Cidade=Gravataí&UF=RS
```

```
PUT /clientes/1  
{nome: "Luiz Fernando", idade: 29, cidade: "Gravataí", uf:  
"RS"}
```

É esperado que o retorno de um PUT seja um 200 OK, mas podem haver variações no corpo da resposta de API para API.

PATCH

Esse é o verbo HTTP para atualizar parcialmente um conteúdo existente no servidor (update parcial), semelhante aos métodos

setters de algumas linguagens, como um `setNome` do Java, por exemplo. O conteúdo que atualizará o antigo deve ser enviado no corpo da requisição em formato pré-acordado. Para que o servidor saiba qual registro deve ser atualizado, é passado junto à URL da requisição o ID do objeto ou chave primária análoga.

Exemplos de requisições PATCH a web APIs podem ser como abaixo, onde estou requisitando a atualização somente do nome do cliente com ID 1, os demais dados permanecerão inalterados:

```
PATCH /clientes/1  
Nome=Luiz Fernando
```

```
PATCH /clientes/1  
{nome: "Luiz Fernando"}
```

É esperado que o retorno de um PATCH seja um 200 OK, mas podem haver variações no corpo da resposta de API para API.

DELETE

Verbo HTTP para excluir um conteúdo do servidor. É análogo ao método `dispose()` e destrutores presentes em algumas linguagens de programação, servindo para excluir um objeto inteiro da fonte de dados. Para que o servidor saiba qual registro deve ser excluído, é passado junto à URL da requisição o ID do objeto ou chave primária análoga.

Não há necessidade de passar dados no corpo da requisição, pois o DELETE baseia-se na chave primária:

```
DELETE /clientes/1
```

É esperado que o retorno de um DELETE seja um 200 OK.

Existem outros verbos HTTP, mas esses com certeza são os mais utilizados nos padrões de comunicação como o REST. Uma web

API é dita RESTful se ela segue todas restrições impostas pelo protocolo.

Formato de Dados

Além disso, você deve ter notado que na maioria dos exemplos foi utilizado dois formatos de dados nos corpos das requisições o x-www-form-urlencoded (chave1=valor1&chave2=valor2) e o JSON. Enquanto que o x-www-form-urlencoded é o formato padrão de envio através de HTML FORM, o JSON é o mais utilizado quando o assunto é transferência de dados via web APIs.

O JavaScript Object Notation, mais conhecido pela sua abreviação JSON, é um formato criado inicialmente para a linguagem JavaScript mas que logo se tornou muito utilizado para transmissão rápida e dinâmica de dados pela Internet. JSON é extremamente flexível a ponto de permitir arquivos tão pequenos quanto CSV e menores que flat-files (arquivos organizados por posições ou separadores). Mas não pense que eles são “bagunçados”, um JSON pode ser tão estruturado quanto um XML, com a vantagem de gerar um arquivo muito menor devido ao problema de repetição de tags do XML.

Pode-se representar qualquer estrutura de dados em um objeto JSON, uma vez que são "orientados a objetos", e não somente tabelas como os formatos de documentos em geral. Desta maneira, podemos criar arrays, objetos com tipos definidos, associação entre objetos (como agregação e composição), campos opcionais e assim por diante.

JSON é largamente utilizado na web e principalmente em APIs de integração com sites, como no caso do Facebook e Twitter. Devido à sua popularidade existem diversas implementações de bibliotecas que fazem o encode/serialização (escrita) e decode/desserialização (leitura) de dados neste formato para todas as linguagens modernas e muitas linguagens antigas também.

Independente do formato escolhido para formatar a transferência dos seus dados, o mais importante é a consistência. Se for usar

JSON, use JSON para o corpo das requisições e das respostas. Se for usar x-www-form-urlencoded, use-o em tudo.

Criando uma Web API

Mesmo que seu objetivo não seja criar aplicações web com Node.js, e sim apenas Web APIs, não recomendo que você leia diretamente esta seção. Leia este livro na ordem em que ele foi escrito para um maior aproveitamento do conteúdo pois os capítulos e seções não possuem essa disposição ao acaso.

Caso realmente não se interesse por front-end, os capítulos iniciados com este título podem ser ignorados para a construção de web-apis, mas os demais não, e isso inclui o de MongoDB, mesmo que não venha a utilizar este banco de dados, pois nele mostro questões importantíssimas do framework Express, que usaremos aqui novamente.

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

Preparando o projeto

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

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

Código 7.1: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
npm init
```

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

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

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

Código 7.2: disponível em <https://www.luiztools.com.br/livro-node-fontes>

npm i express mongodb body-parser

Note que desta vez usei 'i' ao invés de 'install' e que informei o nome de mais de um pacote de uma vez só, o que é perfeitamente possível com o NPM.

Com essas dependências instaladas, vamos começar a programar!

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

Criando a estrutura básica

Vamos começar nosso app.js definindo algumas variáveis locais através do carregamento de alguns módulos:

Código 7.3: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
const port = 3000; //porta padrão
```

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

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

Código 7.4: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

Código 7.5: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

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

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

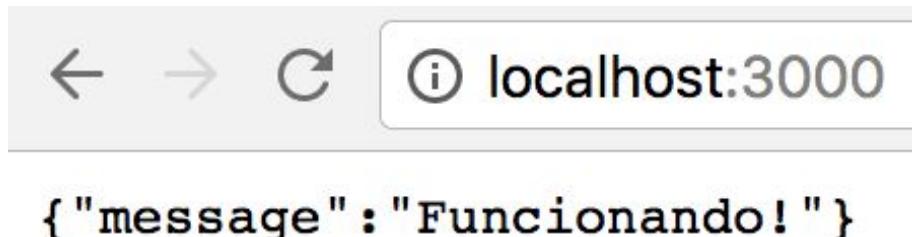
E por fim, vamos escrever o código que inicializa o nosso servidor no final do app.js:

Código 7.6: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

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



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

Programando a API

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

Para cada operação desejamos oferecer através da nossa API devemos criar uma rota em nosso app.js, junto ao router.get que já

deixei lá por padrão no tópico anterior (lembra?).

Usaremos apenas o arquivo app.js neste exemplo de web API, então não ficarei repetindo isso o tempo todo. Comece adicionando as seguintes linhas bem no topo dele, explicarei na sequência.

Código 7.7: disponível em <https://www.luiztools.com.br/livro-node-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;
}
```

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.

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.

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

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 7.8: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
router.get('/clientes', 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 último 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 API 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/clientes> e maravilhe-se com o resultado. Como o acesso do navegador é sempre GET, você deve ver um array JSON com todos os clientes do seu banco de dados nele:



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

Mas e se quisermos oferecer um 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 7.9: disponível em <https://www.luiztools.com.br/livro-node-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 7.10: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
router.get('/clientes/: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());
  }
  catch(ex){
    console.log(ex);
    res.status(400).json({erro: `${ex}`});
  }
})
```

})

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/clientes/id trocando 'id' pelo respectivo id que deseja pesquisar. O retorno no navegador deve ser um único objeto JSON.

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

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

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

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

Código 7.11: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
router.post('/clientes', async function(req, res, next){
  try{
    const customer = req.body;
    const db = await connect();
    res.json(await db.collection("customers").insertOne(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 /clientes. Na sequência, pegamos o body da requisição onde está o JSON do cliente a ser salvo na base de dados. Obviamente no mundo real você irá querer colocar validações, tratamento de erros e tudo mais.

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

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

Temos diversas alternativas no mercado, sendo que vou falar de uma aqui: usando o POSTMAN. Caso não tenha baixado e instalado na sua máquina 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:

The screenshot shows the Postman application interface. At the top, there are tabs for Runner, Import, and Builder, with Builder being the active tab. Below the tabs is a toolbar with icons for opening a new tab, switching environments, and a sync status. The main area has a header bar with 'localhost:3000/cliente' and a 'New Tab' button. There are also buttons for '+' and '...' in the header. To the right of the header is a dropdown for 'No Environment'. Below the header, there's a row with 'GET' (selected), 'Enter request URL', 'Params', and a large blue 'Send' button. Underneath this row are tabs for Authorization, Headers, Body, Pre-request Script, and Tests, with 'Authorization' being the active tab. A dropdown menu labeled 'Type' shows 'No Auth' selected. At the bottom, there's a large text area labeled 'Response'.

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

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

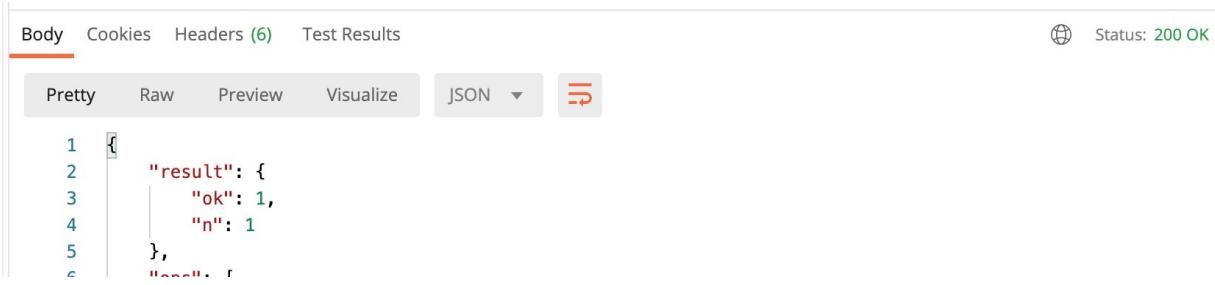
Logo abaixo temos algumas abas. Vá na aba Headers (cabeçalhos) e adicione uma linha com a seguinte configuração: key = Content-Type e value = application/json. Isso diz ao POSTMAN que essa requisição está enviando dados no formato JSON em seu corpo (body).

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

The screenshot shows the Postman application interface with a POST request configuration. The 'Body' tab is selected. In the 'Body' section, the 'raw' option is selected, and the JSON content is: {"nome": "Postman", "idade": 10, "uf": "SP"}. Other options like 'form-data' and 'x-www-form-urlencoded' are available but not selected.

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:



The screenshot shows the POSTMAN interface with the 'Body' tab selected. At the top, there are tabs for 'Body', 'Cookies', 'Headers (6)', and 'Test Results'. On the right, it says 'Status: 200 OK'. Below the tabs, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON'. The JSON response is displayed in a code editor-like area with line numbers 1 through 6. The JSON content is as follows:

```
1  {
2   "result": {
3     "ok": 1,
4     "n": 1
5   },
6   "error": null
```

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

Mas será que funcionou mesmo?

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

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

Código 7.12: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

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 app.js:

Código 7.13: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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:

The screenshot shows the Postman application interface. At the top, there are tabs for 'localhost:3000/cliente' and 'localhost:3000/clientes/59ac8350e07d4f10cf6a74f6'. The main area shows a 'PUT' request being sent to the latter URL. The 'Body' tab is selected, showing a JSON payload: { "nome": "Postman", "idade": 20, "uf": "SP" }. Below the request, the response status is '200 OK' and the time taken is '39 ms'. The response body is displayed in JSON format: { "message": "Cliente atualizado com sucesso!" }.

Incluí no mesmo print acima a resposta à minha requisição.

Nota: Você pode obter o mesmo resultado usando cURL:

Código 7.14: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

Update: check!

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

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

Depois faça a rota no app.js:

Código 7.15: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

Para testar no POSTMAN é muito simples, troque o verbo para PATCH (considerando que você ainda tem o exemplo de PUT aberto) e no body da requisição, coloque um objeto JSON contendo apenas os dados que você deseja alterar do respectivo cliente (o que possui o id informado na URL).

The screenshot shows the Postman interface with a PATCH request to update a customer. The request URL is `localhost:3000/clientes/59ac8350e07d4f10cf6a74f6`. The Body tab contains the following JSON payload:

```

1 {"nome": "POSTMAN"}

```

The response status is `200 OK` and the message is `"Cliente atualizado com sucesso!"`.

No exemplo acima, apenas troquei o nome do customer de Postman para POSTMAN.

Nota: via cURL você pode testar usando:

Código 7.16: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```

curl -X PATCH -d "{\"nome\":\"POSTMAN\"}"
http://localhost:3000/clientes/59ac8350e07d4f10cf6a74f6

```

Para encerrar a construção da nossa API REST, falta nosso delete que é muito simples e rápido de fazer:

Código 7.17: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```

router.delete('/clientes/: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.

The screenshot shows the POSTMAN interface with the following details:

- URL: `localhost:3000/clientes/59ac8350e07d4f10cf6a74f6`
- Method: `DELETE`
- Headers:
 - `Content-Type`: `application/json`
- Body (Pretty):

```
1 {  
2   "message": "Cliente excluído com sucesso!"  
3 }
```
- Status: `200 OK`
- Time: `31 ms`

Nota: e em cURL:

Código 7.18: disponível em <https://www.luiztools.com.br/livro-node-fontes>

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

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

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Node.js e Microservices

Meu livro focado em Web APIs e a sua evolução: os micro serviços, com muito material para enriquecer o seu conhecimento nesta arquitetura, incluindo escala e segurança.

<https://www.luiztools.com.br/livro-node-ii>

RabbitMQ

O que acontece quando nossas APIs recebem mais requests do que elas podem suportar? Com RabbitMQ, isso não se torna um problema!

<https://www.luiztools.com.br/post/processamento-assincrono-de-tarefas-com-filas-no-rabbitmq-e-node-js/>

Artillery.io

Aprenda como fazer testes de carga nas suas web APIs com o Artillery.

<https://www.youtube.com/watch?v=bsDtsWbYqto>

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

8 Front-end: JavaScript Client-Side

*Not all roots are buried down in the ground some are at the top of
the tree.*
— Jinvirle

JavaScript é uma linguagem de programação interpretada originalmente implementada como parte dos navegadores web para que scripts pudessem ser executados do lado do cliente e interagissem com o usuário sem a necessidade deste script passar pelo servidor, controlando o navegador, realizando comunicação assíncrona e alterando o conteúdo do documento exibido.

É atualmente a principal linguagem para programação client-side em navegadores web e foi concebida para ser uma linguagem script com orientação a objetos baseada em protótipos, tipagem fraca e dinâmica e funções de primeira classe. Possui suporte à programação funcional e apresenta recursos como clojures e funções de alta ordem comumente indisponíveis em linguagens populares como Java e C++.

Atualmente considera-se o JavaScript como sendo a linguagem de programação mais utilizada no mundo, sendo padronizada pela Ecma International sob o nome de ECMAScript, nas especificações ECMA-262 e ISO/IEC 16262.

Em capítulos anteriores aprendemos o básico da sintaxe do JavaScript, bem como aprendemos a fazer uma série de provas de conceito usando o terminal interativo Node REPL e o básico de Node.js. Neste capítulo aprenderemos como utilizar o JavaScript como linguagem client-side para tornar as nossas páginas HTML mais dinâmicas.

Como client-side entenda do lado do cliente, ou seja, no front-end, na interface do navegador que está rodando na máquina do seu usuário. O oposto disso seria server-side, ou no lado do servidor, código que roda no datacenter, longe da máquina do cliente.

Atenção: assim como fizemos no outro capítulo sobre front-end, começaremos os estudos de JavaScript client-side sem mexer com Node.js. Abra um editor de texto qualquer e faça os exercícios sem estar em um projeto Node, apenas com um bando de arquivos em uma pasta. Ao término do capítulo juntaremos as duas pontas.

A tag SCRIPT

Utiliza-se a tag HTML SCRIPT para criar blocos de código Javascript ou para carregar um arquivo com extensão .JS que exista no seu projeto. Ambos exemplos são mostrados abaixo.

Código 8.1: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!--... código HTML qualquer ...-->
<script>
    //código Javascript vai aqui
</script>
<!-- ... mais código HTML ... ->
```

No exemplo acima a tag script é usada em formato container. Tudo que for colocado dentro dela será interpretado como código Javascript.

Código 8.2: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!-- ... código HTML qualquer ... -->
<script src="funcoes.js" />
<!-- ...código HTML qualquer ... -->
```

No exemplo acima a tag SCRIPT fecha em si mesma e usa o atributo SRC para apontar para um arquivo com extensão .JS presente no próprio projeto ou na internet (neste caso deve referenciar a URL absoluta do arquivo).

Nota: o carregamento de scripts JS na página HTML possui comportamento bloqueante na requisição. Ou seja, até que o script termine de ser carregado o browser não carrega mais nada. Assim, caso seu script seja grande (algumas dezenas de KB por exemplo), é sempre indicado referenciá-lo apenas no final do documento HTML, o mais próximo possível da tag </BODY> (fecha-body) para

evitar que o usuário tenha de esperar demais pelo carregamento para começar a usar a página.

Document Object Model

O Document Object Model (DOM) é uma interface de programação para os documentos HTML e XML. Representa a página de forma que os programas possam alterar a estrutura do documento, alterar o estilo e conteúdo. O DOM representa o documento com nós e objetos, dessa forma, as linguagens de programação podem se conectar à página.

Uma página da Web é um documento. Este documento pode ser exibido na janela do navegador ou como a fonte HTML. Mas é o mesmo documento nos dois casos. O DOM (Document Object Model) representa o mesmo documento para que possa ser manipulado. O DOM é uma representação orientada a objetos da página da web, que pode ser modificada com uma linguagem de script como JavaScript.

O DOM não é uma linguagem de programação, mas sem ela, a linguagem JavaScript não teria nenhum modelo ou noção de páginas da web, documentos HTML, documentos XML e suas partes componentes (por exemplo, elementos). Cada elemento de um documento - o documento como um todo, o cabeçalho, as tabelas do documento, os cabeçalhos da tabela, o texto nas células da tabela - faz parte do modelo de objeto do documento desse documento, para que todos possam ser acessados e manipulados usando o método DOM e uma linguagem de script como JavaScript.

Os padrões W3C DOM e WHATWG DOM são implementados na maioria dos navegadores modernos. Muitos navegadores estendem o padrão; portanto, é necessário ter cuidado ao usá-los na Web, onde os documentos podem ser acessados por vários navegadores com diferentes DOMs.

Como usar

Você não precisa fazer nada de especial para começar a usar o DOM. Navegadores diferentes têm implementações diferentes do DOM, e essas implementações exibem graus variados de

conformidade com o padrão DOM real (um assunto que tentamos evitar nesta documentação), mas todo navegador usa um modelo de objeto de documento para tornar as páginas da web acessíveis via JavaScript.

Quando você cria um script - seja embutido em um elemento(tag) <script> ou incluído na página da web por meio de uma instrução de carregamento de script - você pode começar imediatamente a usar a API para o document, visando manipular o próprio documento ou obter os filhos desse documento, que são os vários elementos na página da web. Sua programação DOM pode ser algo tão simples quanto o exemplo seguinte, que exibe uma mensagem de alerta usando a função alert() da função window ou pode usar métodos DOM mais sofisticados para criar realmente novo conteúdo, como nos comando que você verá a seguir.

Nota: sugiro criar uma página HTML, colocar uma tag SCRIPT nela e testar cada um dos recursos abaixo para entender realmente o que eles fazem. Mais para frente teremos um exercício, assim como já vínhamos fazendo no capítulo anterior.

O objeto window

O objeto window é o contâiner por fora do DOM, contém informações e funções que afetam a janela atual do navegador:

window.location.href

Pega a URL atual do navegador ou faz com que o navegador acesse a URL especificada. Experimente usar um console.log(window.location.href).

window.location.search

Pega a querystring atual do navegador. Experimente usar um console.log(window.location.search).

window.location.reload()

Atualiza a página atual (F5).

window.history

Acessa o objeto de histórico do navegador, permite inclusive voltar para a página anterior.

setTimeout(function, delay)

Função nativa do objeto window que atrasa o disparo de uma função JS em um tempo definido em milissegundos.

setInterval(function, interval)

Semelhante ao setTimeout, mas dispara a mesma function a cada x milissegundos. Como retorno à chamada deste método é passado o ID do timer, que pode ser cancelado usando a função clearInterval(id).

setImmediate(function)

Executa a function exatamente agora, mas por uma thread em background. Muito útil para processamento não-bloqueante.

Popups

Popups são aquelas janelinhas, muitas vezes irritantes, que saltam no navegador e são fornecidas pela API DOM da janela do navegador (window). Se bem utilizadas podem ser muito úteis para exibir informações ao usuário, por exemplo.

alert(message)

Exibe um popup com uma mensagem dentro e um botão de ok. É um atalho para window.alert(message).

confirm(message)

Exibe um popup com uma mensagem dentro (geralmente um questionamento) e dois botões: ok e cancelar. Caso o usuário clique em ok, esta função irá retornar true, caso contrário false. É um atalho para window.confirm(message).

O objeto document

O objeto document contém informações e funções que afetam a página HTML atual que está sendo exibida na janela (window)

navegador.

document.getElementById(id)

Busca no documento HTML o objeto com o atributo Id passado por parâmetro (string). Este objeto resultante tem os mesmos atributos e funções do objeto original. Note que o atributo Id não é a mesma coisa que o atributo name. O id serve justamente para diferenciar os objetos HTML entre si (não temos dois objetos com mesmo id), enquanto que o name serve para definir o nome dos valores a serem passados dentro de um FORM (ou seja, não possui sentido em tags que não sejam INPUTs e derivados).

Mais à frente ver mais sobre o document, assim que exercitarmos o que acabamos de ver.

Eventos JavaScript

É possível associar funções JS à eventos que possam ocorrer com seus elementos HTML. Assim, quando um botão clicar ou quando digitarmos sobre um campo de texto, algo interessante pode ser executado.

Para configurar um evento em um elemento HTML basta fazer o mesmo que fazíamos com os atributos dos elementos: dentro da tag em que queremos configurar o evento, adicionamos o nome do evento seguido do sinal de igualdade e entre aspas colocamos a chamada à função JS que deve existir em nossa página (incluindo parâmetros, se houverem). Isso ficará mais claro no detalhamento à seguir.

Nota: uma dica de como testar os seus códigos JS rapidamente é usando o site <http://jsfiddle.net> que permite criar código HTML, JS e CSS e testá-lo diretamente no navegador.

onclick

O evento onclick pode ser configurado em qualquer elemento HTML que possa ser clicado. Quando o click do mouse ocorrer sobre o elemento, a função JS configurada será disparada. Veja o exemplo à seguir de uma função JS que apenas exibe uma mensagem (essa função deve estar em um arquivo):

Código 8.3: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function exibirMensagem(){
    alert('funcionou!');
}
```

Agora veja um botão HTML com o evento onclick configurado para a função acima:

Código 8.4: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<input type="button" value="Teste" onclick="exibirMensagem()" />
```

Assim, quando este botão for clicado, a mensagem ‘Funcionou!’ irá aparecer no navegador do cliente. Note que usamos um TYPE=”BUTTON” ao invés de um SUBMIT. Isso porque botões do tipo SUBMIT iriam submeter o FORM HTML ao invés de simplesmente exibir a mensagem, estragando nosso teste. Entretanto, não pense que o onclick funciona apenas com botões, ele funciona com qualquer elemento visível que possa ser clicado com o mouse.

Analogamente, temos o evento ondblclick que é quando o elemento HTML recebe um clique-duplo do mouse.

onchange

O evento onchange pode ser configurado em qualquer elemento HTML que possa ter uma opção selecionada. Quando o usuário escolhe uma nova opção em um SELECT, esse evento é disparado, por exemplo.

Código 8.5: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function exibirUF(uf){  
    alert(uf)  
}  
  
<select onchange="exibirUF(this.value)">  
    <option value="RS">Rio Grande do Sul</option>  
    <option value="SC">Santa Catarina</option>  
    <option value="PR">Paraná</option>  
</select>
```

Note que esta função JS é ligeiramente diferente da anterior, ela recebe um parâmetro UF, que é o texto que será exibido no alert.

Quando configuramos qualquer evento podemos passar parâmetros para a função JS que será chamada e neste caso optei por passar o atributo value do próprio elemento HTML. Com isso, ao selecionar a opção ‘Santa Catarina’, a uf ‘SC’ irá aparecer na mensagem.

onmouseenter e onmouseleave

Quando queremos mapear os eventos do ponteiro do mouse sobre nossos elementos HTML podemos usar os eventos onmouseenter e onmouseleave. O primeiro é disparado quando o ponteiro do mouse entra na área do elemento, e o segundo quando o ponteiro do mouse sai da área do elemento.

Código 8.6: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function exibirMensagem(msg){  
    alert(msg);  
}  
  
<div onmouseenter="exibirMensagem('entrou!')"  
onmouseleave="exibirMensagem('saiu!')">Teste</div>
```

No exemplo acima, quando o usuário passar o mouse sobre a DIV a mensagem exibida será ‘entrou’ e quando o ponteiro do mouse sair da área da DIV aparecerá ‘saiu’. Note que aqui usamos apenas uma função JS mapeada em dois eventos diferentes, apenas mudando a string literal que passamos por parâmetro. Note também que usamos aspas simples para passar o parâmetro string, isso porque já havíamos usado aspas duplas na definição dos eventos.

onfocus e onblur

Quando queremos mapear os eventos de ganho de foco e perda de foco em um elemento HTML, usamos os eventos onfocus, que ocorre quando um elemento ganha o foco do usuário, e onblur, quando o elemento HTML perde o foco do usuário. Por ‘ganhar’ e ‘perder’ o foco entenda que o elemento recebe o foco quando o

cursor para sobre ele, seja com o uso do mouse ou do teclado (usando TAB).

Código 8.7: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function exibirMensagem(msg){  
    alert(msg);  
}  
  
<input type="text" onfocus="exibirMensagem('entrou!')"  
onblur="exibirMensagem('saiu!') />
```

No exemplo acima nosso campo de texto exibirá o alert ‘entrou’ quando ganhar o foco e a mensagem ‘saiu’ quando ele perder o foco. Experimente no seu navegador usando a tecla TAB e usando o mouse para ver como o comportamento é o mesmo.

onkeypress, onkeydown e onkeyup

Também podemos mapear os eventos do teclado quando o foco está sobre nosso elemento HTML. O evento onkeydown é disparado quando uma tecla está atualmente sendo pressionada no teclado, o onkeyup quando uma tecla está atualmente sendo solta no teclado e o onkeypress corresponde à ação completa de pressionar e soltar uma tecla.

Estes eventos são especialmente úteis em campos de texto, para verificar em tempo real o que fazer com as informações que o usuário está digitando. Com isso é possível criarmos validações e máscaras em nosso campo. Estes eventos disparam funções JS comuns, porém através do objeto event podemos saber qual a tecla pressionada e opcionalmente alterar o conteúdo do campo de texto ou até invalidar a última tecla digitada, como abaixo.

Código 8.8: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function validarNumeros(e){  
    var unicode=e.charCodeAt : e.keyCode;  
    if (unicode!=8){ //8 = backspace  
        if (unicode<48||unicode>57) //se não é um número  
            return false; //disable key press  
    }  
}  
  
<input type="text" onkeypress="return validarNumeros(event)" />
```

A função validarNumeros retorna false quando a tecla digitada não é um número. Como o evento onkeypress está mapeado para retornar o resultado de validarNumeros, se a função retornar false a tecla digitada não irá aparecer no campo de texto, isso porque o onkeypress acontece antes da tecla realmente aparecer no campo.

Note também que passamos o objeto event por parâmetro na função, que possui informações sobre o evento em si, incluindo o caracter digitado ou o código da tecla, de acordo com o que foi digitado.

Existem muitos outros eventos JS que você pode mapear, como o arrastar e soltar do mouse (onmousedown e onmouseup), o redimensionar (onresize) de elementos, o carregar/descarregar (onload e onunload) do body HTML e muitos outros.

Vamos exercitar o que vimos até aqui?

Calculadora em HTML+JS

O que acha de fazermos uma calculadora que roda no navegador?

Crie uma página HTML contendo um INPUT de texto para o display, botões para os número de 0 a 9 e para as operações elementares. Também inclua um botão para a igualdade e um de limpar. Ela deve se comportar exatamente como uma calculadora digital que estamos acostumados a usar.

Em termos de aparência, tente construir da seguinte forma:

0			
7	8	9	*
4	5	6	-
1	2	3	+
C	0	=	/

Não se preocupe tanto com a beleza da interface. Veremos estilização de HTML mais à frente para dar uma aparência mais profissional às nossas aplicações web.

Se não conseguir construir por si próprio este layout HTML, use os fontes abaixo:

Código 8.9: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Calculadora</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <table>
      <tr>
        <td colspan="6"><input type="text" id="display" value="0" disabled></td>
      </tr>
      <tr>
```

```

<td><input type="button" value="7" /></td>
<td><input type="button" value="8" /></td>
<td><input type="button" value="9" /></td>
<td><input type="button" value="*" /></td>
</tr>
<tr>
    <td><input type="button" value="4" /></td>
    <td><input type="button" value="5" /></td>
    <td><input type="button" value="6" /></td>
    <td><input type="button" value="-" /></td>
</tr>
<tr>
    <td><input type="button" value="1" /></td>
    <td><input type="button" value="2" /></td>
    <td><input type="button" value="3" /></td>
    <td><input type="button" value="+" /></td>
</tr>
<tr>
    <td><input type="button" value="C" /></td>
    <td><input type="button" value="0" /></td>
    <td><input type="button" value="=" /></td>
    <td><input type="button" value="/" /></td>
</tr>
</table>
</body>
</html>

```

Agora vamos aos comportamentos. Cada vez que um botão numérico for pressionado, o número do botão deve ser adicionado à direita do input/display. Note que coloquei o id display no input, para podermos manipulá-lo via JavaScript e também coloquei o atributo disabled nele, para que não seja permitido digitar diretamente nele (o usuário terá de usar os botões).

Para fazer isso, abra uma tag SCRIPT antes de </body> e vamos criar uma função de atualização do display nele:

Código 8.10: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<script>
  function atualizarDisplay(btn){
    const display = document.getElementById('display');
    if(display.value.length === 9) return;
    if(display.value === '0') display.value = btn.value;
    else display.value += btn.value;
  }
</script>
```

O que fazemos aqui: a função recebe um objeto btn que representa um botão por parâmetro. Daí usamos document.getElementById para carregar o input display em uma variável local. Com essa variável local display podemos ler e escrever no value do display, alterando o seu texto.

Ao invés de editar o texto diretamente, coloquei uma verificação para impedir que sejam digitados mais de 9 números e também fiz uma lógica para que caso o display esteja zerado, que o primeiro número vai substituir esse zero. Por fim, faço com que o value do botão seja adicionado ao value do display.

Mas como fazer para que essa função seja adicionada a todos os botões numéricos?

Basta usar o atributo onclick dos botões que permite dizer qual a função JS será disparada toda vez que o botão for clicado. O uso de 'this' no parâmetro faz com que o próprio botão seja passado por parâmetro, como no exemplo abaixo do botão 7 (você deve replicar para todos os numéricos):

Código 8.11: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<td><input type="button" value="7" onclick="atualizarDisplay(this)" /></td>
```

Para testar é muito simples, abra a sua calculadora.html no navegador e pressione os botões numéricos, verá que já estão funcionando.

Para fazer o botão de limpar (C de Clear) é bem simples, basta criar uma função que zera o display novamente (coloque essa função no mesmo bloco SCRIPT da outra função):

Código 8.12: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function limparDisplay(){  
    document.getElementById('display').value = '0';  
}
```

E depois referenciar esta função no atributo onclick do botão C:

Código 8.13: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<td><input type="button" value="C" onclick="limparDisplay()" />  
</td>
```

Teste e verá que funciona perfeitamente.

Agora é a hora de fazer as operações funcionarem. Para isso, vamos criar uma função para o click dos botões de operação, bem como algumas variáveis:

Código 8.14: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
let operador = " ";  
let valor1 = 0;  
function atualizarOperacao(btn){
```

```
const display = document.getElementById('display');
operador = btn.value;
valor1 = parseInt(display.value);
display.value = '0';
}
```

Aqui guardamos a operação que foi digitada em uma variável, bem como o valor numérico que estava no display no momento que o operador foi pressionado. Vincule esta função aos quatro botões de operação, como neste exemplo com o botão de multiplicação:

Código 8.15: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<td><input type="button" value="*" onclick="atualizarOperacao(this)" /></td>
```

Visualmente isso não tem efeito algum, exceto zerar o display e permitir que você digite outro número enquanto que o primeiro ficou armazenado em memória. Para fazer com que as operações funcionem, vamos criar mais uma função, que usaremos para o botão de igualdade:

Código 8.16: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function calcularOperacao(){
    const display = document.getElementById('display');
    const valor2 = parseInt(display.value);
    valor1 = eval(valor1+operador+valor2);
    display.value = valor1;
    operador = "=";
}
```

Essa função merece uma atenção especial!

Primeiro, carregamos o input de display e lemos o número atualmente nele (já convertendo para numérico). Depois, usamos a função eval do JS para calcular a equação matemática composta pelo valor1 (que guardamos anteriormente), o operador matemático e o valor2 (que acabamos de ler). Guardamos o resultado eval no valor1 caso o usuário queira encadear operações e por fim limpamos a variável operador.

Para fazer o botão de igualdade funcionar, vamos vincular esta última função no onclick dele:

Código 8.17: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<td><input type="button" value="=" onclick="calcularOperacao()" />
</td>
```

E com isso você terá uma calculadora web perfeitamente funcional!

Nota: o HTML não armazena estado, logo, a cada refresh do seu navegador ou troca de página você perderá os dados que tinha em memória. Isso é normal da web.

Vamos colocar uma perfumaria? É muito chato não poder usar o teclado do computador para usar a calculadora, certo?

Então vamos criar uma última função no bloco SCRIPT da nossa calculadora.html que vai pegar as teclas digitadas do teclado e decidir se elas vão ou não aparecer no display:

Código 8.18: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function manipularTeclado(){
    if(/[0-9]/.test(event.key))
        atualizarDisplay({value: event.key});
}
```

event é um objeto que representa o evento JS que acabou de ser disparado. Do jeito que está não dá pra entender direito o que event vai representar mas confie em mim. Já event.key é um atributo que existe apenas quando o evento é de teclas pressionadas.

O que fiz ali no if é uma expressão regular. Existem livros apenas sobre a construção e uso de expressões regulares como o fantástico Expressões Regulares: Uma Abordagem Divertida, de Aurélio Marinho Jargas, mas basta entender por ora que uma expressão regular é um padrão de texto sobre o qual testamos para ver se outro texto está de acordo.

Quando uso barras no JavaScript ele entende que vou definir uma expressão regular entre as barras. Aqui no caso usei [0-9] que é uma expressão regular bem comum para validar se um texto é um número de 0-9. O teste é feito usando a função test, passando por parâmetro o texto que queremos validar. No nosso caso, somente entrará no if se a tecla que foi pressionada é um número.

Para que essa função passe a valer em nossa calculadora, devemos colocar ela no atributo onkeypress do body da sua página. Sim, isso mesmo, no body. Estamos definido no body, essa função irá manipular TODOS os pressionamentos de teclado do usuário para ver se algum dele é um número que possa ser colocado no display:

Código 8.19: disponível em <https://www.luiztools.com.br/livro-node-fontes>

<body onkeypress="manipularTeclado()">

Teste agora e verá que, com exceção da aparência, nossa calculadora ficou bem profissional.

O JavaScript client-side é muito poderoso e mesmo antes da invenção do Node.js ele já era uma linguagem muito utilizada

mundialmente, uma vez que a tecnologia padrão para o client-side dos browsers.

Agora vamos aprender como fazer alguns truques bem interessantes usando recursos de manipulação do DOM. Sim, eu vou explicar o que é DOM também...

Manipulando o DOM

Vimos o básico do DOM anteriormente e entendemos do que se trata, agora chegou a hora de extrair mais poderes dele usando JavaScript.

Seletores

O primeiro e mais fundamental conceito de manipulação do DOM é o de seletores. Isso porque, quando vamos manipular algo, a primeira coisa que devemos aprender é como encontrar o elemento que vamos manipular.

Para fazer isso, geralmente os programadores têm de definir um id no elemento e depois carregá-lo usando `document.getElementById`, como já citei antes.

Considere o seguinte trecho HTML:

Código 8.20: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<!DOCTYPE html>
<html>
  <head><meta charset="utf-8" /></head>
  <body>
    <input type="text" name="nome" id="txtNome" />
  </body>
</html>
```

Para então exercitar os comandos abaixo.

document.getElementById(id)

Busca no documento HTML o objeto com o atributo Id passado por parâmetro (string). Este objeto resultante tem os mesmos atributos e funções do objeto original. Experimente usar `document.getElementById("txtNome")` no console do navegador com esta página aberta e verá ser retornada a respectiva tag input.

Mas além desta função, temos:

document.getElementsByName(name)

Busca no documento HTML todos os objetos com o name referenciado. Experimente usar document.getElementByName("nome") no console do navegador com esta página aberta e verá ser retornada ambas tags input, pois elas possuem o mesmo nome.

document.getElementsByTagName(name)

Quando queremos trazer todos os elementos do HTML que sejam de uma mesma tag, usamos este seletor. Experimente trazer todos inputs ou algum outro elemento à sua escolha, como em document.getElementsByTagName("input").

document.getElementsByClassName(name)

Ainda não vimos CSS, mas existe outro atributo chamado class, que também pode ser usado como seletor do DOM.

E por último, temos a função mais poderosa...

document.querySelector(filtro) e document.querySelectorAll(filtro)

Tudo o que fizemos com as demais funções e coisas ainda mais poderosas podem ser feitas com as funções acima, sendo que a primeira sempre retorna apenas um elemento e a segunda pode retornar vários. Em ambas, devemos passar por parâmetro o filtro a ser utilizado. Este filtro pode ser (usando o HTML acima como exemplo):

- um id, prefixado com sustenido: ("#txtNome");
- tipo de tag, sem prefixo algum: ("input");
- um atributo definido após a tag, entre colchetes: ("input[name='nome']");
- começo de um valor de atributo, usando circunflexo: ("input[name^='nom']");

- final de um valor de atributo, usando cifrão:
("input[name\$='ome']") ;
- contendo um valor de atributo, usando asterisco:
("input[name*= 'om']");
- um elemento dentro de outro elemento: ("body > input");
- mais de um elemento ao mesmo tempo: ("head,body");

Todos os seletores exemplificados acima carregam o elemento input presente no HTML de exemplo.

O uso da função de seleção irá retornar um objeto JS contendo o elemento HTML e todas suas propriedades e funções. Além disso, a busca é feita em todo o documento, logo, se mais de um elemento atender ao filtro, todos serão retornados, motivo pelo qual devemos sempre buscar utilizar filtros específicos, como por id, por exemplo.

Uma vez que selecionamos o elemento que queremos, podemos manipulá-lo usando os atributos e funções nativos do JS ou nativas do DOM Element, como as funções a seguir.

Propriedades de Elemento

Todo elemento HTML/DOM possui uma lista de propriedades que é muito grande para eu referenciar aqui. No próprio console do Google Chrome, quando você manipula o DOM e obtém um elemento, ele tem um recurso de autocomplete que lhe mostra todas as possibilidades daquele elemento.

Ainda assim, tentando trazer as propriedades mais comuns e úteis, temos:

element.value

Esta propriedade, existente em inputs, define ou retorna o conteúdo do atributo value dele. Em alguns inputs, o value é o texto no seu interior. Em outros, é o item selecionado.

element.getAttribute(attributeName) e setAttribute(attributeName, newValue)

Esta propriedade retorna o conteúdo de um atributo da tag HTML, como por exemplo o src de uma imagem ou o href de uma âncora.

O oposto de getAttribute é o setAttribute, que espera o nome do atributo e o seu novo valor como parâmetros.

element.style

Esta propriedade define tudo relacionado à aparência do elemento. Qualquer subpropriedade de estilo que você quiser aplicar pode ser feito através de element.style.propriedade, como abaixo (exercite escrevendo código JavaScript):

- element.style.display: retorna a visibilidade atual do elemento;
- element.style.display = "none": esconde o elemento em questão;
- element.style.display = "block": exibe o elemento em questão;

element.innerHTML

Esta propriedade define o conteúdo HTML de uma tag, útil para ler ou alterar sua aparência também:

- element.innerHTML: retorna o conteúdo HTML do componente;
- element.innerHTML = "<p>teste</p>": insere um parágrafo dentro do componente;

element.innerText

Esta propriedade define o conteúdo texto de uma tag, útil para ler ou alterar sua aparência também.

Funções de Elemento

Para usar as funções abaixo, você deve primeiro carregar o elemento (ou documento) ao qual deseja disparar a função.

Algumas funções esperam outras funções por parâmetro, o que indica que elas irão disparar a sua função em algum momento futuro, quando algo acontecer, como sendo um evento.

A primeira coisa que você tem de entender é que o exemplo acima é bem básico e que algumas páginas podem ser tão complexas que o navegador pode demorar um pouco até carregá-las. Se você quiser saber o exato momento em que o DOM de uma página está 100% carregado, o código abaixo pode lhe dizer isso.

Código 8.21: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
document.addEventListener("DOMContentLoaded", function(event)
{
    //faz alguma coisa
});
```

Neste caso, a função passada como segundo parâmetro será disparado uma única vez, assim que o DOM terminar de ser carregado.

Talvez vá demorar algum tempo até você precisar usar um código como o acima, mas tenha em mente que todas as funções de manipulação de DOM que eu mostrei nesta seção, dependem que o DOM da página já esteja carregando, ok?

element.onclick = funcao

Define uma função JavaScript que será disparado quando este element for clicado. Mesmo efeito do atributo onclick no HTML. Opcionalmente, você pode disparar este evento como qualquer outra função, para forçar o clique do elemento.

element.onchange = funcao

Define uma função que será disparada quando este element for alterado (selects principalmente). Mesmo efeito do atributo onchange.

element.onfocus e element.onblur

Define uma função para quando o usuário coloca ou tira o foco de um elemento, respectivamente.

element.onkeypress, onkeydown e onkeyup

Define funções para manipulação de teclas pressionados quando o foco está sobre o elemento em questão. Mesmo efeito dos atributos onkeypress, onkeydown e onkeyup.

element.onmouseenter, onmouseleave

Adiciona um gatilho no elemento que irá disparar uma função toda vez que o mouse entrar ou sair do elemento respectivamente (passando por cima, sem clicar).

element.onsubmit = funcão

Com esta propriedade você pode definir uma função JS que vai ser disparada toda vez que um formulário HTML for submetido, ou seja, que seu input[type=submit] for pressionado.

element.addEventListener(eventName, callback)

Esta função é a genérica para definir qualquer gatilho existente nos elementos. O primeiro parâmetro é o evento a ser mapeado e o segundo é a função que vai ser disparada quando esse evento acontecer.

Exercitando

Vamos exercitar o que vimos até aqui de manipulação de DOM criando um exercício que mais tarde evoluiremos com outra tecnologia importantíssima chamada Ajax.

Crie um arquivo HTML padrão chamado index.html, com HEAD e BODY, como abaixo:

Código 8.22: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Exemplo DOM</title>
    <meta charset="utf-8" />
```

```
</head>
<body>
  <h1>Exemplo DOM</h1>
  <p>Sistema para exemplificar manipulação de DOM</p>
</body>
</html>
```

Agora coloque duas DIVs no BODY deste HTML, uma com id 'divCadastro' e outra com id 'divListagem'. A ideia aqui é construir um único HTML que contenha tanto um formulário de cadastro quanto a listagem dos clientes cadastrados. Tornaremos essa tela dinâmica apenas usando JavaScript e nossos conhecimentos de DOM, sem qualquer tipo de backend ou banco de dados.

Código 8.23: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div id="divCadastro">
  <h2>Cadastro</h2>
</div>
<div id="divListagem">
  <h2>Listagem</h2>
</div>
```

Note que se você abrir este arquivo HTML no seu navegador verá as duas DIVs visíveis, mas não é o que queremos, certo? De alguma forma temos que garantir que apenas uma delas esteja visível de cada vez.

Para fazer isso, vamos adicionar botões de navegação em ambas DIVs, como abaixo, visando ocultar/exibir as DIVs de acordo com cada botão pressionado:

Código 8.24: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div id="divCadastro">
  <h2>Cadastro</h2>
  <input type="button" id="btnListar" value="Listar">
</div>
<div id="divListagem">
  <h2>Listagem</h2>
  <input type="button" id="btnCadastrar" value="Cadastrar">
</div>
```

Mas e como podemos programar esses botões? Com JavaScript, é claro!

Para não ficarmos escrevendo JavaScript no meio do HTML, o que é considerado uma má prática, crie um arquivo scripts.js em uma pasta js ao lado do seu index.html e referencie-o na index.html:

Código 8.25: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
< script src = "js/scripts.js" ></ script >
```

Nota: a tag SCRIPT exige que ela seja fechada com uma /SCRIPT, jamais fechada nela mesma. Se você não respeitar essa regra, sua página não irá funcionar.

Agora abra seu scripts.js e vamos definir o evento DOMContentLoaded, que é disparado assim que os componentes do documento estão todos prontos para serem manipulados:

Código 8.26: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
document.addEventListener("DOMContentLoaded", function(event)
{
```

});

O que vamos fazer aqui?

Primeiro, esconder a divListagem, pois queremos que o HTML inicie exibindo somente a divCadastro. Fazemos isso facilmente em JS selecionando o elemento que queremos esconder e depois alterando o estilo de display dele como none, que serve para esconder elementos HTML:

Código 8.27: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
document.addEventListener("DOMContentLoaded", function(event)
{
  const divListagem = document.getElementById("divListagem");
  divListagem.style.display = "none";
});
```

Implementando este código você notará que quando abrir seu arquivo index.html no navegador, ele exibirá somente uma das DIVs, mas que o botão dela não funciona ainda.

Para fazê-lo funcionar é simples, precisamos manipular o evento click deles! Ainda dentro do evento DOMContentLoaded do seu scripts.js, inclua os seguintes eventos nos botões (note que sempre começamos selecionando o componente que vamos manipular):

Código 8.28: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
document.addEventListener("DOMContentLoaded", function(event)
{
  const divListagem = document.getElementById("divListagem");
  divListagem.style.display = "none";
});
```

```
const divCadastro = document.getElementById("divCadastro");

document.getElementById("btnListar").onclick = (evt) => {
    divListagem.style.display = "block";
    divCadastro.style.display = "none";
}

document.getElementById("btnCadastrar").onclick = (evt) => {
    divListagem.style.display = "none";
    divCadastro.style.display = "block";
}
});
```

Teste agora e verá que é perfeitamente possível manipular a exibição das divs usando os botões.

Com esses conceitos e conhecimentos dominados, podemos avançar no nosso exercícios. Agora vamos criar um formulário de cadastro de cliente (nome, idade e UF) na primeira div e uma tabela de clientes na segunda. Sim, já fizemos algo parecido antes, você pode aproveitar a 'base', mas será ligeiramente diferente desta vez.

Dentro da divCadastro, segue o formulário:

Código 8.29: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<form id="frmCadastro" action="" method="">
    <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>
    <input type="button" id="btnListar" value="Listar" | <input
type="submit" value="Salvar" />
</p>
</form>
```

Tomei a liberdade de trocar o btnListar de lugar, colocando-o dentro do form também. Note que este form não possui action nem method, pois não vamos enviá-lo ao servidor, vamos fazer coisas bem mais legais. :)

Já a divListagem, segue o HTML também:

Código 8.30: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<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>Ações</td>
    </tr>
</thead>
<tbody>
    <tr>
        <td colspan="4">Nenhum cliente cadastrado.</td>
    </tr>
</tbody>
<tfoot>
```

```
<tr>
  <td colspan="4">
    <input type="button" id="btnCadastrar" value="Cadastrar">
  </td>
</tr>
</tfoot>
</table>
```

Como ainda não vimos como estilizar nossas páginas HTML, ignore os atributos style que usei e não se importe se a aparência de tudo ficar muito feia, como nas imagens abaixo:

Cadastro

Nome:

Idade:

UF: 

|

E nessa outra aqui:

Listagem

Nome	Idade	UF	Ações
------	-------	----	-------

Nenhum cliente cadastrado.

Agora vamos programar o nosso formulário usando JavaScript!

Como não temos back-end nesse exercício, quando o FORM for submetido, vamos capturar os dados enviados usando JavaScript e usá-los para preencher a tabela com a lista de clientes. Claro, esse armazenamento dos cadastros será efêmero, e toda vez que atualizarmos essa página no navegador ela virá zerada.

No entanto, servirá para nosso intuito de exercitar o que vimos de JavaScript. Como já estamos fazendo, aliás!

Para programar a submissão do form, vamos novamente abrir nosso scripts.js e inserir novos códigos dentro do evento DOMContentLoaded do document (logo abaixo daqueles scripts de clique dos botões das divs).

Código 8.31: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
const frmCadastro = document.getElementById("frmCadastro");
frmCadastro.onsubmit = (evt) => {

    let linha = '<tr>';
    var data = new FormData(frmCadastro)
    for(let item of data)
        linha += `<td>${item[1]}</td>`;
    linha += '<td><input type="button" value="X" /></td></tr>';

    //se tem apenas uma TD, é a default
    const tbody = document.querySelector('table > tbody');
    if(tbody.querySelectorAll('tr > td').length === 1)
        tbody.innerHTML = "";

    tbody.innerHTML += linha;

    divListagem.style.display = "block";
    divCadastro.style.display = "none";
```

```
frmCadastro.reset();  
  
evt.preventDefault();  
}
```

Esse código ficou extenso, mas vou explicar.

Primeiro, o evento submit do form é disparado toda vez que o form é submetido, o que no nosso caso acontece apenas quando o input de submit é pressionado. Para evitar o refresh na página, comportamento natural de submissões de formulário, coloquei um evt.preventDefault() ao final da função, para cancelar a "submissão de verdade".

Para pegar os dados que foram submetidos temos duas opções: usando formdata (que foi o que eu fiz, que retorna uma coleção de chave-valor) ou pegando campo a campo.

Usei um for na coleção formData para construir uma string html de uma linha que precisa ser inserida na tabela. Mas antes de inserir, fiz um teste para ver quantas TDs (células) existiam dentro da tr de tbody. Se existir só uma, é porque é aquela default (não existem cadastros...) e devemos removê-la apagando o innerHTML do tbody.

Para finalizar, coloquei nossa linha HTML em conjunto com o restante de HTML que já existia no tbody, para que a tabela seja atualizada e troquei a visibilidade das divs para que seja possível ver o cliente recém cadastrado. Note que em um momento usei um seletor com o formato 'element > child', onde consigo descer níveis dentro da hierarquia de elementos HTML.

Se você testar agora, verá que funciona perfeitamente.

Mas... e aquele botão de exclusão que eu coloquei na última coluna da TD e que não funciona ainda?

Adicione um último código dentro da function de submit, uma linha antes do evt.preventDefault() que fará com que todos os botões de exclusão da tabela removam a própria linha onde estão (adicionei uma confirmação JavaScript também, só pra evitar cliques acidentais):

Código 8.32: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
const buttons = document.querySelectorAll("input[value='X']");
for(let btn of buttons){
    if(btn.onclick !== null) continue;
    btn.onclick = (evt) => {
        if(confirm('Tem certeza que deseja excluir este cliente?')){
            btn.closest('tr').remove();
        }
    }
}
```

Basicamente seleciono todos os botões com "X" e aqueles que ainda não possuem uma função de click, eu adiciono a nossa de remover a tr mais próxima com as funções closest (busca um elemento determinado mais próximo) e remove (remove o elemento em questão).

Bem simples.

Listagem

Nome	Idade	UF	Ações
Luiz	29	RS	<input type="button" value="X"/>
<input type="button" value="Cadastrar"/>			

O funcionamento é muito legal na minha opinião. Uma aplicação simples e dinâmica para o usuário, muito rápida e eficiente, sem aqueles refreshs de tela, redirecionamentos, etc.

Mas como podemos manter esse mesmo nível de experiência agradável tendo um back-end com banco de dados e tudo mais? Ou carregando de uma web API da Internet?

Com Ajax.

Ajax

AJAX (acrônimo em inglês de Asynchronous JavaScript and XML , em português "JavaScript e XML Assíncrono") é o uso metodológico de tecnologias como Javascript e XML, providas por navegadores, para tornar páginas Web mais interativas com o usuário, utilizando-se de solicitações assíncronas de informações. Foi inicialmente desenvolvida pelo estudioso Jessé James Garret e mais tarde encabeçada por diversas associações. Apesar do nome, a utilização de XML não é obrigatória (JSON é frequentemente utilizado) e as solicitações também não necessitam ser assíncronas obrigatoriamente.

Quando submetemos um formulário ao servidor, seja via GET ou via POST, acabamos enviando junto muito mais informações do que gostaríamos (geralmente a página inteira é enviada), bem como direcionando a resposta do usuário para longe da página atual. Muitas vezes gostaríamos de consumir conteúdo do servidor sem ter de trocar de página e mesmo que nossa tecnologia de back-end permita redirecionar o fluxo novamente para a mesma tela, temos aquelas 'piscadas' inconvenientes no navegador que fazem com que todos os dados que estavam na página se percam.

Essa é a motivação por trás do Ajax: economia de dados trafegados e melhor experiência para o usuário.

A ideia por trás da tecnologia é que quando for necessário enviar alguma informação para o servidor, que somente a própria informação seja enviada. Que quando queremos obter alguma informação do servidor, que somente esta informação seja obtida. Em ambos os casos, que isso ocorra sem que haja um carregamento de toda a página atual, mas sim somente das partes que devem sofrer mudanças após a requisição assíncrona tiver sido concluída.

Ajax com fetch

Ajax é uma tecnologia embutida na linguagem de scripting Javascript, através do objeto XMLHttpRequest (XHR), que permite ao navegador fazer chamadas assíncronas a recursos em outro endereço do servidor.

Atualmente, com JavaScript moderno (suportado pelos navegadores mais modernos também), é extremamente simples de fazer requisições AJAX usando a função `fetch`.

```
fetch(url[, opcoes]);
```

Esta função pode ser usada apenas passando a URL que se deseja requisitar de maneira assíncrona. Nesse caso, será feito um HTTP GET na URL e o fluxo de execução seguirá normalmente. Quando a requisição retornar com uma resposta (o que pode ser praticamente instantâneo ou demorar minutos), devemos executar a função de retorno usando um conceito chamado Promises do JavaScript.

Usaremos neste primeiro exemplo uma web API aberta e gratuita existente na Internet chamada IPStack. Ela permite que você informe um endereço IP e ela vai lhe responder com a geolocalização daquele IP.

Acesse <https://ipstack.com> e crie uma conta free/gratuita para obter uma API Key, que é uma forma de segurança bem popular na Internet. É bem rápido.

Vamos criar uma página HTML simples, para que um usuário possa inserir um endereço IP em um input e, pressionando um botão, descobrir de onde ele é.

Código 8.33: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<label>IP:  
<input type="text" id="txtIP" value="8.8.8.8" />
```

```
</label>
<p>
  <input type="button" value="Buscar" onclick="buscarIP()" />
</p>
<p id="paragrafo"></p>
```

Nenhuma novidade aqui, mas note que o botão está esperando uma função buscarIP que ainda não criamos. Assim, crie um bloco script no final desta página HTML com o código do nosso fetch e incluindo a sua API Key como abaixo.

Código 8.34: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<script>
  function buscarIP() {
    const apiKey = "coloque a sua API Key aqui";
    const ip = document.getElementById('txtIP').value;//ex: 8.8.8.8
    const paragrafo = document.getElementById('paragrafo');
    fetch(`http://api.ipstack.com/${ip}?access_key=${apiKey}&format=1`)
      .then(req => req.json())
      .then(json => paragrafo.innerText = json.country_name)
      .catch(err => paragrafo.innerText = err);
  }
</script>
```

Neste bloco, além das primeiras linhas que não devem ser novidade para você, está o fetch. O primeiro parâmetro dele é a string com a URL de acesso à web API. Aqui, eu monto ela usando um conceito chamado Template Literals que permite juntar variáveis com texto de forma mais organizada do que usando +.

A novidade aqui fica por conta do then e do catch.

Como eu mencionei antes, o fetch é assíncrono, ou seja, ele vai buscar a sua informação e libera sua página para fazer outra coisa.

Mas como saber quando ele voltou com a resposta que eu queria?

Através do `then`. O `then` só é executado quando a resposta retorna com sucesso. Se der erro, vai ser executado o `catch`, logo abaixo.

Mas por que eu coloquei dois `thens`?

O `fetch` é uma função bem "crua", serve para fazer requisições de qualquer coisa. Como essa API retorna dados no formato JSON (experimente acessar aquela URL do `fetch` no navegador, para ver ela por completo), precisamos do primeiro `then` para converter a resposta para JSON (por padrão ela vem em bytes). Essa conversão também é assíncrona e cairá a resposta para o segundo `then`, que aí sim exibe a informação que queremos na tela (o nome do país).

O resultado você confere na imagem abaixo.

IP: **8.8.8.8**

Buscar

United States

Opções do Fetch

Como eu mencionei antes, o `fetch` tem um segundo parâmetro opcional que são as opções ou configurações da requisição. Isso porque nem sempre você vai querer um HTTP GET e muitas vezes você vai precisar enviar dados no corpo da requisição. Isso tudo são opções do `fetch`, sendo que abaixo listo algumas delas que devem ser enviadas em um único objeto JSON como segundo parâmetro do `fetch`.

- method: string com GET, POST, etc
- headers: um objeto do tipo Headers;
- mode: usado para cors, por exemplo (conceito avançado);
- body: o conteúdo da sua requisição (deve coincidir com o header);

Um exemplo de requisição hipotética, com opções, pode ser visto abaixo.

```
fetch('URL', {  
  headers: myHeaders,  
  method: 'POST',  
  mode: 'cors',  
  body: myData  
})
```

Note duas coisas aqui: primeiro, em headers eu passei um objeto. Isso porque a propriedade headers do fetch espera um objeto do tipo Headers, como este que criei abaixo.

```
const myHeaders = new Headers();  
myHeaders.append("Content-Type", "application/json");
```

E segundo, que no body eu passei um objeto também, que na verdade deve ser uma string. Se o que eu quiser enviar for um objeto JavaScript, eu posso serializá-lo usando JSON.stringify, como abaixo.

```
const myData = JSON.stringify({numero: 1})
```

Infelizmente não conheço nenhuma API pública na Internet que permita POST para testarmos essas possibilidades de uso do fetch. Mas vamos fazer mais um exercício abaixo, usando uma API nossa mesmo, escrita em Node.js do capítulo 7.

Ajax na prática com JavaScript e Node.js

Note que todas as funções Ajax só fazem sentido quando temos um backend pronto para atendê-las.

Vimos em capítulos anteriores como fazer aplicações web que possuem formulários. Esses formulários quando são submetidos enviam a página inteira HTML para o servidor e exigem a renderização completa do HTML novamente, caracterizado por uma 'piscada' na tela do navegador. Entretanto, nem sempre queremos submeter todos os dados de um formulário ao servidor e muitas vezes nem mesmo queremos enviar a página em si ao servidor, apenas obter dados de lá, por exemplo.

Nestes casos, é útil termos uma Web API no backend que responderá à chamadas Ajax, principalmente aquelas usando o verbo GET, para poder carregar dinamicamente trechos da tela como opções de um select ou divs dinâmicas. Web APIs como essas que estou mencionando já foram ensinadas em capítulo anterior e, se você acompanhou os exercícios práticos, possui uma pronta agora que manipula dados de clientes no banco MongoDB.

Atenção: parto do pressuposto que você possui um banco de dados MongoDB rodando na sua máquina neste exato momento, tal qual foi ensinado no capítulo de Mongo. Também considero que sua WebAPI em Node.js está pronta e rodando em localhost:3000. E por fim, que você fez o exercício prático da seção de manipulação do DOM com JavaScript Client-Side, e possui a index.html com o formulário de cadastro e listagem pronto.

Vou chamar este projeto de exemplojavascript, uma continuação do projeto da seção de manipulação do DOM, onde temos apenas uma página index.html que serve para cadastrar e listar clientes,

alternando entre divs e com todo comportamento de front-end pronto. O que vamos fazer agora é conectá-la ao nosso back-end Node.js através de requisições HTTP Ajax.

Como mandam as boas práticas, nosso index.html não possui código JavaScript algum, sendo apenas uma view estática. Em nossa pasta js guardamos os arquivos JS que tornam essa página dinâmica, basicamente um arquivo scripts.js com nossos scripts personalizados. Vamos abrir este último.

No scripts.js temos um evento DOMContentLoaded do document (nossa página HTML) que define alguns comportamentos e eventos aos componentes do documento usando seletores DOM.

Um desses eventos é o submit do frmCadastro, que hoje apenas adiciona uma nova linha na tabela de listagem. Além desse comportamento, queremos que ele poste os dados do formulário para nossa web API, para que o cliente seja salvo no banco também. Antes de aumentarmos a quantidade de código neste submit, vamos organizá-lo melhor.

Crie uma função em scripts.js, fora do listener de DOMContentLoaded, para encapsular toda a lógica de adicionar a linha na tabela de listagem, como abaixo:

Código 8.35: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function updateTable(data){
    let linha = '<tr>';
    for(let item of data)
        linha += `<td>${item[1]}</td>`;
    linha += '<td><input type="button" value="X" /></td></tr>';

    //se tem apenas uma TD, é a default
    const tbody = document.querySelector('table > tbody');
    if(tbody.querySelectorAll('tr > td').length === 1)
        tbody.innerHTML = "";
```

```
tbody.innerHTML += linha;

divListagem.style.display = "block";
divCadastro.style.display = "none";

frmCadastro.reset();

const buttons = document.querySelectorAll("input[value='X']");
for(let btn of buttons){
    if(btn.onclick !== null) continue;
    btn.onclick = (evt) => {
        if(confirm('Tem certeza que deseja excluir este cliente?')){
            btn.closest('tr').remove();
        }
    }
}
```

Enquanto que nosso evento submit vai ficar assim:

Código 8.36: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const frmCadastro = document.getElementById("frmCadastro");
frmCadastro.onsubmit = (evt) => {

    var data = new FormData(frmCadastro);
    updateTable(data);

    evt.preventDefault();
}
```

Agora, vamos criar outra função, que postará via Ajax os dados do formulário, em formato JSON, para a nossa webapi Node.js:

Código 8.37: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const webApiDomain = 'http://localhost:3000'
async function updateDatabase(data){
  const json = {};
  for(let item of data)
    json[item[0]] = item[1];

  const headers = new Headers();
  headers.append("Content-Type", "application/json");
  const response = await fetch(` ${webApiDomain}/clientes`, {
    headers,
    method: 'POST',
    body: JSON.stringify(json)
  });
  return await response.json();
}
```

Nota: repare como coloquei o domínio da web API em uma constante separada. Isso porque essa informação será usada mais de uma vez e, se um dia você publicar isso em um webhosting, você deverá mudar localhost:3000 para o domínio final da sua API, o que fica muito mais fácil se essa informação estiver em um único lugar.

Essa função pega o array de campos do formulário e os transforma em um objeto JSON, pois é isso que nossa API espera. Com esse objeto JSON pronto, usamos o fetch configurado para POST na URL passada por parâmetro, com o JSON no body e a usando await para que seja mais fácil de gerenciar o fluxo de processamento (isso requer que a function seja async).

Bem simples a essa altura do campeonato.

E vamos chamar essa função dentro do evento de submit do form também, ficando assim:

Código 8.38: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const frmCadastro = document.getElementById("frmCadastro");
frmCadastro.onsubmit = (evt) => {

  var data = new FormData(frmCadastro);
  updateDatabase(data)
    .then(result => {
      const cliente = result.ops[0];
      alert(`Cliente ${cliente.nome} cadastrado com sucesso!`);
      updateTable(cliente);
    })
    .catch(error => alert(`Ocorreu um erro: ${error}`));

  evt.preventDefault();
}
```

Aqui, eu pego o retorno do updateDatabase, que é uma Promise JavaScript (falamos dela quando expliquei o fetch, lembra?) e passo uma arrow function de callback para ela, recebendo o cliente que acabou de ser inserido no banco de dados (ele vem meio escondido no result, mas vem).

Esse cliente vai ser usado pela updateTable para atualização da tela, mas ele não está esperando um cliente hoje, então vamos lá na nossa função e updateTable para dar um update em apenas duas linhas dela.

Código 8.39: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function updateTable(cliente){
  let linha = `<tr><td>${cliente.nome}</td><td>${cliente.idade}</td>
<td>${cliente.uf}</td><td><input type="button" value="X" /></td>
</tr>`;
```

Simplifiquei a construção da nossa linha da tabela com os dados obtidos no cliente retorna pela WebAPI (ela retorna o último

cadastrado).

Isso teoricamente seria o suficiente para que o cadastro passasse a funcionar, enviando os dados ao back-end. No entanto, a vida não é tão fácil assim. Se você abrir a index.html no navegador e testar o formulário, aparentemente nada acontecerá, mas se usar o F12 do Chrome (Developer Tools), você deve encontrar na aba console um erro como abaixo:

XMLHttpRequest cannot load http://localhost:3000/clientes. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access.

Se a sua página HTML e sua web api estiverem rodando no mesmo projeto, você não terá esse problema. Mas aqui como estamos rodando a index.html em um projeto separado da web API, existe uma regra de segurança dos browsers que proíbe chamadas Ajax de um domínio serem enviadas para outro, a menos que o outro domínio permita isso. Essa regra se chama de Same Origin Policy (Política da Mesma Origem) e a melhor maneira de driblarmos ela é usando CORS.

CORS

CORS ou Cross-Origin Resource Sharing (Compartilhamento de Recursos de Origem Cruzada) é uma especificação técnica que permite a um domínio responder a requisições de outro domínio. Podemos especificar qual(is) domínio(s) é(são) esse(s) ou simplesmente liberar o acesso geral, o que é ok se essa API for pública.

Para usar CORS, teremos de voltar no nosso projeto webapi e instalar uma nova dependência, do módulo cors:

Código 8.40: disponível em <https://www.luiztools.com.br/livro-node-fontes>

npm install cors

E no seu app.js, o coração da webapi, adicione apenas esta linha depois do primeiro bloco que carrega os módulos e começam os 'app.use'. Ela carrega e habilita o CORS para todos domínios, inserindo-o no middleware do Express:

Código 8.41: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
app.use(require('cors')())
```

Agora sim sua web API está permitindo requisições Ajax de domínios diferentes e, se abrir a index.html no browser novamente e tentar realizar um cadastro, verá que agora ele funciona plenamente!

Atenção: na documentação do módulo cors você encontra maneiras de torná-lo mais seguro do que essa implementação que usei. Certifique-se de ler a documentação caso vá usar essa API em produção (é bem simples): <https://github.com/expressjs/cors>

Porém, mesmo com nosso cadastro funcionando (você pode se certificar que realmente salvou os dados acessando o MongoDB via linha de comando), ainda temos o problema da listagem, pois toda vez que carregamos nossa index.html pela primeira vez, a tabela vem vazia, independente do banco de dados.

Para arrumar isso, vamos criar uma nova função no scripts.js que traz os clientes do banco de dados pra gente:

Código 8.42: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
async function loadDatabase(){
  const response = await fetch(`{$webApiDomain}/clientes`);
  return await response.json();
}
```

Esta função vai retornar pra gente, de forma assíncrona, um array de clientes, certo?

Só que hoje não temos um jeito de colocar vários clientes na tela, pois a nossa função updateTable trabalha com apenas um cliente de cada vez. Sendo assim, vamos voltar na updateTable para atualizar ela mais um pouco (somente as primeiras linhas dela):

Código 8.43: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function updateTable(clientes){  
    let linha = "";  
    if(!Array.isArray(clientes)) clientes = [clientes];  
    for(let cliente of clientes)  
        linha += `<tr><td>${cliente.nome}</td><td>${cliente.idade}</td>  
        <td>${cliente.uf}</td><td><input type="button" value="X" /></td>  
        </tr>`;  
  
    //se tem apenas uma TD, é a default
```

Agora, a nossa updateTable pode receber um ou vários clientes que ela vai se comportar como deveria, montando o HTML, colocando na TABLE, registrando os eventos dos botões e etc.

Mas, ainda falta juntarmos as duas pontas, a loadDatabase com a updateTable, e vamos fazer isso no DOMContentLoaded, para que aconteça somente uma vez quando a página for carregada, certo?

Código 8.44: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
document.addEventListener("DOMContentLoaded", function(event){  
    const divListagem = document.getElementById("divListagem");  
    const divCadastro = document.getElementById("divCadastro");  
    divCadastro.style.display = "none";
```

```
loadDatabase()
  .then(clientes => updateTable(clientes))
  .catch(error => alert(`Ocorreu um erro ao carregar os clientes:
${error}`));
```

Agora abra novamente o seu index.html e verá o resultado do código acima onde a div inicialmente exibida será a de listagem e ela já virá carregada com todos os clientes previamente cadastrados na base!!!

Listagem

Nome	Idade	UF	Ações
Luiz	29	RS	<input type="button" value="x"/>
Teste	28	RS	<input type="button" value="x"/>
jquery	15	SC	<input type="button" value="x"/>
ajax	12	SC	<input type="button" value="x"/>

Mas nosso trabalho ainda não acabou, afinal, aquele nosso botão de excluir ali da imagem ainda não funciona 'de verdade'. Hoje eles apenas removem a linha da tabela, mas não o cliente do banco.

Para fazer ele funcionar é um pouco mais complicado pois precisamos ir no banco para remover o cliente daquela linha. O jeito certo de fazer isso é tendo uma informação única, que não se repita entre os clientes, como o `_id` do MongoDB. No entanto, hoje não temos essa informação à nossa disposição no HTML.

Na função `updateTable()` isso é fácil de resolver, vamos modificar levemente ela, colocando essa informação em um data attribute no input de exclusão. Os data attributes são atributos personalizados iniciados com o prefixo 'data-' que a W3C (órgão que mantém os

padrões do HTML) recomenda quando precisar de dados personalizados em uma tag HTML.

Assim, queremos algo como

```
<input type="button" value="X" data-  
id="59ab46e433959e2724be2cbc" />
```

E para fazer isso no updateTable, é bem simples (a única linha que mudou foi a que monta o HTML da linha):

Código 8.45: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
linha += `<tr><td>${cliente.nome}</td><td>${cliente.idade}</td>  
<td>${cliente.uf}</td><td><input type="button" value="X" data-  
id="${cliente._id}" /></td></tr>`;
```

Agora quando a tabela for carregada à partir do banco de dados, ela terá o data-id definido no botão de excluir.

Para fazer com que a exclusão funcione de verdade, vamos primeiro criar uma função nova no scripts.js, para fazer a requisição Ajax de DELETE. Será bem parecido com o POST que fizemos antes, para cadastrar clientes:

Código 8.46: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
async function deleteDatabase(id){  
  const response = await fetch(`.${webApiDomain}/clientes/${id}`, {  
    method: 'DELETE'  
  });  
  return await response.json();  
}
```

Esta chamada é até mais simples que a POST que fizemos antes pois o DELETE não precisa de headers especiais ou de body, pois a informação necessária é apenas o id a ser excluído, que vai na própria URL da requisição.

E no evento de click dos nossos botões de excluir, modificamos da seguinte forma:

Código 8.47: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const buttons = document.querySelectorAll("input[value='X']");
for(let btn of buttons){
  if(btn.onclick !== null) continue;
  btn.onclick = (evt) => {
    if(confirm('Tem certeza que deseja excluir este cliente?')){
      deleteDatabase(btn.getAttribute('data-id'))
        .then(result => {
          alert('Cliente excluído com sucesso!');
          btn.closest('tr').remove();
        })
        .catch(error => alert(`Ocorreu um erro ao excluir o cliente:
${error}`))
    }
  }
}
```

Aqui eu chamo a função deleteDatabase passando o id obtido com o btn.getAttribute, buscando pelo data-id que é o atributo personalizado que criei, lembra?

O código que remove a linha no HTML foi colocado como callback no then, para garantir que só será feito após a exclusão ter sido realizada no banco de dados.

Isso é o suficiente para que a exclusão de clientes funcione, encerrando aqui nossa prática de Ajax com JavaScript client-side,

usando Node.js como back-end.

Em nosso próximo capítulo, finalmente vamos estilizar nossas páginas, usando essa mesma index.html como base.

Nota: nos capítulos deste livro lhe mostrei duas abordagens completamente diferentes de criação de aplicações web. Uma usando Node.js como backend e EJS para construção das views. Outra usando Node.js como backend e JavaScript Ajax para construção das views.

Existem abordagens ainda mais profissionais e complexas, usando frameworks de front-end responsivo, como ReactJS, Angular e VUE.js, que dariam um livro inteiro para cada uma.

Por enquanto, o mais comum são abordagens mistas, onde a página base é renderizada no servidor usando Node.js + EJS e os comportamentos dinâmicos das telas são feitos usando JQuery Ajax, mas não há uma regra. Cabe à você entender as necessidades do seu projeto e escolher a abordagem mais adequada.

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Por que Stanford trocou Java por JavaScript?

Entenda a transformação que JavaScript está causando e como podemos usá-lo para se preparar às mudanças que estão acontecendo.

<https://www.luiztools.com.br/post/por-que-stanford-trocou-java-por-javascript/>

JSON Web Token

Aprenda a adicionar segurança nas suas Web APIs Node.js com JWT.

<https://www.luiztools.com.br/post/autenticacao-json-web-token-jwt-em-nodejs/>

Agilidade e Microservices

Artigo que escrevi para a revista do iMasters sobre a revolução que o uso de microservices (Web APIs independentes) e o uso de métodos ágeis está mudando a forma como as empresas trabalham ao redor do mundo.

<https://www.luiztools.com.br/post/microservices-e-agile-o-futuro-da-programacao/>

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

9 Front-end: CSS

Everyone knows that debugging is twice as hard as writing a program in the first place.

So if you're as clever as you can be when you write it, how will you ever debug it?

- Brian W. Kernighan

Outra tecnologia muito comum quando o assunto é front-end é o CSS. CSS significa Cascading Style Sheets, ou Folhas de Estilo em Cascata. Enquanto que HTML é o esqueleto da página, o CSS são os órgãos e principalmente, a pele, o que dá a aparência, o estilo.

Estilos podem ser adicionados de três maneiras nos elementos HTML:

- Inline: usando o atributo style
- Internal: usando a tag STYLE
- External: usando arquivos CSS externos

A forma mais comum de ser utilizado CSS é a terceira, principalmente por ser a mais organizada, em analogia ao que fazemos com JavaScript também. Ainda assim, veremos as três formas neste capítulo.

CSS Inline

Todo elemento HTML possui um estilo padrão. A cor dos textos é preta, e a cor do plano de fundo é branca, só para citar dois exemplos. Alterar o estilo de um elemento HTML pode ser feito usando o atributo style, presente em todos elementos.

O exemplo a seguir mostra como alterar a cor do plano de fundo de branco (default) para cinza claro.

Código 9.1: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<body style="background-color: lightgrey">
  <h1>This is a heading</h1>
  <p>This is a paragraph.</p>
</body>
```

Nota: sugiro abrir uma página HTML e testar todos os exemplos de estilos CSS vendo como eles se comportam no navegador. Pode inclusive usar a index.html que criamos no capítulo anterior.

O atributo style possui a seguinte sintaxe:

```
style="property:value"
```

Sendo property uma propriedade CSS e value um valor CSS. Entenderemos melhor do que se trata CSS mais tarde neste mesmo material. Por enquanto, vamos ver como podemos mudar o estilo de nossos textos.

Estilos de texto

Para alterar a cor do texto usamos a propriedade color.

Código 9.2: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<h1 style="color:blue">This is a heading</h1>
<p style="color:red">This is a paragraph.</p>
```

Nota: todas os valores de cores CSS aceitam literais em Inglês (blue, red, black, etc) e valores hexadecimais iniciados com # (como #CCCCCC para cinza claro e #FFFFFF para branco, por exemplo).

Para alterar a tipografia, usamos a propriedade font-family.

Código 9.3: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<h1 style="font-family:verdana">This is a heading</h1>
<p style="font-family:courier">This is a paragraph.</p>
```

Nota: todas as fontes são aceitas pelo CSS, no entanto, uma vez que os estilos são interpretados client-side, você não deve utilizar fonts que não existam por padrão nos navegadores e/ou sistemas operacionais de uso comum. Caso a font que você definiu não exista na máquina do usuário, o browser irá renderizar como um fonte comum, provavelmente estragando seu layout.

O tamanho do texto pode ser definido pela propriedade font-size.

Código 9.4: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<h1 style="font-size:300%">This is a heading</h1>
<p style="font-size:160%">This is a paragraph.</p>
```

Nota: o tamanho pode ser definido em porcentagem, pontos e pixels, conforme a sua preferência.

Enquanto que o alinhamento é definido pela propriedade text-align.

Código 9.5: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<h1 style="text-align:center">Centered heading</h1>
<p>This is a paragraph.</p>
```

Classes de Estilos em CSS Internal

Muitas vezes temos o mesmo estilo aplicado a diversos elementos em nossas páginas HTML. Da mesma forma, às vezes queremos que um determinado estilo se aplique a todas ocorrências de um elemento HTML no documento. Em todas essas ocasiões em que teríamos atributos style com propriedades e valores repetidos devemos utilizar classes de estilo, mais comumente chamadas classes CSS.

Existe uma tag que ainda não utilizamos em nossas lições que é a tag STYLE. A tag STYLE define um bloco no seu documento HTML onde você poderá definir os estilos-base da sua página bem como as suas classes de estilo. A sua utilização é bem semelhante à da tag SCRIPT, ou seja, a tag STYLE é um contâiner que conterá dentro dela estilos aplicados aos elementos desta página HTML em específico.

Código 9.6: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<style>
/* seu estilo vai aqui. A propósito, isto é um comentário CSS */
</style>
```

A tag STYLE pode ir em qualquer ponto do HTML, mas preferivelmente dentro da tag HEAD da página, para que seja carregado antes dos elementos HTML, para que quando eles sejam renderizados, já o sejam na maneira correta (com o novo estilo ao invés do padrão).

Dentro da tag STYLE podemos definir um estilo que se aplica a todos elementos de uma tag HTML específica, como abaixo, onde definimos que a cor dos textos dentro de todos parágrafos será verde (usei o nome da cor, mas você também pode usar valores hexadecimais).

Código 9.7: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<style>
p {
    color: green;
}
</style>
```

A sintaxe é bem simples: definimos a tag à qual vamos aplicar o estilo e abrimos chaves para colocar o estilo dentro. A sintaxe para definição do estilo é a mesma que vimos no atributo style anteriormente, com propriedades e valores CSS.

```
element { property:value; property:value; }
```

Note que esse estilo será aplicado automaticamente sobre todas tags P (os parágrafos), sem exceção. Muitas vezes não queremos isso, queremos que um determinado estilo seja aplicado a todos os parágrafos que iniciam um novo capítulo, por exemplo. Neste caso, devemos criar uma classe de estilo, como abaixo.

Código 9.8: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<style>
.paragrafolnicial {
    color: green;
}
</style>
```

Note que as classes começam com um ponto, ao invés de simplesmente conterem o nome de uma tag. Além disso, sua nomenclatura é a mesma de variáveis Java e JavaScript (Camel Case) sem espaços, acentos ou caracteres especiais, raramente usando _ (underline) ou – (hífen). O estilo da classe .paragrafolnicial

será aplicado em todos os parágrafos que nós definirmos explicitamente através do uso do atributo CLASS, presente em todos elementos HTML, da seguinte maneira.

Código 9.9: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<p class="paragrafolnicial">  
    Este texto ficará verde.  
</p>  
<p>  
    Este texto terá a cor normal (preta).  
</p>
```

Note que o atributo class dispensa o ponto do início da classe, colocamos apenas o nome dela em todos os elementos HTML que quisermos aplicar seu estilo.

Vale ressaltar também que os estilos funcionam em cascata, ou seja, podemos definir um estilo global para todos parágrafos, mais um estilo específico apenas para os parágrafos iniciais. Por exemplo:

Código 9.10: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<style>  
    p { font-family: Arial; }  
  
    .paragrafolnicial { Color: green; }  
</style>
```

Assim, todos os parágrafos da aplicação terão a fonte Arial, mas somente os parágrafos com o atributo class definido como paragrafolnicial é que terão a cor da fonte verde.

Dicas e Truques

Existem diversos truques que podemos utilizar para aumentar ainda mais o poder dos estilos. Podemos definir estilos específicos para tags específicas dentro da hierarquia HTML de uma página, como abaixo, onde definimos que somente a cor dos textos dos parágrafos dentro de tabelas é que será verde, os demais manterão a cor normal (preta).

Código 9.11: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<style>
  table p {
    color: green;
  }
</style>
```

Também podemos definir que um elemento possua mais de uma classe de estilo, apenas separando o nome das nossas classes por espaços:

Código 9.12: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<p class="paragrafolInicial paragrafolImportante">Parágrafo cheio de
estilo.</p>
```

Assim o parágrafo receberá ambos estilos.

E por fim, quando queremos que somente um elemento receba determinado estilo, mas ao mesmo tempo não queremos usar o atributo style nele (pois é um estilo complexo, por exemplo), podemos criar um estilo baseado no atributo id do elemento, como abaixo.

Código 9.13: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<style>
#paragrafoLegal{
    color: red;
}
</style>
```

Assim, o elemento HTML cujo atributo id for ‘paragrafoLegal’ receberá automaticamente o estilo determinado na tag STYLE acima.

Nota: JavaScript usa essa mesma notação em seus seletores, usando o nome literal das tags, classes CSS iniciadas em '.' e ids de componentes iniciados com '#'.

Arquivos CSS Externos

O uso de arquivos CSS externos (que nada mais são do que arquivos de texto com a extensão .CSS), geralmente em uma pasta chamada ‘styles’ ou ‘css’, é preferível uma vez que provavelmente você irá querer aproveitar os estilos entre mais de uma página HTML da mesma aplicação web. Com o uso correto de arquivos CSS você pode alterar a aparência inteira de um website ou sistema web apenas mexendo em um arquivo, da mesma forma que com arquivos JS alteramos o comportamento sem mexer no HTML manualmente.

Para definir um arquivo CSS à um documento HTML devemos referenciá-lo usando a tag LINK dentro do HEAD da página, como abaixo.

Código 9.14: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<!DOCTYPE html>
<html>
<head><link rel="stylesheet" href="estilos.css" /></head>
<body>
```

Neste exemplo estamos referenciando um arquivo estilos.css que está na mesma pasta do arquivo HTML, caso contrário nosso href seria maior, indicando o caminho até o mesmo.

Imagine cada arquivo CSS como um bloco STYLE. Escreva quantos estilos quiser e sinta-se livre para usar identação, quebras de linha e comentários para tornar seu CSS mais legível.

Para conseguirmos construir estilos que façam sentido para nossos componentes, não basta apenas conhecermos o nome das propriedades. Temos de entender o conceito de 'Styling Boxes' do CSS.

Estilos de Caixas

Imagine que todo elemento HTML está dentro de uma caixa. Existem diversos estilos que podemos aplicar à essa caixa imaginária e vamos ver alguns logo abaixo.

Exibição

Quando queremos alterar a exibição da nossa caixa usamos a propriedade display, com os valores none ou block, seja para esconder ou exibir, respectivamente.

Código 9.15: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
p {  
    display: none; /* todos p's ficarão invisíveis*/  
}
```

Nota: esse é exatamente o comportamento da função hide() do JQuery (e inversamente o efeito da show())

Plano de Fundo

Também podemos alterar o plano de fundo das nossas caixas, usando as propriedades background-color e background-image:

Código 9.16: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
p {  
    background-color: #d0e4fe;  
    background-image: url('url da imagem');  
    background-repeat: no-repeat;  
}
```

No exemplo acima definimos a cor do plano de fundo de todos nossos parágrafos para um valor hexadecimal com a propriedade background-color. Também definimos que todos nossos parágrafos terão como imagem de fundo a imagem cuja url deverá estar entre as aspas dentro dos parênteses, com as mesmas regras do atributo src da tag IMG, isso tudo com a propriedade background-image. E por fim, apenas dissemos que a imagem de fundo não se repetirá, caso a caixa seja maior que a imagem em si.

Nota: por padrão, caso a caixa seja maior que a imagem, a imagem se repetirá tanto na horizontal quanto na vertical, o que muitas vezes não é um comportamento interessante pois fica feio.

Largura e Altura

Caso queiramos definir uma altura e largura fixa para nossa caixa, usamos as propriedades width e height, que permitem definir porcentagens ou tamanho fixo em pixels.

Código 9.17: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
p {  
    width: 100%;  
    height: 50px;  
}
```

Caso queiramos definir largura ou altura máxima ou mínima, neste caso utilizamos as variantes das propriedades width e height: max-width/max-height e min-width/min-height.

Borda, espaçamento e margem

Para definir uma borda ao redor do nosso elemento HTML usamos a propriedade CSS border, onde definimos a espessura, o tipo de borda e sua cor, sendo os três valores separados por espaço.

Código 9.18: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
p { border: 1px solid black; }
```

Aqui, todos os parágrafos terão uma borda de 1px preta e sólida ao seu redor.

Ainda pensando em nossos parágrafos-caixas (embora o conceito de box se aplique a qualquer elemento), que tal colocarmos um espaçamento do texto do parágrafo (seu conteúdo) em relação às suas bordas? Fazemos isso com padding (margem interna).

Código 9.19: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
p { border: 1px solid black; padding: 10px; }
```

Agora todos os parágrafos manterão seus textos 10px distantes das quatro bordas da caixa imaginária (top, bottom, left e right).

No exemplo abaixo, além do espaçamento interno, definiremos uma margem (externa) de 30px de cada um dos nossos parágrafos em relação à qualquer elemento à sua volta.

Código 9.20: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
p {  
    border: 1px solid black;  
    padding: 10px;  
    margin: 30px;  
}
```

Nota: todos esses estilos de caixa aplicam-se às 4 bordas da caixa: top (topo), bottom (rodapé), left (esquerda) e right (direita). Assim, caso queira aplicar um estilo de caixa à somente uma borda específica, use a propriedade correspondente seguida de um traço e o nome da borda, como abaixo, onde definimo que somente a margin do rodapé será de 15px.

Código 9.21: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
p {  
    margin-bottom: 15px;  
}
```

Para uma referência completa de todas propriedades CSS e como utilizá-las, acesse: <http://www.w3schools.com/cssref/default.asp>

CSS e JavaScript

Podemos definir estilos em nossos elementos HTML utilizando JavaScript também, de maneira dinâmica. Já falei disso em capítulo anterior, onde mostrei como fazer usando a propriedade `element.style.propriedade`. Para reforçar, aqui pegamos um parágrafo com id ‘myP’ e trocamos a cor do seu texto dinamicamente.

Código 9.22: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
document.getElementById('#myP').style.color = 'red';
```

A propriedade `style` está presente em todos elementos HTML e possui como sub-propriedades as mesmas características disponíveis no CSS, com a única atenção é que estilos CSS com hífen mudam de nomenclatura, pois o padrão do JS é camel case.

- `background-color` no CSS vira `backgroundColor` no JS;
- `border-left` no CSS vira `borderLeft` no JS;

E assim por diante.

Além disso, podemos adicionar e remover classes CSS a um elemento usando a propriedade `classList` do `element`, como abaixo.

Código 9.22: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
document.getElementById('#myP').classList.add('nomeClasse');
```

Outra questão importante é que quando vimos o capítulo de JavaScript client-side aprendemos a utilizar seus seletores. Pois bem, um poderoso seletor que está disponível no JS é o seletor por classe, como abaixo.

Código 9.23: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
document.getElementsByClassName('.paragrafolnicial');
```

Aqui selecionamos todos os elementos HTML que possuam a classe paragrafolnicial definida em seu atributo class. Simples assim. Essa é uma maneira extremamente útil de, com uso de laços de repetição, executar operações sobre grupo de elementos, como esconder todos...

Código 9.24: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
const col = document.getElementsByClassName("teste");
for(let item of col)
  item.style.display = "none";
```

Note que com o exemplo acima já é possível abrir a sua cabeça para as possibilidades de manipulação de elementos em lote.

Repare também como usei o for com uma construção ligeiramente diferente. Isso porque o retorno do getElements (ByClassName, ByTagName, etc) é um Iterator e não um array comum. Assim sendo, aquela construção de for ali em cima, que chamaos de for-each percorrerá todos os elementos do Iterator independente da quantidade, ordem, etc.

Exercitando

Vamos fazer um exercício bem rápido e simples com CSS antes de entrarmos em um tópico mais avançado de estilização de páginas, apenas para ver se você conseguiu entender como CSS funciona. Vamos usar aqui o mesmo projeto final do capítulo anterior, aquele com formulário de cadastro e tabela e listagem.

Primeiro, vamos entender onde queremos chegar:

Cadastro

Nome:

Idade:

UF:

|

Nenhuma obra-prima, mas bem mais agradável que o layout anterior na minha opinião.

E a tabela de clientes:

Listagem

Nome	Idade	UF	Ações
Luiz	29	RS	<input type="button" value="x"/>
Teste	28	RS	<input type="button" value="x"/>
jquery	15	SC	<input type="button" value="x"/>
ajax	12	SC	<input type="button" value="x"/>
mais um	80	RS	<input type="button" value="x"/>
outro cliente	30	RS	<input type="button" value="x"/>

Nota: não investirei pesado em CSS aqui pois este não é exatamente um livro de webdesign, algo que requer habilidades além da minha capacidade enquanto programador. Geralmente os times de desenvolvimento possuem um designer que é responsável pelo visual geral da aplicação web, cabendo aos programadores, tanto front-end quanto back-end, tornarem funcional.

Além disso, na próxima seção você verá muitos truques que vão tornar obsoletos os estilos que criaremos aqui...Ops, spoiler...

Você acha que consegue sozinho? Tente!

Comece criando uma pasta css na raiz do seu projeto e dentro dela um arquivo estilos.css vazio, referenciando-o no HEAD do seu index.html:

Código 9.27: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

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

Agora dê uma olhada no HTML. Notará que algumas vezes usei o atributo STYLE em algumas tags. Embora isso funcione perfeitamente, o ideal é não utilizar estilo inline, ou seja, estilos ao longo do HTML. O mais elegante e correto é colocar todos os estilos em arquivos CSS, referenciados no HTML, assim como estamos fazendo com nossos JavaScripts.

Nota: a regra geral diz que HTML é a forma, CSS o estilo e JS o comportamento e que cada um desses três elementos que formam a base do front-end moderno deve estar em seus respectivos arquivos.

Primeiro, note o uso do atributo STYLE da nossa table de listagem. Ele está indicando que a table deve ocupar no máximo metade da largura do contâiner mais externo à ela (a DIV).

Código 9.28: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<table style="width:50%">
```

Remova esse atributo style e vamos colocar esse estilo de tabela em nosso arquivo estilos.css:

Código 9.29: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
table { width: 50%; }
```

Agora, repare no style da TR que fica no THEAD da table. Ele diz que a cor de fundo desta TR deve ser cinza claro (#CCC é um tom de cinza claro).

Código 9.30: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<tr style="background-color: #CCC">
```

Remova este estilo e vamos colocá-lo via arquivo estilos.css:

Código 9.31: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

table thead tr { background-color: #CCC; }

Note que aqui eu tive de dizer toda a hierarquia até chegar no componente que desejo alterar. Experimente colocar apenas `tr { ... }` e verá que TODAS as TRs da página ficarão com fundo cinza e não é o que queremos.

Agora, repare no estilo das colunas dessa TR. A primeira é maior que as outras, ocupando 50% do tamanho total da tabela. As demais estão com 15% cada, sendo que a última terá o restante que é 20%.

Código 9.32: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<td style="width:50%">Nome</td>
<td style="width:15%">Idade</td>
<td style="width:15%">UF</td>
<td>Ações</td>
```

Como removemos esses estilos individuais mantendo a diferença de tamanho entre eles?

Uma ideia é criando classes CSS para as duas variações que temos (sendo que a última célula recebe o que sobrar). No entanto, isso fica um tanto verboso demais no HTML e podemos fazer algo mais elegante usando alguns recursos do CSS. Remova os estilos daquelas TDs e coloque o seguinte no seu estilos.css:

Código 9.33: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

table thead tr td:first-child { width:50%; }

```
table thead tr td:last-child { width:20%; }
```

Aqui usei os modificadores first-child e last-child para dizer que a primeira e última células da linha (tr) da thead possuem tamanho fixado em 50% e 20%, respectivamente, sobrando outros 15% para cada uma das células intermediárias.

Essa é uma notação mais profissional e caso você tenha resolvido com classes, não há problema.

Com esse último ajuste conseguimos eliminar todos os estilos que haviam sido escritos diretamente no HTML. Agora é a hora de fazer novos estilos para adequar nosso HTML atual à imagem do início desse exercício.

Primeiro, não é exatamente um estilo mas incomoda: use o atributo cellspacing da table para remover os espaços entre as células:

Código 9.34: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<table cellspacing="0">
```

Agora, vamos começar adicionando uma margem interna lateral pois tudo está muito "grudado" na esquerda da página. Fazemos isso definindo um estilo para a tag body no estilos.css:

Código 9.35: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
body{ padding-left: 10px; }
```

Já que a primeira coisa que vemos é o formulário de cadastro, vamos começar por ele, estilizando o form:

Código 9.36: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
form{  
    width: 50%;  
    text-align: right;  
}
```

Dei uma largura máxima de 50% e alinhei-o à direita, pois isso deixará o alinhamento de labels e campos mais interessante.

Agora para deixar todos os nossos inputs mais agradáveis visualmente, crie um estilo global para todos eles, deixando-os mais arredondados (border-radius) e com mais espaçamento interno (padding):

Código 9.37: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
input {  
    padding: 5px;  
    border-radius: 4px;  
}
```

Você vai notar que isso não afeta o select, uma vez que ele não é um input. Mesmo que você coloque essas mesmas propriedades CSS em um estilo para o select não vai adiantar pois os navegadores possuem aparências próprias pra ele. Para conseguir estilizar o select você precisa primeiro remover o estilo atual dele, como abaixo:

Código 9.38: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
select{  
    -webkit-appearance: none;  
    -moz-appearance: none;  
    appearance: none;  
    padding: 5px;
```

```
    width: 53%;  
}
```

As propriedades iniciadas com '-' são exclusivas de alguns browsers como Firefox (moz) e Chrome (webkit). Assim conseguiremos que inputs e selects fiquem mais parecidos visualmente.

Opa, mas a largura deles está diferente, não? Isso porque o select está com um width de 53% da largura do contâiner externo. Vamos criar estilos para a largura dos inputs do formulário também:

Código 9.39: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
input[type="text"], input[type="number"] { width: 50%; }
```

Aqui usei um truque de CSS para aplicar um estilo a dois tipos de componentes diferentes, separados por vírgulas. Assim, todos os textos (como o campo de nome) e os números (como o campo de idade) terão a mesma largura, que apesar de ser 3% menor que o select, é visualmente idêntica.

Mas e os botões? Se você olhar a imagem verá que eles são do mesmo tamanho. Sendo assim, vamos criar um estilo idêntico para os dois:

Código 9.40: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
#btnListar, input[type="submit"] { width: 25%; }
```

Como o btnListar tem id, usamos ele. Já o outro botão tratei como um submit genérico, visto que é o único na página.

Pra completar nossa página, vamos apenas alinhar o título de maneira mais adequada ao formulário, definindo o seguinte estilo:

Código 9.41: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
h2 {  
    text-align: right;  
    padding-right: 50%;  
}
```

Se eu colocasse apenas um text-align para direita, ele ia grudar no lado direito da tela. Então resolvi colocar um padding de 50% na direita para empurrá-lo de volta mais para o centro. Note que isso é completamente diferente de usar um "text-align: center", teste no seu navegador verá a diferença.

E agora a nossa tabela!

Vamos começar botando um padding para todas as informações das células ficarem menos coladas em relação às bordas, bem como vamos centralizar essas informações:

Código 9.42: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
table tr td{  
    padding: 5px;  
    text-align: center;  
}
```

Eu particularmente não gosto do nome centralizado pois ele é muito comprido, então vou adicionar outro estilo apenas para a primeira célula da tabela, usando o recurso ':first-child' novamente:

Código 9.43: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
table tr td:first-child{ text-align: left; }
```

E por fim, a cereja do bolo, vamos colocar um efeito CSS que vai trocar a cor do fundo de uma linha da tabela quando o mouse estiver passando por cima dela:

Código 9.44: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
table tbody tr:hover{  
    background-color: #ccc;  
}
```

Isso é particularmente útil em tabelas muito grandes, para facilitar a leitura. Obtemos este efeito de sobreposição graças ao efeito ':hover' que aplicamos em todas as TRs, definindo uma nova cor de fundo.

E com isso finalizamos esta seção para partir para coisas mais avançadas em estilos!

Bootstrap

Bootstrap é uma coleção gratuita e open-source de ferramentas para criação de websites e aplicações web. Ele contém modelos de design baseados em CSS e HTML para tipografia, formulários, botões, navegação e outros componentes de interface, bem como extensões opcionais Javascript. Ele objetiva facilitar o desenvolvimento de websites dinâmicos e aplicações web.

Bootstrap, originalmente chamado Twitter Blueprint, foi desenvolvido por Mark Otto e Jacob Thornton no Twitter, como um framework para encorajar consistência entre os projetos internos da empresa. Antes do Bootstrap, muitas bibliotecas foram usadas no desenvolvimento da interface, o que levou a inconsistências e alto custo de manutenção.

Bootstrap é compatível com as últimas versões dos navegadores Google Chrome, Firefox, Internet Explorer, Opera e Safari. Desde a versão 2.0 possui design responsivo e à partir da versão 3.0 adotou a filosofia mobile-first, que prioriza a construção de aplicações web e sites com boa aparência mesmo em dispositivos móveis.

Usaremos neste livro a versão 4.5, que é estável e pode ser baixada (zip) nesta página: <https://getbootstrap.com/docs/4.5/getting-started/download/>

Use o primeiro botão de Download que aparece na página do link acima, baixe o zip e descompacte-o antes de avançar.

Você vai precisar baixar mais uma dependência da qual o Bootstrap precisa para funcionar plenamente, chamada jQuery. Falarei em mais detalhes dele, se quiser conhecer, no Apêndice 2 ao final deste livro.

Por enquanto, apenas acesse <https://jquery.com> e baixe a versão mais recente na sua máquina.

Como usar

Primeiro, copie o arquivo do jQuery com o final ".min.js" para a pasta js da mesma aplicação que estávamos mexendo neste capítulo.

Nota: um arquivo 'min' é um arquivo que passou por uma técnica de minificação, para ficar menor e mais leve, sem perder o seu conteúdo. São arquivos que se você abrir, não vai entender nada pois os nomes de variáveis foram trocados, os espaços removidos e muito mais. Como não vamos editar estes arquivos, usaremos a versão minificada de cada um.

Os demais arquivos que vieram junto no download são irrelevantes neste momento.

Segundo, vá onde você extraiu o zip do Bootstrap. Copie os arquivos css/bootstrap.min.css para a sua pasta CSS e o js/bootstrap.bundle.min.js para a sua pasta js.

Agora na sua página HTML, référencia o arquivo CSS no seu HEAD e os arquivos JS um pouco antes do </body>, respeitando as ordens abaixo:

Código 9.45: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Exemplo DOM</title>
    <link href="css/bootstrap.min.css" rel="stylesheet" />
  </head>
  ...
  <script src="js/jquery.min.js"></script>
  <script src="js/bootstrap.bundle.min.js"></script>
  <script src="js/scripts.js"></script>
</body>
</html>
```

Nota: joguei fora o nosso estilos.css, para não gerar conflitos durante os estudos de Bootstrap.

Ao abrir este arquivo HTML novamente no browser, verá que ele já está completamente diferente. Não exatamente melhor, na verdade bem bagunçado. Temos de entender alguns conceitos importantes do Bootstrap antes que nossas aplicações web fiquem realmente belas.

Primeiro, uma vez que Bootstrap é um framework responsivo orientado ao desenvolvimento para múltiplos dispositivos, para que ele funcione corretamente é necessário incluir a tag META VIEWPORT no HEAD do seu documento, como abaixo. Isso fará com que o conteúdo da sua página se ajuste à largura do dispositivo que estiver visualizando ela.

Código 9.46: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
```

Para que o viewport funcione corretamente devemos definir o nosso container principal dentro da página HTML. As características fluidas e responsivas serão aplicadas a esse container. Para definir o seu container basta colocar a classe CSS container nesse elemento (geralmente uma DIV).

Código 9.47: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<!DOCTYPE html>
<html>
<head>
<title>Exemplo Bootstrap</title>
<meta charset="utf-8" />
```

```

<meta name="viewport" content="width=device-width, initial-
scale=1, shrink-to-fit=no">
<link href="css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
<div class="container">
    ... todo o restante do body aqui ...
</div>
<script src="js/jquery.min.js"></script>
<script src="js/bootstrap.bundle.min.js"></script>
<script src="js/scripts.js"></script>
</body>
</html>

```

Estrutura de página

Dentro do container da página o Bootstrap organiza-se em linhas e colunas, em um modelo chamado Grid System. Você pode ter quantas linhas quiser em uma página, mas apenas 12 colunas por linha. Uma vez que o design é responsivo, a largura de cada coluna será automaticamente reajustada de maneira igual conforme a tela do dispositivo, desde que se use as classes CSS corretamente.

Cada linha da sua página deve ser definida com a classe row e dentro das linhas nós devemos ter divs como colunas. Cada coluna deve ser definido com uma classe col-x-y, onde x é o tipo de dispositivo mínimo que vai rodar esta página (conforme legenda abaixo) e y é o tamanho da coluna, sendo que a soma das colunas deve totalizar 12.

- **col**: coluna sem tamanho pré-definido;
- **col-1**: coluna tamanho um para smartphones
- **col-sm-1**: coluna tamanho um para tablets
- **col-md-1**: coluna tamanho um para desktops

- **col-lg-1**: coluna tamanho um para telas grandes (monitores HD)
- **col-xl-1**: coluna tamanho um para telas enormes (monitores full-HD, TVs, etc)

Exemplo de linha (row) usando 3 colunas de tamanho 4 (totalizando tamanho 12), definindo tamanho mínimo de tela como tablets:

Código 9.48: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div class="row">
    <div class="col-sm-4">.col-sm-4</div>
    <div class="col-sm-4">.col-sm-4</div>
    <div class="col-sm-4">.col-sm-4</div>
</div>
```

Outro exemplo de linha (row) usando 2 colunas de tamanhos 4 e 8 (totalizando tamanho 12), definindo tamanho mínimo de tela como tablets:

Código 9.49: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div class="row">
    <div class="col-sm-4">.col-sm-4</div>
    <div class="col-sm-8">.col-sm-8</div>
</div>
```

Se o tamanho da coluna não for passado (junto ao nome da classe), elas terão tamanhos iguais, dividindo a largura da linha entre elas.

Em nosso projeto web que chamamos lá atrás de exemploquery, vamos organizar o topo usando uma única linha e coluna (eu mudei o título e subtítulo):

Código 9.50: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div class="container">
  <div class="row">
    <div class="col">
      <h1>Exemplo Bootstrap</h1>
      <span class="text-muted">Sistema para exemplificar uso
de Bootstrap</span>
    </div>
  </div>
```

Tipografia com Bootstrap

Por padrão todas as fontes em páginas Bootstrap são tamanho 16px e as linhas possuem altura 1.428. Além disso os parágrafos `<p>` possuem uma margem extra no rodapé de 10px.

Títulos em Bootstraps são definidos usando as tags `<h1>` a `<h6>`, enquanto que adicionei um texto com classe `text-muted`, gerando o efeito abaixo.

Exemplo Bootstrap

Sistema para exemplificar uso de Bootstrap

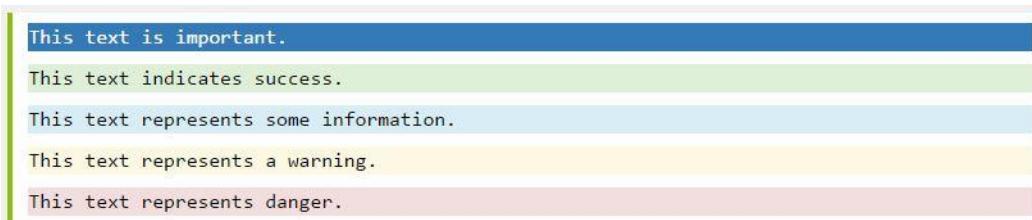
Para alterar cor de texto existem classes pré-definidas no Bootstrap, são elas: `text-muted`, `text-primary`, `text-success`, `text-info`, `text-warning` e `text-danger` como nos exemplos abaixo, respectivamente (usei a `text-muted` anteriormente):



This text is muted.
This text is important.
This text indicates success.
This text represents some information.
This text represents a warning.
This text represents danger.

Já se queremos trocar a cor de fundo dos textos, usandos as classes de background `bg-primary`, `bg-success`, `bg-info`, `bg-warning`

e bg-danger, respectivamente.



Nota: repare no padrão de nomenclatura nos sufixos primary (azul escuro), danger (vermelho), info (azul claro), etc. O Bootstrap segue rigidamente esses padrões para todos os efeitos de seus componentes. Mesmo que você não goste dessas cores, se você estiver usando o padrão 'primary' para a cor principal das coisas, basta mudar essa cor no CSS de tema da sua aplicação e tudo que é 'primary' seguirá a o novo padrão de cor.

Ou seja, basta mudar o tema que toda sua aplicação muda de aparência junto, desde que seguindo as regras do Bootstrap.

Outras classes relacionadas à tipografia incluem:

- **text-left:** alinha o texto à esquerda
- **text-right:** alinha o texto à direita
- **text-uppercase:** capitaliza o texto
- **list-inline:** coloca todos itens de uma lista em uma linha só

A referência completa de elementos para tipografia pode ser encontrada em:

<https://getbootstrap.com/docs/4.5/content/typography/>

Tabelas com Bootstrap

Lembra que no exercício anterior nós perdemos um tempão e escrevemos vários estilos CSS para deixar nossa tabela...ainda feia?

O primeiro passo para estilizar tabelas com Bootstrap e que já resolve muita coisa é adicionar a classe 'table' à sua tag <TABLE>,

que deixa as tabelas com a aparência abaixo por padrão:

Código 9.51: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<table cellspacing="0" class="table">
```

Já a classe `table-striped` cria o efeito "zebrado" na tabela.

Código 9.52: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<table cellspacing="0" class="table table-striped">
```

A classe `table-bordered` adiciona borda à sua tabela, `table-hover` adiciona efeito de mouse-over nas linhas, `table-condensed` diminui os espaços na tabela, `table-responsive` torna a tabela responsiva em larguras menores que 768px e por aí vai. Não esquecendo que você pode combinar os efeitos adicionando mais de uma classe, como abaixo.

Listagem

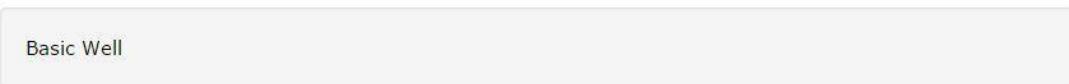
Nome	Idade	UF	Ações
Luiz	29	RS	<input type="button" value="X"/>
Teste	28	RS	<input type="button" value="X"/>
jquery	15	SC	<input type="button" value="X"/>
ajax	12	SC	<input type="button" value="X"/>

Código 9.53: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<table cellspacing="0" class="table table-striped table-bordered table-hover">
```

DIVs e SPANs Bacanas

A classe 'well' coloca uma borda arredondada em qualquer div, como a renderizada abaixo.



Basic Well

Código 9.54: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div class="well">Basic Well</div>
```

Já quando queremos criar “wells” para exibir mensagens, podemos usar a classe alert ao invés disso. A classe alert sempre é usada em pares, sendo a tag contextual alert e outra tag alert com o estilo da div, como alert-danger, alert-success, alert-warning e alert-info, como nos exemplos abaixo.

Código 9.55: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div class="alert alert-success">
  <strong>Success!</strong> This alert box indicates a successful
  or positive action.
</div>
```

Como resultado temos:



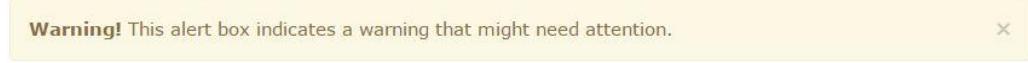
Success! This alert box indicates a successful or positive action.

×



Info! This alert box indicates a neutral informative change or action.

×



Warning! This alert box indicates a warning that might need attention.

×



Danger! This alert box indicates a dangerous or potentially negative action.

×

Um efeito semelhante pode ser obtido com a classe panel:

Esse comportamento é muito interessante para, por exemplo, substituir os nossos alerts de sucesso quando um cliente é cadastrado ou excluído. Para fazer isso, primeiro adicione o seguinte trecho de código HTML em nosso index.html, na divListagem, logo abaixo do h2:

Código 9.56: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div id="alertListagem" class="alert alert-success">
    <strong>Sucesso!</strong> Blá blá blá!
</div>
```

Para que essa div venha escondida por padrão, adicione o seguinte código no evento DOMContentLoaded do nosso scripts.js:

Código 9.57: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
document.addEventListener("DOMContentLoaded", function(event)
{
    const alertListagem = document.querySelector('#alertListagem');
    alertListagem.style.display = "none";
```

E agora em nosso scripts.js, vamos substituir no evento de submit do formulário, na hora que retorna da função updateDatabase, vamos trocar o uso do alert padrão do browser pela exibição e ocultação dessa div de sucesso, passando a mensagem adequada:

Código 9.58: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
updateDatabase(data)
    .then(result => {
        const cliente = result.ops[0];
```

```

        alertListagem.innerHTML = `<strong>Sucesso!</strong>
Cliente ${cliente.nome} cadastrado com sucesso!';
        alertListagem.style.display = "block";
        setTimeout(() => { alertListagem.style.display = "none" },
2000);

        updateTable(cliente);
    }
    .catch(error => alert(`Ocorreu um erro: ${error}`));

```

Note que primeiro eu troquei o html interno da div de alerta usando a propriedade innerHTML. Isso permite que tenhamos apenas um alerta e que possamos trocar seu texto conforme e necessidade. Além disso, eu defini um setTimeout para esconder novamente esta mensagem após 2000ms.

O Bootstrap tem formas até mais poderosas de fazer este tipo de mensagem aparecer e desaparecer, mas não quero complicar muito.

Como resultado temos a exibição dele (e sua posterior ocultação) ao fazermos um cadastro:

Listagem

Sucesso! Cliente cadastrado com sucesso!			
Nome	Idade	UF	Ações
Luiz	29	RS	<input type="button" value="X"/>

Lhe encorajo a encapsular este código em uma function específica e usá-la nos pontos da sua aplicação que precisem desse tipo de comportamento, vou chamar a minha de exibirMensagem, você pode consultar como ficou nos fontes deste livro.

Outro uso bem interessante desses alertas são para mensagens de erro. Por exemplo, nosso formulário de cadastro não deveria permitir

salvar clientes sem nome (pela regra do nosso banco de dados, lembra?). Nesses casos, deveria exibir uma mensagem de erro ao usuário.

Para fazer isso, vamos adicionar uma div com as classes de alerta de erro na divCadastro, logo abaixo do h2:

Código 9.59: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div id="alertCadastro" class="alert alert-danger">
    <strong>Erro!</strong> Nome é obrigatório!
</div>
```

Esconda esta div no evento DOMContentLoaded, do mesmo jeito que fizemos com a outra div da área de listagem, setando o seu display para none. Tente fazer por conta, observando o código que já em e, se ficar em dúvida, consulte os fontes do projeto.

Agora o que precisamos fazer é no evento de submit do form, verificarmos se os campos obrigatórios foram preenchidos. Caso contrário, não podemos enviar os dados para a API e exibiremos a mensagem de erro:

Código 9.60: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
const frmCadastro = document.getElementById("frmCadastro");
frmCadastro.onsubmit = (evt) => {
    evt.preventDefault();

    if(!document.querySelector('input[name="nome"]').value){
        alertCadastro.innerHTML = '<strong>Erro!</strong> O campo
        nome é obrigatório!';
        alertCadastro.style.display = "block";
        setTimeout(() => { alertCadastro.style.display = "none" },
        3000);
        return false;
    }
}
```

```

    }

const data = new FormData(frmCadastro);
updateDatabase(data)
  .then(result => {
    const cliente = result.ops[0];
    exibirMensagem('#alertListagem', `<strong>Sucesso!
</strong> Cliente ${cliente.nome} cadastrado com sucesso!`);
    updateTable(cliente);
  })
  .catch(error => alert(`Ocorreu um erro: ${error}`));
}

```

Meu teste foi bem simples: eu selecionei com JavaScript o campo nome e vejo se o value dele está vazio. Se não tiver sido preenchido, exibiremos o alerta de erro, que em poucos segundos se fecha sozinho.

Note que coloquei o evt.preventDefault() para o topo também, para garantir que ele vai cancelar o comportamento original do submit e que também coloquei um return false ao final da exibição do erro, para garantir que não vá acontecer as etapas seguintes do nosso código.

O resultado é esse quando se tenta salvar um cadastro sem digitar o campo obrigatório:

Cadastro

Erro! Nome e idade são dados obrigatórios!

Nome:

Idade:

UF: RS

|

Agora quando o assunto são SPANs, o Bootstrap possui muitas opções, como a classe label, que quando adicionado a um SPAN cria um efeito visual muito interessante:

```
<span class="label label-default">Default Label</span>
<span class="label label-primary">Default Label</span>
<span class="label label-success">Default Label</span>
<span class="label label-info">Default Label</span>
<span class="label label-warning">Default Label</span>
<span class="label label-danger">Default Label</span>
```



Botões bacanas

É muito simples criar botões estilosos com Bootstrap, basta adicionarmos a classe btn, seguido de outra classe btn-x onde x é o tipo de botão, dentre os seguintes: btn-danger, btn-success, btn-primary, btn-link, btn-warning, btn-info e btn-default todos representados abaixo:



Já o tamanho do botão é definido com as classes btn-lg, btn-md, btn-sm e btn-xs, como segue:



Ou use btn-block que faz o botão ocupar toda largura da tela. Além disso, caso queira desabilitar um botão, você pode usar a classe disabled.

Experimente aplicar a classe `btn` em todos os botões do nosso `index.html`, bem como a classe `btn-danger` nos botões de exclusão e `btn-primary` no botão de salvar para que você tenha um resultado muito profissional facilmente (apenas um exemplo abaixo):

Código 9.61: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<input type="submit" value="Salvar" class="btn btn-primary" />
```

Cadastro

Nome:

Idade:

UF:

RS ▲

Listar

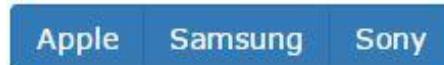
Salvar

Caso queira agrupar seus botões, experimente colocar eles dentro de uma única DIV e nesta div colocar a classe `btn-group` (ou `btn-group-vertical` para criar na vertical e `btn-group-justified` para ocupar toda largura da tela) o que resulta no exemplo abaixo:

Código 9.62: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div class="btn-group">
    <button type="button" class="btn btn-primary">Apple</button>
    <button type="button" class="btn btn-primary">Samsung</button>
```

```
<button type="button" class="btn btn-primary">Sony</button>
</div>
```



Agora se você quer criar um grupo de botões para usar com menus suspensos, dê uma olhada no exemplo abaixo, que é simples mas que prova resultados muito bons:

Código 9.63: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div class="btn-group">
  <button type="button" class="btn btn-primary">Apple</button>
  <button type="button" class="btn btn-primary">Samsung</button>
  <div class="btn-group">
    <button type="button" class="btn btn-primary dropdown-toggle" data-toggle="dropdown">Sony <span class="caret"></span></button>
    <ul class="dropdown-menu" role="menu">
      <li><a href="#">Tablet</a></li>
      <li><a href="#">Smartphone</a></li>
    </ul>
  </div>
</div>
```



Mas como podemos aproveitar isso em nosso exemplo que estamos trabalhando?

Em nossa função updateTable (scripts.js), troque a linha que gera a linha de cada cliente na tabela por essa, que modificou o botão de excluir para o novo padrão:

Código 9.64: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
linha += `<tr><td>${cliente.nome}</td><td>${cliente.idade}</td>
<td>${cliente.uf}</td><td><input type="button" class="btn btn-
danger" value="X" data-id="${cliente._id}" /></td></tr>`;
```

Não esqueça de exibir nossa nova mensagem de sucesso quando um cliente for excluído. Abaixo, um exemplo de function genérica que você pode ter no scripts.js que, quando chamada, faz o comportamento de exibir e ocultar um element com o seletor que você passar.

Código 9.65: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
function exibirMensagem(selector, html){
  const div = document.querySelector(selector);
  div.innerHTML = html;
  div.style.display = "block";
  setTimeout(() => { div.style.display = "none" }, 2000);
}
```

FORMs com Bootstrap

Já fizemos diversas melhorias de aparência na nossa index.html, tornado-a muito mais agradável e porque não dizer, profissional. No entanto, alguns itens do nosso HTML FORM ainda estão deixando a desejar. O primeiro passo para estilizarmos nosso FORMs usando Bootstrap é na própria tag FORM colocarmos o atributo role="form".

Código 9.66: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<form id="frmCadastro" action="" method="" role="form">
```

Cada linha do nosso formulário deve estar dentro de uma DIV com a classe form-group e cada controle INPUT (e o select também) deve ter a classe form-control.

Código 9.67: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<form action="" method="" role="form">
  <div class="form-group">
    <label>Nome: <input type="text" name="nome" class="form-control" /></label>
  </div>
  <div class="form-group">
    <label>Idade: <input type="number" name="idade" class="form-control" /></label>
  </div>
  <div class="form-group">
    <label>UF: <select name="uf" class="form-control">
      <option>RS</option>
      <option>SC</option>
      <option>PR</option>
      <!-- coloque os estados que quiser -->
    </select></label>
  </div>
  <div class="form-group">
    <input type="button" id="btnListar" value="Listar" class="btn btn-default" /> | <input type="submit" value="Salvar" class="btn btn-primary" />
  </div>
</form>
```

Cadastro

Nome:

Idade:

UF:

RS ▲

Listar | Salvar

DIVs com checkbox dentro devem usar a classe checkbox ao invés de form-group e se quiséssemos usar radiobuttons, na DIV usariámos a classe radio como mostrado abaixo ou então a classe radio-inline, para radiobuttons na horizontal (o padrão é vertical).

Código 9.68: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div class="radio">
    <label><input type="radio" name="optradio">Option 1</label>
</div>
<div class="radio">
    <label><input type="radio" name="optradio">Option 2</label>
</div>
<div class="radio disabled">
    <label><input type="radio" name="optradio">Option 3</label>
</div>
```

Sempre que quisermos desabilitar um input, podemos usar a classe disabled para isso.

Bootstrap não tem apenas classes para deixar estilosos os componentes de sempre. Existem estilos e efeitos muito poderosos como abas (tab), carrosséis (carousel), modais (modal), tooltips (tooltip) e por aí vai. Sugiro dar uma olhada no site oficial ou nos tutoriais da W3Schools que vale a pena!

JavaScript e Bootstrap com Node.js

Embora tenhamos feito todos os exemplos de JavaScript e Bootstrap usando um index.html simples, tudo que foi visto nestes capítulos sobre front-end aplica-se normalmente à criação de aplicações web usando Node.js, e não apenas usando-o como web API, mas com Express e EJS também, como vimos anteriormente.

No entanto, pequenos cuidados são necessários, tais como:

- a extensão das suas páginas HTML vai ser .EJS;
- suas pastas de fonts, css e js devem ficar dentro da pasta public;
- durante a renderização das páginas você não tem acesso ao DOM tal qual o JavaScript tem, apenas pode usar as server-tags;
- após a renderização das páginas você não tem mais acesso ao back-end Node.js, exceto através do uso de Ajax ou através de submissões de formulários;
- o front-end processa no navegador do usuário, já o back-end processa no seu servidor. Tenha em mente que delegar processamento para o usuário diminui a carga do seu servidor;

Para provar que funciona, vamos voltar rapidamente ao projeto de exemplo do capítulo de MongoDB, que era um CRUD completo em Node.js, lembra? (caso não tenha feito esse projeto, volte ao referido capítulo e faça)

Vou passar os códigos mais rapidamente que o normal pois principalmente a estilização é a praticamente a mesma utilizada no exercício anterior com o index.html.

O express-generator já deixa pastas javascript e stylesheets dentro de public, mas não gosto desses nomes, sugiro mudar para js e css, respectivamente. Caso não se lembre, na pasta js deve ter o bootstrap.bundle.min.js e o jquery.min.js e na pasta css coloque apenas o bootstrap.min.css ou use um Bootstrap modificado na Internet, como eu usarei no exemplo abaixo. Por padrão devia ter

um CSS nessa pasta, que o express-generator cria para nós, mas pode excluí-lo.

Em nossa views/index.ejs existe alguns códigos JS no final do HTML, o que conforme já conversamos antes, não é indicado. Sendo assim, crie um arquivo scripts-index.js na sua pasta public/js e cole os scripts JS da index.ejs nele:

Código 9.69: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
if(location.href.indexOf('delete=true') != -1){
    alert('Cliente excluído com sucesso!');
}
else if(location.href.indexOf('edit=true') != -1){
    alert('Cliente editado com sucesso!');
}
else if(location.href.indexOf('new=true') != -1){
    alert('Cliente cadastrado com sucesso!');
}
else if(location.href.indexOf('erro') != -1){
    alert('Ocorreu um erro!');
}
```

E refencie esse arquivo JS no final do index.ejs:

Código 9.70: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<script src="/js/jquery.min.js"></script>
<script src="/js/bootstrap.bundle.min.js"></script>
<script src="/js/scripts-index.js"></script>
</body>
</html>
```

Abra o index.ejs novamente e troque o HTML do HEAD dele pelo abaixo, para estilizá-lo usando Bootstrap, da mesma maneira que

fizemos no exercício anterior:

Código 9.71: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<head>
  <title>CRUD de Clientes</title>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-
scale=1, shrink-to-fit=no">
  <link href="https://bootswatch.com/4/cyborg/bootstrap.min.css"
rel="stylesheet" />
</head>
```

Aqui estou usando um template de Bootstrap free da Internet, do site Bootswatch. O legal do Bootstrap é isso, você aprende como estilizar a página uma vez e você vai ver que tudo que aprendemos continua funcionando, mas resultando em outra aparência.

Dentro do body da sua index.ejs, substitua os códigos antigos por esse:

Código 9.72: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<div class="container">
  <h1>Listagem de Clientes</h1>
  <p>Clientes já cadastrados no sistema.</p>
  <div id="alertListagem" class="alert alert-success">
    <strong>Sucesso!</strong> Cliente cadastrado com sucesso!
  </div>
  <table class="table table-striped table-bordered table-hover">
    <thead>
      <tr><td>Nome</td><td>Idade</td><td>UF</td>
    <td>Ações</td></tr>
    </thead>
    <tbody>
```

```
<% if(!docs || docs.length == 0) { %>
<tr>
  <td colspan="4">Nenhum cliente cadastrado.</td>
</tr>
<% } else {
  docs.forEach(function(customer){ %>
    <tr>
      <td><%= customer.nome %></td>
      <td><%= customer.idade %></td>
      <td><%= customer.uf %></td>
      <td>
        <a href="/edit/<%= customer._id %>" class="btn btn-info">edit</a>
        <a href="/delete/<%= customer._id %>" class="btn btn-danger" onclick="return confirm('Tem certeza?');">x</a>
      </td>
    </tr>
  <% })
}
</tbody>
</table>
<a href="/new" class="btn btn-primary">Cadastrar Novo</a>
</div>
```

Se executar esse projeto com 'npm start' já deve estar com a aparência abaixo:

Nome	Idade	UF	Ações
Luiz	29	RS	<input type="checkbox"/> <input checked="" type="checkbox"/>
Teste	28	RS	<input type="checkbox"/> <input checked="" type="checkbox"/>
jquery	15	SC	<input type="checkbox"/> <input checked="" type="checkbox"/>
ajax	12	SC	<input type="checkbox"/> <input checked="" type="checkbox"/>
rafael	25	RS	<input type="checkbox"/> <input checked="" type="checkbox"/>
orestes	67	PR	<input type="checkbox"/> <input checked="" type="checkbox"/>
maria	16	RS	<input type="checkbox"/> <input checked="" type="checkbox"/>
cecilia	31	RS	<input type="checkbox"/> <input checked="" type="checkbox"/>

Legal, não?

Note que nosso alerta de sucesso está sempre visível. Vamos corrigir isso?

É bem fácil, abra nosso public/js/scripts-index.js e mude completamente o código lá de dentro, pois não usaremos mais aqueles alerts antiquados:

Código 9.73: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```

const alertListagem = document.getElementById("alertListagem");

function alertar(mensagem){
    alertListagem.innerHTML = `<strong>Sucesso!</strong>
${mensagem}`;
    alertListagem.style.display = "block";
    setTimeout((() => { alertListagem.style.display = "none"; }), 2000);
}

if(location.href.indexOf('delete=true') != -1){
    alertar('Cliente excluído com sucesso!');
}
else if(location.href.indexOf('edit=true') != -1){

```

```
        alertar('Cliente editado com sucesso!');
    }
} else if(location.href.indexOf('new=true') != -1){
    alertar('Cliente cadastrado com sucesso!');
}
else if(location.href.indexOf('erro') != -1){
    alertar('Ocorreu um erro!');
}
else{
    alertListagem.style.display = "none";
}
```

Isso vai apresentar a mensagem correta conforme a ação que foi realizada ou ocultar o alerta caso tenha apenas iniciado o sistema nesta tela.

Vamos arrumar agora o views/new.ejs?

Primeiro o HEAD:

Código 9.74: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<head>
    <title>CRUD de Clientes</title>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1, shrink-to-fit=no" />
    <link href="https://bootswatch.com/4/cyborg/bootstrap.min.css"
rel="stylesheet" />
</head>
```

Depois o BODY:

Código 9.75: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<div class="container">
  <h1>
    <%= title %>
  </h1>
  <p>Preencha os dados abaixo para salvar o cliente.</p>
  <div id="alertCadastro" class="alert alert-danger col-sm-6">
    <strong>Erro!</strong> Nome é um dado obrigatório!
  </div>
  <form id="frmCadastro" action="<%= action %>" method="POST" role="form" class="form-horizontal">
    <div class="form-group">
      <label class="control-label" for="nome">Nome:<br/>
        <input type="text" id="nome" name="nome" value="<%=
doc.nome %>" class="form-control" />
      </label>
    </div>
    <div class="form-group">
      <label class="control-label" for="idade">Idade:<br/>
        <input type="number" id="idade" name="idade" value="<%=
<%= doc.idade %>" class="form-control" />
      </label>
    </div>
    <div class="form-group">
      <label class="control-label">UF:<br/>
        <select name="uf" class="form-control">
          <% const s = "selected" %>
          <option <% if(doc.uf === "RS") { %><%= s %><% } %>>RS</option>
          <option <% if(doc.uf === "SC") { %><%= s %><% } %>>SC</option>
          <option <% if(doc.uf === "PR") { %><%= s %><% } %>>PR</option>
        <!-- coloque os estados que quiser -->
        </select>
      </label>
    </div>
    <div class="form-group">
```

```

    <a href="/" class="btn btn-default">Cancelar</a> <input
type="submit" value="Salvar" class="btn btn-primary" />
    </div>
</form>
</div>

```

O resultado desse BODY pode ser conferido na imagem abaixo:

The screenshot shows a dark-themed web page titled 'Cadastro de Cliente'. Below the title, a sub-instruction reads 'Preencha os dados abaixo para salvar o cliente.' A red horizontal bar contains the error message 'Erro! Nome e idade são dados obrigatórios!' (Error! Name and age are mandatory fields!). Below the message are three input fields: 'Nome:' (Name), 'Idade:' (Age), and 'UF:' (UF). The 'Nome:' field is empty. The 'Idade:' field is empty. The 'UF:' field contains 'RS' and has a dropdown arrow icon. At the bottom are two buttons: 'Cancelar' (Cancel) in grey and 'Salvar' (Save) in blue.

Massa hein!

E para programar o nosso alerta de campos obrigatórios, vamos criar um novo arquivo chamado scripts-new.js na pasta public/js:

Código 9.76: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```

document.addEventListener('DOMContentLoaded', (event) => {
  const alertCadastro =
    document.getElementById("alertCadastro");
  alertCadastro.style.display = "none";

  document.getElementById("frmCadastro").onsubmit = (evt) => {

```

```
if(!document.getElementById("nome").value){  
    alertCadastro.style.display = "block";  
    setTimeout(() => { alertCadastro.style.display = "none"; },  
2000);  
    evt.preventDefault()  
}  
}  
})
```

Esse exemplo é um pouco diferente pois vamos interceptar o submit do form, mas não vamos anular o submit (com event.preventDefault()), pois depois que execute nosso código JS queremos que o submit ocorra normalmente. A menos, é claro, que a validação dê errado, daí cancelaremos o submit.

Atenção: confiar sua segurança apenas em validação JS client-side é loucura, pois o usuário pode desabilitar o JS no navegador. O ideal é possuir validação no back-end também, essa sim segura se bem executada.

Agora basta referenciar esse arquivo JS no final do new.ejs e tudo deve voltar a funcionar como antes, mas muito mais bonito. :)

Código 9.77: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
<script src="/js/jquery.min.js"></script>  
<script src="/js/bootstrap.min.js"></script>  
<script src="/js/scripts-new.js"></script>
```

E com isso terminamos mais este capítulo de front-end!

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Autenticação com Passport

Aprenda a fazer funcionalidade de login profissional com o pacote Passport.

<https://www.luiztools.com.br/post/autenticacao-em-node-js-com-passport/>

Socket.io

Aprenda a criar uma aplicação de chat em tempo real usando Socket.io no Node.js.

<https://www.luiztools.com.br/post/criando-um-chat-com-nodejs-e-socketio/>

Casos reais

Artigo sobre porque aprender Node.js vale a pena, considerando casos reais de empresas bem sucedidas com esta tecnologia.

<https://www.luiztools.com.br/post/por-que-aprender-nodejs/>

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

10 Boas práticas com Node.js

*I'm not a great programmer;
I'm just a good programmer with great habits.*
- Kent Beck

Quando estamos começando com uma nova tecnologia sempre tudo é muito confuso, complexo e, por que não dizer, assustador?

Trabalho desde 2006 com programação e antes disso eu já programava de maneira não profissional, desde um técnico em eletrônica que fiz em 2004 que incluía programação de microcontroladores para a indústria. Nesse ínterim tive de trocar sucessivas vezes de tecnologias, seja por questões acadêmicas, seja por questões de mercado: Assembly, C/C++, Lisp, Prolog, Visual Basic, Java, Lua, .NET, PHP e, agora, Node.js.

E a cada troca eu volto a ser um calouro, tendo de aprender não apenas sintaxe, semântica, todo o "ferramental" relacionado à tecnologia, mas o mais difícil de tudo: boas práticas!

Como saber se o código que estou escrevendo está dentro dos padrões de mercado?

Como tirar o máximo proveito da linguagem através do uso das melhores ferramentas e extensões?

Como é a arquitetura ou o mindset correto para os tipos de projetos que são criados com esta tecnologia?

Como ...

E tudo fica pior quando estamos falando de uma tecnologia que tem pouco mais de uma década de vida e até poucos anos atrás era coisa de devs malucos em eventos. São muitas as dúvidas sobre

Node.js e, mesmo com o todo-poderoso Google em nossas mãos, parece que essas respostas estão desconectadas e jogadas cada uma em um canto obscuro da Internet, fazendo com que dê um trabalho enorme encontrá-las.

Não trago aqui verdades universais ou respostas para todas as perguntas que você vai ter sobre Node.js, mas trago um apanhado da experiência coletiva de milhares de desenvolvedores ao redor do globo para formar um guia de boas práticas com Node.js. Pessoas com muito mais conhecimento de Node do que eu, e algumas dicas minhas também, afinal eu também sou um desses milhares de desenvolvedores!

Bons Hábitos e Princípios Gerais

Acima de qualquer boa prática que eu possa propor neste livro, vale ressaltar alguns princípios e motivações que devem estar acima de qualquer coisa em seus projetos com Node.js.

Nem todos princípios vão se encaixar apenas no Node.js e talvez você reconheça uma ou outra coisa que já utilize hoje em outra tecnologia, como PHP. Ótimo!

Por outro lado, talvez encontre críticas a hábitos ruins que você possui hoje. Não se assuste, parte do processo de evolução como programadores (e como pessoas!) envolve manter bons hábitos e evitar hábitos ruins o tempo todo.

Afinal, é como dizem, "o hábito faz o monge".

Use Node para os projetos certos

Existe um dito popular que diz: "Nem todo problema é um prego e nem toda ferramenta um martelo".

Qualquer tecnologia que lhe vendam como sendo a solução para todos seus problemas é no mínimo para se desconfiar. Node não é um canivete-suíço e possui um propósito de existir muito claro desde sua homepage até a seção de about no site oficial: ele te proporciona uma plataforma orientada à eventos, assíncrona e focada em I/O e aplicações de rede não-bloqueantes.

Isso não quer dizer que você não possa fazer coisas com ele para as quais ele não foi originalmente concebido, apenas que ele pode não ser a melhor ferramenta para resolver o problema e você pode acabar frustrado achando que a culpa é do Node. Por isso, use Node preferencialmente para:

- APIs;
- Bots;

- Mensageria;
- IoT;
- Aplicações real-time;

Node.js foi criado para async

Isso pode parecer óbvio quando falamos de uma plataforma focada em I/O não-bloqueante, mas parece que tem programador que a todo custo quer mudar isso em suas aplicações, fazendo todo tipo de malabarismo para transformar trechos de código e módulos originalmente assíncronos em tarefas síncronas.

Esqueça. Nem perca seu tempo.

Se não gosta de callbacks (será que alguém gosta?) use promises e async/await, mas não tente transformar Node.js em uma plataforma síncrona pois esse não é o objetivo dele e ele não funciona bem dessa forma, afinal ele é single-thread, lembra? Se realmente precisa que tudo rode sincronamente, use PHP ou outra plataforma que crie threads separadas e seja feliz.

Ajude seu cérebro a entender o projeto

Conseguimos lidar com apenas pequenas quantidades de informação de cada vez. Alguns cientistas falam de sete coisas ao mesmo tempo, outros falam de quatro coisas. Por isso que usamos pastas, módulos e por isso que criamos funções. Elas nos ajudam a lidar com a complexidade do sistema nos permitindo mexer em pequenas porções dele de cada vez.

Crie apenas os diretórios que precisa

Não saia criando dezenas de diretórios que acabarão vazios ou com apenas um arquivo dentro. Comece com o básico de pastas que você precisa e vá adicionando conforme a complexidade for aumentando. Afinal, você não compra um ônibus como primeiro veículo pensando no dia em que pode querer dar carona para muita gente, certo?

Evite o *over engineering* de querer ter a melhor arquitetura possível no "dia um" do seu projeto. Você deve conhecer (e evitar) o termo 'código espaguete', certo? Um projeto com mais pastas do que o necessário é tão nocivo quanto, é o chamado 'código lasanha' (muitas camadas)!

Torne fácil localizar arquivos de código

Os nomes de pasta devem ser significativos, óbvios e fáceis de entender. Código que não é mais usado, bem como arquivos antigos, devem ser removidos do projeto, e não apenas comentados ou renomeados. Afinal, quem gosta de coisas antigas e que não são mais usadas é museu, certo?

Diminua o acoplamento e aumente o reuso de software.

A sugestão aqui é ter mais módulos especialistas na sua aplicação o mais desacoplados possível, até mesmo em projetos separados registrados no NPM, por exemplo. Uma vez que você tenha módulos independentes, sua aplicação fica menos acoplada e a manutenção mais simples.

Uma dica para aumentar o reuso de software mantendo o acoplamento baixo é não colocar código "especializado" nas suas chamadas de model e controller. Por exemplo, se após salvar um cliente no banco você precisa enviar um email pra ele, não coloque o código que envia email logo após o salvamento no banco, ao invés disso, dispare uma função de um módulo de email.

Outra dica é programar com um mindset mais funcional, conforme citado no capítulo sobre codificação.

Uma terceira e última dica é aproveitar os módulos já existentes, sempre procurando a solução para o que você precisa antes de começar a desenvolver ela. A lista de módulos recomendados no

capítulo final é um bom começo, mas use e abuse do site do NPM e do StackOverflow para isso.

Priorize configuration over convention

Não faça coisas mágicas como assumir que o nome de um arquivo é x ou que os uploads vão parar sempre em uma pasta y. Esse hábito é chamado popularmente de *convention over configuration*. Isso é frágil e facilmente quebra sua aplicação quando alguma coisa muda e você não lembra de todas as convenções que criou para sua arquitetura. Programe de maneira que o código em si possa fazer com que o programador entenda todo o projeto.

Use variáveis de ambiente (como será citado mais pra frente) e/ou arquivos JSON de configuração, o que preferir, mas configure o seu projeto adequadamente e de preferência de maneira independente de infraestrutura, para que você consiga subir sua aplicação rapidamente em qualquer lugar.

Nomes de arquivos no formato 'lower-kebab-case'

Use apenas minúsculas, sem acentos e '-' como separador entre palavras. Isso evita problemas de sistemas de arquivos em sistemas operacionais diferentes que sua aplicação Node possa rodar. Ex: `cadastro-cliente.js`

Este é o padrão do NPM para arquivos e pastas, então é melhor seguir esta regra, por mais que não goste dela.

Veja mais regras adiante.

Tenha um padrão de código a.k.a. Guia de Estilo

Muitos programadores que conheço não gostam de padrões de código. No entanto, um ponto importante a se entender é: padrões

de código não são para o programador, mas sim para a empresa ou para a comunidade (no caso dos projetos open-source).

Uma vez que em empresas e projetos abertos existe uma certa rotatividade de profissionais trabalhando no mesmo código, é natural que programadores com diferente visões de como o software deve ser escrito podem mexer nele e fazer com que facilmente não exista coerência na escrita. Padrões evitam isso.

Por mais que alguns podem achar isso preciosismo, só quem já trabalhou em grandes bases e código sem padrão que sabe o quanto difícil é se achar no meio de um projeto bagunçado. Para resolver isso, empresas e projetos de todos os tipos e tamanhos decidem padrões. Um ponto importante aqui é que não é o mesmo padrão para todos(as). A regra é clara: pegue um padrão que você se sinta confortável e que funcione na sua empresa/projeto.

Aqui vai uma sugestão que está ganhando cada vez mais força: StandardJS, disponível em <http://standardjs.com>. É um padrão rígido, inflexível, construído com opiniões fortes mas que rapidamente está dominando diversos projetos e empresas mundo afora. Eu não concordo com tudo o que prega o StandardJS, mas que ajuda a tornar o código muito mais legível e enxuto, isso com certeza.

Para ajudar com projetos já existentes, ele possui um fixer automático que corrige a maior parte dos códigos fora do estilo e, dependendo da ferramenta que estiver usando (eu uso o Visual Studio Code), te dá sugestões real-time do que você está fazendo "errado". Costumo trabalhar em projetos remotos com outros desenvolvedores e é uma excelente maneira de garantir uma padronização entre os códigos escritos por todos. Basta rodar:

Código 10.1: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

standard --fix

e praticamente tudo se resolve!

E se eu não consegui te convencer, a página deles está cheia de empresas/projetos que usem esse estilo, como: NPM, GitHub, ZenDesk, MongoDB, Typeform, ExpressJS e muito mais. Falando de NPM especificamente, muitos, mas muitos, dos mais populares módulos existentes foram escritos usando StandardJS como guia de estilo, entre eles o mocha, dotenv, mssql, express-generator e muito mais.

Boas Práticas na Codificação

Node.js se tornou uma das plataformas mais populares nos últimos anos. Parte desse sucesso se deve ao fato de que é muito fácil começar projetos em Node.js, mas uma vez que você vá além do Olá Mundo básico, saber como codificar corretamente sua aplicação e como lidar com tantos erros pode facilmente se tornar um pesadelo (aliás, como na maioria das linguagens de programação).

Infelizmente, evitar que este pesadelo se torne realidade faz toda a diferença entre uma aplicação sólida em produção e um desastre em forma de software.

Com isso em mente, vamos passar neste capítulo por uma série de boas práticas usando Node.js que te ajudarão a evitar a maioria das armadilhas desta plataforma.

Comece todos os projetos com npm init

Vamos começar do básico, ok?

Todos estamos acostumados a usar o NPM para instalar novos pacotes em nosso projeto, mas sua utilidade vai além disso. Primeiramente, eu recomendo que comece todos os seus projetos usando npm init, como abaixo:

Código 10.2: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
mkdir novo-projeto
cd novo-projeto
npm init
```

Isto faz com que o package.json seja criado, o que permite que você adicione diversos metadados que ajudarão você e sua equipe mais tarde, definindo a versão do node que roda o projeto, quais

dependências ele possui, qual é o comando para inicializar o projeto...opa essa é uma dica quente, vamos explorá-la?

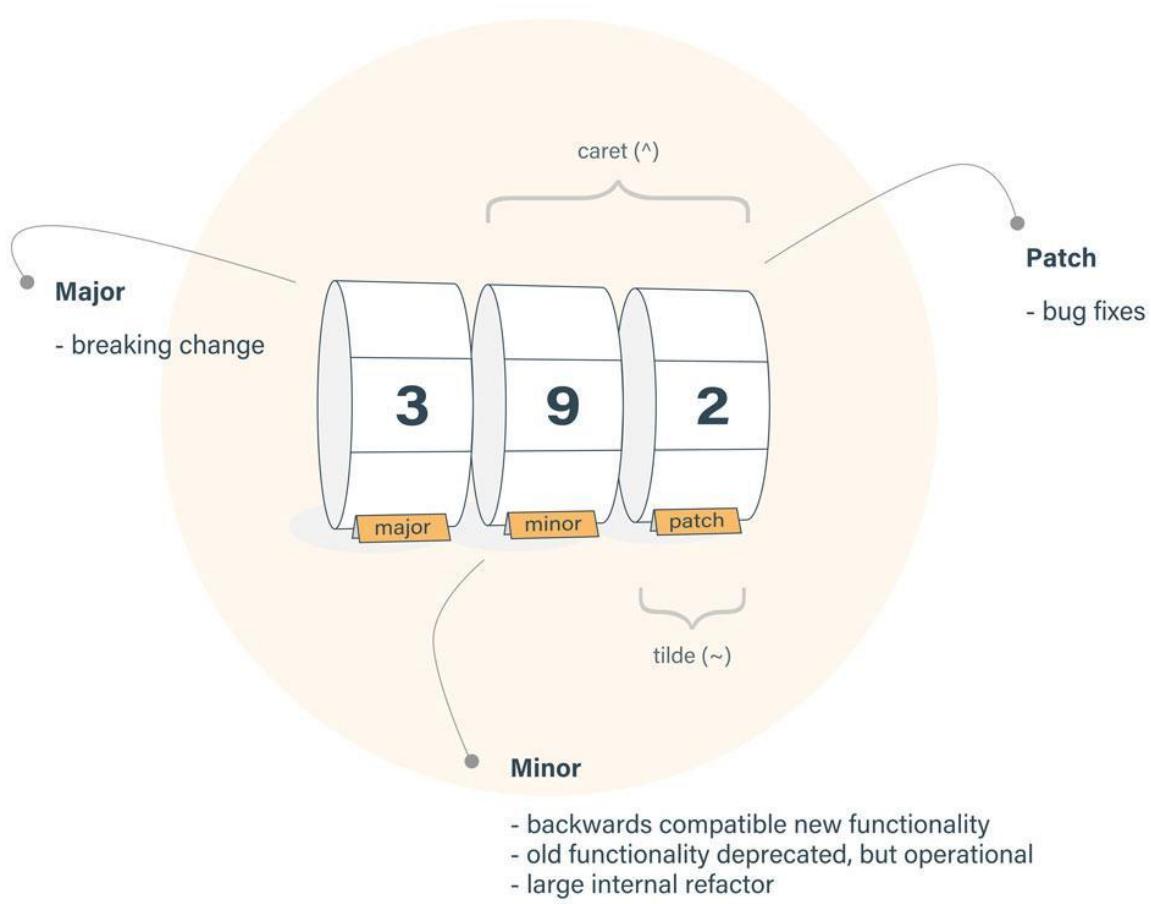
Entenda o versionamento de pacotes

Então você abre o package.json e rapidamente entende que as *dependencies* são os pacotes que sua aplicação usa, mas onde deveriam estar listadas as versões dos pacotes tem um monte de símbolos que não dizem muita coisa.

Resumidamente funciona assim:

- O til garante que o pacote seja sempre carregado respeitando o número do meio da versão. Ex: `~1.2.3` pega o pacote mais recente da versão `1.2.x`, mas não vai atualizar para `1.3`. Geralmente garante que correções de bugs sejam atualizados no seu pacote.
- O circunflexo garante que o pacote seja sempre carregado respeitando o primeiro número da versão. Ex: `^1.2.3` pega o pacote mais recente da versão `1.x`, mas não vai atualizar para `2.0`. Garante que bugs e novas funcionalidades do seu pacote sejam atualizados, mas não novas versões "major" dele.

A imagem abaixo ajuda a entender o template de versões dos pacotes do NPM, que aliás usa um padrão bem comum da indústria de software:



Outros símbolos incluem:

- >, >=, <, <=1.0: a versão deve ser superior, superior ou igual, inferior, inferior ou igual à 1.0, respectivamente.
- 1.2.x: equivalente a ~1.2.0
- *: qualquer versão do pacote
- latest: a versão mais recente do pacote

Se você não tiver símbolo algum, aí o pacote deve ser sempre carregado usando a versão especificada.

Uma última dica bem valiosa para quando se quer atualizar todos pacotes é colocar * na versão de todos e rodar o comando "npm update --save" sobre a pasta do projeto.

Configure scripts no package.json

Lembra da dica de priorizar "configuration over convention" que mencionei lá atrás?

Pois é, em diversas ocasiões temos de rodar scripts para fazer tarefas em nossas aplicações e se tivermos de lembrar qual script rodar, em qual ordem, em qual situação, etc certamente uma hora alguém vai fazer alguma besteira por esquecimento.

Uma constante são os scripts de inicialização, mas existem outros, todos podendo ser configurados na propriedade scripts do package.json, como abaixo:

Código 10.3: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
"scripts": {  
  "start": "node index.js"  
}
```

Neste caso, quando executarmos o comando npm start dentro da pasta deste projeto, ele irá executar "node index.js", sem que tenhamos de escrever este comando manualmente. Ok, aqui foi um exemplo de comando simples, mas poderia ser mais complicado, como passando flags e até mesmo variáveis de ambiente.

Além do script "start" para inicialização, temos:

- **test**: para rodar os seus testes automaticamente (útil para facilitar CI);
- **preinstall**: para executar alguma coisa logo que um "npm install" é chamado, antes da instalação realmente acontecer;
- **postinstall**: para executar alguma coisa depois que um "npm install" acontecer;

E por fim, caso queira scripts personalizados, você pode adicionar o nome que quiser nos scripts, mas para chamá-lo depois deve usar o

comando "npm run-script nomeDoScript", facilitando a sua vida caso seu time tenha vários scripts diferentes.

Use variáveis de ambiente

Uma das premissas centrais do Node.js é o desacoplamento. Mais tarde falaremos de micro serviços (que é uma excelente prática para Node), mas, por ora, entenda que um dos acoplamentos que mais tomam tempo de gerência de configuração são os relacionados à serviços externos como bases de dados, APIs, etc.

Isso porque esses serviços externos possuem diferentes configurações nos diferentes ambientes que sua aplicação Node vai passar: configs de produção, de teste, de homologação, na sua máquina, etc. Como lidar com isso sem que os desenvolvedores tenham de conhecer as configurações de produção, sem ferrar com seu CI, etc? Com variáveis de ambiente!

Sempre que subimos uma aplicação Node podemos passar à ela as variáveis de ambiente daquele processo (`process.env`) antes do comando de inicialização:

```
NOME_VARIAVEL=valor node index.js
```

ou em Windows

```
SET NOME_VARIAVEL=valor  
node index.js
```

Assim, podemos passar configurações locais como variáveis de ambiente para aquela sessão da aplicação, evitando ter connection strings, por exemplo, hard-coded na sua aplicação.

Se você não achou prático, saiba que eu também não gosto de digitar isso aí toda vez que vou executar uma aplicação Node. Sendo assim, uma maneira mais profissional seria usar essa dica em conjunto com a dica anterior, de configurar scripts no package.json. Basta que você adicione as variáveis de ambiente no comando de start, como abaixo:

```
scripts": { "start": "set NOME_VARIAVEL=valor && node app" }
```

No entanto, se seu package.json estiver versionado, e geralmente está, isso pode não ser uma boa, porque ao fazer commit na master ele pode ser enviado ao servidor (se estiver usando CI) e estragar sua aplicação de produção. Neste caso, sugiro uma abordagem ainda mais profissional: **arquivos .env**.

Alguns desenvolvedores criam arquivos de configuração próprios que são lidos na inicialização da aplicação para carregar estas variáveis, no entanto, arquivos .env e o módulo dotenv (e a versão "segura" dotenv-safe) resolvem este problema facilmente. Primeiro, crie o seu arquivo .env (sem nome, apenas ".env" mesmo) e coloque uma variável de ambiente por linha, como abaixo, podendo usar # para comentários:

```
#meu primeiro .env
VARIAVEL1=valor1
VARIAVEL2=valor2
```

Agora adicione no gitignore este arquivo, para ele não ser versionado. Certifique-se de fazer cópias dele nos seus diversos ambientes com os valores de teste, homologação, produção, etc, assim, você nunca envia ele e nunca sobrescreve as configs de produção.

Para carregá-lo, é muito simples, chame esta linha de código na primeira linha do arquivo principal da sua aplicação, para que ele carregue as variáveis de ambiente antes de tudo:

Código 10.4: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
require('dotenv').config()
```

E *voilá!* Está pronto.

A única diferença entre dotenv e dotenv-safe (que eu prefiro, aliás) é que o segundo exige um arquivo `.env.example` que descreva todas as variáveis de ambiente obrigatórias que sua aplicação necessita para funcionar (apenas nomes das variáveis, uma por linha). Caso na inicialização não seja encontrada alguma das variáveis, dá um erro e a aplicação não sobe.

Carregue suas dependências antecipadamente

Muitos desenvolvedores Node.js programam desta forma os seus `requires`:

Código 10.5: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
app.get("/meu-servico", function(request, response) {
  var datastore = require("db-module")(config);
  datastore.get(req.query.someKey)
  // etc, ...
})
```

O problema com o código acima é que quando alguém faz uma requisição para `/meu-servico`, o código então irá carregar todos os arquivos requeridos por "db-module" - o que pode causar uma exceção caso estejam bugados. Adicionalmente, quando passamos

config para o objeto pode acontecer um erro de configuração (ou até de conexão) que derrube o processo inteiro. Além disso, não sabemos quanto tempo aquele require, que funciona de maneira síncrona, irá levar e isso é terrível pois ele bloqueia todas as requisições até que termine.

O problema só não é pior porque depois que o require sobre aquele módulo for chamado uma vez, ele ficará em cache. Sendo assim, todas as suas dependências devem ser carregadas e configuradas antecipadamente, o que ajudará a descobrir erros de conexão e configuração bem cedo e deixar o funcionamento da aplicação mais fluida.

Centralize a criação de conexões ao banco

Passe as conexões criadas para os subsistemas ao invés de ficar passando informações de conexão por parâmetro e permitindo que eles criem suas próprias conexões.

Claro que toda regra tem sua exceção caso venha a utilizar Node.js com bancos relacionais em PaaS, talvez tenha que abrir e fechar as conexões por conta própria e não poderá ficar repassando o objeto entre as funções, ainda assim você pode centralizar o acesso à dados em um módulo específico para tal, como um db.js, por exemplo.

Mindset funcional

Você já deve ter ouvido falar que funções são objetos de primeira classe em JavaScript, certo? Isto é dito porque em JS funções não são tratadas de maneira muito distinta de outros objetos, diferente de outras linguagens que costumam deixar muito clara a diferença entre funções e variáveis.

Programar com um mindset funcional é aproveitar essa característica inerente da linguagem e usar e abusar de funções coesas, desacopladas, que aumentem o reuso de código, a

legibilidade do código, facilitem a criação de testes (que veremos mais pra frente) e muito mais. Algumas regras básicas que você pode adotar com JS “funcional” sem medo de errar são:

SRP - Single Responsibility Principle

Cada função deve ter uma única razão de existir e, portanto, uma única razão para ser alterada. Jamais crie funções que atuem como “canivetes suíços”. Isso garante uma alta coesão e é uma boa dica para criação de módulos em JS também, evitando módulos como “utils.js”.

DRY - Don't Repeat Yourself

Trechos de código com lógica parecida? Encapsule em uma função. Funções chamadas repetidas vezes? Encapsule em um laço. Funções repetidas entre projetos? Encapsule em um módulo. Etc.

Imutabilidade

Sempre que possível, trabalhe com objetos imutáveis. Isso se tornou bem mais fácil com a adição do ‘const’ no ES6.

Baixo Acoplamento

Lembra da definição matemática do que é uma função? Funções são relações de um conjunto A com um conjunto B, ou seja, dados determinados parâmetros de entrada (A), teremos determinado retorno como saída (B). Sempre relate A com B, jamais A com C (variáveis externas?) ou C com B, ou seja, para manter um baixo acoplamento a função apenas deve conhecer o conjunto de entradas para construir a partir dele as suas saídas.

Declarativo ao invés de Imperativo

Use as funções declarativas nativas como `foreach`, `map`, `reduce`, `filter`, etc trabalharem duro por você sem se preocupar com os detalhes de tais lógicas fica mais legível e muitas vezes mais eficiente.

Anonymous Functions

Aprenda a usar e se acostume com elas, especialmente closures e *arrow functions*.

Desde o ES6 que JS tem obtido características mais orientadas à objetos e alguns conceitos de OO inclusive foram citados logo acima. No entanto, focar em um JS OO, na minha opinião, não agrega tanto valor à linguagem quanto suas características funcionais.

Existem bons livros sobre programação funcional, inclusive com JavaScript, além de cursos específicos, então não entrarei em detalhes aqui. Apenas reservo este espaço para alertar sobre a importância desse mindset para o uso correto desta tecnologia e sugerir a leitura deste material simples, porém muito inteligente: <http://jrsinclair.com/articles/2016/gentle-introduction-to-functional-javascript-intro/>.

JS Full-stack

Muitas pessoas chegam até o Node com a promessa de escrever tanto o client-side quanto o server-side na mesma linguagem. No entanto, para que realmente isso seja vantajoso o reuso de software tem que ser possível em ambos os lados da aplicação, certo?!

Uma boa prática é conseguir reutilizar os seus módulos JS tanto no browser quanto no back-end (via Node.js).

Primeiramente, em Node quando queremos expor um módulo, usamos um código semelhante a esse:

Código 10.6: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
exports.test = function(){
    return 'hello world';
}
```

No entanto, no browser isso dá erro uma vez que exports é undefined neste ambiente. O primeiro passo para contornar este problema é verificar a existência ou não do exports, caso contrário, teremos que criar um objeto para o qual possamos exportar as funções:

Código 10.7: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
if(typeof exports === 'undefined'){
    var exports = this['mymodule'] = {};
}
```

O problema com essa abordagem é que no browser as funções que não foram exportadas também ficam disponíveis como funções globais, o que não é algo desejável. É possível resolver isso usando closures, como no exemplo abaixo:

Código 10.8: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
(function(exports){
    // seu código vai aqui
    exports.test = function(){
        return 'hello world'
    };
})(typeof exports === 'undefined'? this['mymodule']={}: exports);
```

O uso do objeto this representa o browser, e o uso de this['mymodule'] é o local de exportação no browser. Esse código está pronto para ser usado tanto no browser quanto no server. Considerando que ele está em um arquivo mymodule.js, usamos esse módulo da seguinte maneira em Node:

Código 10.9: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
var mymodule = require('./mymodule'),  
    sys = require('sys');  
  
sys.puts(mymodule.test());
```

E no browser usamos assim:

Código 10.10: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<script src="mymodule.js"></script>  
<script>  
    alert(mymodule.test());  
</script>
```

Claro, existem códigos JS escritos em Node que não serão suportados pelo browser, como o comando require por exemplo. Sendo assim, os módulos compartilhados entre os dois deverão ser o mais genéricos possíveis para que haja compatibilidade.

Tome cuidado também com as features mais recentes da linguagem Javascript que muitas vezes são suportadas pela engine V8 que o Node usa mas não pelos demais browsers e vice-versa (em casos mais raros). Nestes casos é muito útil usar um transpiler como o [Babel](#).

Boas Práticas de Testes

Parece um tanto óbvio e estúpido incluir um capítulo sobre testes dentro de um guia de boas práticas, certo?

Quem dera fosse.

A grande maioria dos projetos que vivenciei (e ouvi falar), independente da plataforma, não possuem uma boa cobertura de testes, não fazem testes unitários corretamente, não possuem rotinas de regressão, etc. E nem estou falando do quase utópico TDD.

Coloque em sua mente que não importa a “idade” do seu projeto, nunca é tarde para iniciar uma cultura de testes. E não, testes não prejudicam projetos. Bugs sim. Cliente insatisfeito sim. Comece pequeno, comece simples, no mínimo com as dicas que serão dadas neste capítulo.

Use uma boa ferramenta

Assim como o JavaScript tradicional, em Node podemos usar qualquer editor de texto para programar nossas aplicações. No entanto, opções simplificadas demais como o Notepad e editores de linha de comando (como nano), embora práticos de usar, não são muito úteis quando precisamos depurar aplicações "bugadas". Além disso, não contam com de outros benefícios como navegar entre funções, diversos tipos de highlights e autocompletes, etc.

Vou dar duas sugestões de como depurar programas Node.js.

Opção 1: troque seu editor de texto

O melhor jeito para mim atualmente é usando o [Visual Studio Code](#), que vem com suporte nativo a Node.js, é gratuito e roda em Windows, Mac e Linux. Nele você pode colocar breakpoints em

arquivos .js, inspecionar variáveis, testar expressões no console, fazer integração com Git, [StandardJS](#) e muito mais.

Outra alternativa, mais pesada é o [Visual Studio Community](#) (2015 com [plugin para Node](#) ou 2017 com suporte nativo).

Opção 2: use um depurador externo ao seu editor

Agora se você não quiser abrir mão de usar editores de texto nativos do seu SO ou os clássicos com os quais já está acostumado, você pode depurar seus programas Node.js diretamente no Google Chrome também, usando o F12. Você consegue mais informações [neste post do Medium](#).

E, por fim, uma outra alternativa para não largar seus editores de texto favoritos é usando o Node Inspector, um [projeto open-source disponível no GitHub](#).

Trate bem os seus erros

Não há nada mais agradável do que de repente a sua aplicação Node sair do ar e você descobrir no servidor que foi uma exception que fez isso. Aquele trecho de código mal testado, aquela resposta não prevista da API, etc. Não importa o que era, mas sim que derrubou tudo. E você não pode deixar isso acontecer.

Um bom gerenciamento de exceções é importante para qualquer aplicação, e a melhor maneira para lidar com erros é usar as ferramentas fornecidas pelo Node.js para isso, como em promises, em que temos o handler `.catch()`, que propaga todos os erros para serem tratados em um único local, como no exemplo abaixo:

Código 10.11: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
doSomething()
  .then(doNextStage)
  .then(recordTheWorkSoFar)
  .then(updateAnyInterestedParties)
```

```
.then(tidyUp)  
.catch(errorHandler)
```

Qualquer erro que aconteça nos then() será capturado para tratamento no catch().

Aumente a visibilidade dos erros

A dica aqui não é jogar erro para o usuário, mas sim usar uma biblioteca de logging para aumentar a visibilidade dos erros. O console.log é ótimo, mas tem sérias limitações em uma aplicação em produção. Tentar encontrar a causa de um bug dentre milhares de linhas de logs é terrível e uma biblioteca de logging madura pode ajudar com isso.

Primeiro, bibliotecas de logging permitem definir níveis para as mensagens como debug, info, warning ou error mesmo. Segundo, geralmente elas permitem quebrar os logs em diferentes arquivos ou mesmo persistir remotamente, o que, aliás, é uma ótima ideia na minha opinião.

Uma sugestão muito boa é mandar seus logs para a Loggly.com que tem um plano free bem interessante de 200MB de logs por dia e armazenamento dos últimos sete dias. O serviço dessa startup permite buscar rapidamente por mensagens usando padrões e definir alertas caso muitos erros estejam acontecendo.

Falando de bibliotecas especificamente, a recomendada aqui é a [winston](#).

Vendo os objetos completamente no console

Certas vezes quando temos objetos complexos em Node.js e queremos ver o que ele está guardando dentro de si usamos o console do Google Chrome ou mesmo do Visual Studio para entender o que se passa com nosso objeto. No entanto,

dependendo do quanto "profundo" é o nosso objeto (quantos objetos ele possui dentro de si), essa tarefa não é muito fácil.

Aqui vão algumas formas de imprimir no console o seu objeto JSON inteiro, não importando quantos níveis hierárquicos ele tenha:

Opção 1: `console.log`

Tente usar a função `console.log` passando o objeto or parâmetro, isso funciona na maioria dos casos.

Código 10.12: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
console.log(myObject);
```

Opção 2: `util.inspect`

Use o seguinte código abaixo para usar a função `util.inspect` e retornar todo o conteúdo de um objeto JSON.

Código 10.13: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
const util = require('util')
console.log(util.inspect(myObject, {showHidden: false, depth: null}))
```

Opção 3: `JSON.stringify`

Use a função `JSON.stringify` passando o objeto e o nível de indentação que deseja dentro do objeto, como abaixo.

Código 10.14: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
console.log(JSON.stringify(myObject, null, 4))
```

Escreva testes unitários

Escrever testes unitários permite que você tenha mais confiança que as menores partes do seu projeto funcionam como deveriam.

Além disso, resolve a maior parte dos bugs toscos que tiram muitas noites de sono dos desenvolvedores. Unit tests são facilmente automatizados através de extensões adequadas, oferecendo um ganho de qualidade e velocidade muito grande.

Existem diversos módulos para ajudar com testes unitários em Node.js e não há um consenso sobre qual o melhor deles, sendo os mais comumente recomendados:

- Jasmine
- Mocha
- Chai
- Tap

Especificamente para web applications e web APIs, usar o [Supertest](#) ou o [hippie](#) para abstrair a camada de front-end é uma ótima ideia também.

Não vou entrar em detalhes aqui, já que não sei qual biblioteca você vai adotar e todas possuem excelente documentação e tutoriais na Internet. Os excelentes livros do Kent Beck como TDD (ele é o inventor do termo) e até mesmo o livro Programação Extrema (seu método ágil de desenvolvimento de software) são altamente recomendados neste tópico.

Boas Práticas de Deploy

Então você usou os melhores princípios do Node.js, escreveu sua aplicação perfeitamente, depurou e tratou todos os erros, agora é só colocar em produção, certo?

'Só' é uma palavra meio fraca para esta tarefa que, dependendo da sua aplicação, pode exigir bastante trabalho ou ao menos bastante configuração de sua parte. Há ainda a questão de que apenas 'colocar em produção' não resolve todo seu problema, afinal você terá de ter mecanismos que mantenham a sua aplicação disponível, mecanismos de logging, etc.

Neste capítulo falaremos de várias coisas sobre colocar sua aplicação no ar e mantê-la lá!

Não use FTP!

Antes de entrar nas boas práticas do que você deve usar, falarei do que você não deve usar: FTP. Estamos falando de uma tecnologia da década de 80 que, embora ainda seja utilizada até hoje em diversos contextos, nunca foi criada com o objetivo de implantar software web.

Implantar software na web profissionalmente raramente é apenas subir arquivos para uma pasta no seu servidor. Geralmente envolve fazer um backup da versão atual em produção, subir a nova versão, substituir as duas, etc. No caso do Node.js, pode envolver também baixar os módulos corretos via NPM e esse é o tipo de coisa que você não vai querer subir via FTP.

Se você usa um provedor de hospedagem para manter suas aplicações Node.js 24x7, tenha em mente que FTP inevitavelmente falha e que isso não é bom para deploy em produção. Afinal, quem nunca ficou com um arquivo trancado em 0 bytes no servidor e você não consegue nem excluí-lo, bem sobrescrevê-lo?

Na [Umbler](#) a hospedagem Node.js não usa FTP, mas sim Git para deploy. Na nossa abordagem, tanto faz se você enviar a sua pasta node_modules ou não, e você pode também adicionar outros usuários com permissão para fazer deploy também, se necessário.

Cuide da segurança

Se o seu projeto é uma web application, existem um monte de boas práticas de segurança específicas visando manter sua app segura:

- Proteção a XSS;
- Prevenção de ClickingJacking usando X-Frame-Options;
- Forçar conexões como HTTPS;
- Configurar um cabeçalho Context-Security-Policy;
- Desabilitar o cabeçalho X-Powered-By para que os atacantes não tenham informações úteis para ataques;

Ao invés de tentar se lembrar de todas elas você pode usar o módulo Helmet, que permite facilmente alterar todas configurações default (e inseguras) de web applications e deixa você customizar as que forem necessárias.

Como tudo em Node.js, é incrivelmente simples de instalar o Helmet:

Código 10.15: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

\$ npm install helmet

e muito simples e usar na sua aplicação:

Código 10.16: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
var helmet = require('helmet');
app.use(helmet());
```

Mantendo seu processo no ar!

Se você optar por uma solução hands-on, em que você terá de configurar o servidor para manter sua aplicação rodando, certifique-se de fazer com que seu processo fique sempre rodando, mesmo que seja derrubado e mesmo que o servidor inteiro reinicie. Não importa se você tratou todos os erros da sua aplicação, inevitavelmente algo vai acontecer e tirar seu processo do ar e ele deve ser capaz de subir novamente.

PM2

PM2 é um projeto open-source criado e mantido pela empresa Keymetrics.io, que, além do PM2 (que é gratuito), vende um serviço de gerenciamento de performance de aplicações Node.js homônimo. Só para você ter uma ideia do que o PM2 é hoje, são mais de 20M de downloads e empresas como IBM, Microsoft e PayPal usando, o que o torna, disparado, a melhor solução de process manager pra Node, muito mais do que seus principais concorrentes, o Forever e o Nodemon.

Para usar o PM2 é muito simples, primeiro instale globalmente o módulo do PM2:

Código 10.17: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

\$ npm install pm2 -g

Depois, quando quiser iniciar o processo da sua aplicação Node.js:

Código 10.18: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

\$ pm2 start myApp.js

Para garantir que ele rodará eternamente, se recuperará sozinho de falhas, etc, você pode estudar o [guia oficial dele](#).

Além do PM2, a Keymetrics.io oferece uma ferramenta de monitoramento e alertas muito boa e útil que inclusive conta com um plano gratuito que atende boa parte dos desenvolvedores. Usar uma boa ferramenta de monitoramento ajuda bastante a evitar dores de cabeça principalmente se seu serviço possui clientes pagantes. Ao contrário de ferramentas de monitoramento de uptime simples como Pingdom e StatusCake, a Keymetrics.io monitora consumo de recursos como RAM e CPU.

Trabalhando com vários cores

O Node.js trabalha com uma única thread dentro de um único processo na sua máquina. Dessa forma, é natural que ele utilize apenas um processador, mesmo que você esteja rodando sua aplicação em um webserver com 16 núcleos ou mais. Sendo assim, uma dúvida bem comum é: como escalar um projeto Node para que use todo o poder do seu servidor?

Basicamente você tem algumas opções de como fazer o deploy de uma aplicação Node.js que use os diversos núcleos da máquina:

Usar uma arquitetura de micro-serviços.

Cada módulo da sua aplicação deve ser uma sub-aplicação autônoma, que responde a requisições e realiza apenas as tarefas que são de sua responsabilidade. Sendo assim, teríamos diversas aplicações pequenas escritas em Node.js, cada uma usando um core da sua máquina, recebendo (e processando) as requisições que cabem a elas. Uma aplicação principal recebe a requisição original do usuário e delega as tarefas para as demais sub-aplicações. Falei disso em mais detalhes no capítulo de Codificação.

Note que nesta abordagem usaremos vários cores pois teremos vários serviços Node rodando e, consequentemente, vários processos. Dentro da plataforma da [Umbler](#) você consegue adotar esta estratégia com vários subdomínios apontando para instâncias

pequenas ao invés de uma instância grande rodando um único app Node.js.

Usar um "webproxy" na frente do Node

Você pode colocar um Apache, Nginx ou IIS à frente da sua aplicação e deixar com ele essa tarefa de controlar a carga de requisições, balanceando entre diferentes nós idênticos da sua aplicação, cada um em um processador. Você pode fazer isso com Node.js também, mas geralmente Apache e cia. já possuem muito mais maturidade para isso.

Nesta abordagem usaremos vários cores rodando o mesmo serviço de maneira repetida. Tenha isto em mente quando estiver criando sua aplicação, pois cada processo irá fazer a sua conexão com o banco e não compartilhará memória com os demais. Uma alternativa para solucionar esse problema de memória compartilhada é usar um Redis ou similar para compartilhar recursos.

Além disso, ao invés de usar um webproxy à frente do Node você pode usar o PM2. Basicamente você consegue subir uma instância da sua aplicação Node por CPU usando o comando abaixo:

Código 10.19: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
$ pm2 start myApp.js -i max
```

Consulte o [guia oficial do PM2](#) para conhecer mais sobre clusterização com ele.

Existem outras opções? Sim, mas essas duas acima são recomendadas.

Use Integração/entrega/implantação contínua

Algumas definições rápidas pois existe muita confusão sobre estes termos:

Continuous Integration

É o processo de fundir o trabalho desenvolvido com a master várias vezes ao dia e/ou constantemente (dependendo do seu ritmo de trabalho), ajudando a capturar falhas rapidamente e evitar problemas de integração, sendo boa parte desses objetivos atingidos com uma boa bateria de testes automatizados.

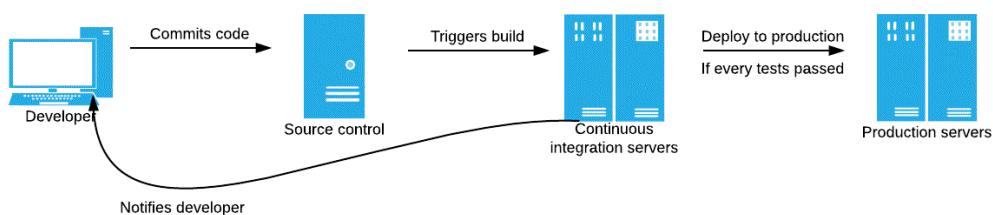
Continuous Delivery

É o próximo passo após a entrega contínua, sendo a prática de entregar código para um ambiente, seja para o time de QA ou para clientes, para que o mesmo seja revisado. Uma vez que as alterações sejam aprovadas, eles podem ir para produção.

Continuous Deployment

O último passo nessa sequência é a implantação contínua, quando o código que passou pelas etapas anteriores com sucesso é implantado em produção automaticamente. Continuous deployment depende pesadamente de uma infraestrutura organizada e automatizada.

A imagem abaixo, do site Rising Stack ilustra o processo:



Embora estas definições variem um pouco dependendo da linha de pensamento adotada pelos autores de livros e professores universitários, todas elas estão associadas a conceitos mais universais como ALM e Agile e até mesmo com necessidades bem específicas como as entregas contínuas do Scrum e do Lean Startup. Independente disso, têm se mostrado cada vez mais importantes, e valiosas, nos projetos atuais.

Felizmente, Node.js parece ter nascido para ser entregue continuamente pois possui diversas características (cortesia do JavaScript) propícias para tal como a modularização, a dinamicidade, a facilidade de uso como micro serviços, entre muitas outras.

Considero que existem cinco pilares a se terem em mente se o objetivo é alcançar um ambiente de entrega contínua realmente funcional:

Pilar 1: Versionamento

Seu código deve estar versionado em um repositório (geralmente Git), pois todo o processo de integração começa com um commit.

Pilar 2: Testes automatizados

O commit deve disparar (ou ao menos ser precedido, você escolhe) por uma bateria de testes automatizados, o que inclusive pode ser feito com as bibliotecas citadas no capítulo sobre testes.

Uma boa cobertura de testes é crucial se você quiser ser bem sucedido com entrega contínua, caso contrário a estratégia pode sair pela culatra, com bugs sendo entregues em produção mais rápido do que nunca!

Pilar 3: Chaveamento de features

Como os commits vão parar na master bem cedo, geralmente antes do final do projeto, é natural que existam features que devem esperar para serem lançadas e outras que devem ser lançadas

imediatamente. O chaveamento de features (*feature toggle* no original) também é muito útil quando se deseja subir correções pra master sem adicionar as funcionalidades novas nas quais você ainda está trabalhando e são uma alternativa para as dores de cabeça com *branches*. Existe inclusive um módulo no NPM chamado feature-toggles com esse objetivo.

Pilar 4: Ferramenta de CI

Existem diversas opções no mercado, tanto open-source quanto proprietárias, tanto pagas quanto gratuitas, fica a seu critério. A dica aqui é: não reinvente a roda. Existem excelentes opções consolidadas e bem testadas, apenas escolha uma e siga em frente.

Algumas opções são:

- Jenkins (open-source)
- Travis
- Codeship
- Strider (open-source)
- CircleCI

Basicamente o que essas ferramentas fazem são orquestrar builds complexos que são iniciados a partir de commits no seu repositório em uma branch específica. Esses builds podem incluir testes automatizados, execução de scripts, transferência de arquivos, notificações por email e muito mais. Tudo vai depender do que seu projeto precisa.

Pilar 5: Rollback

Sempre tenha em mente que o processo pode falhar, ainda mais em uma arquitetura complexa de deployment contínuo e que, se isso acontecer, tudo deve continuar como estava antes do processo de CI ter sido inicializado. Um rollback bem pensado e 100% funcional deve ser uma prioridade.

Com certeza não estressei esse assunto com estas poucas sugestões. Existem excelentes livros sobre ALM, CI, etc mas espero

ter despertado no mínimo o interesse pelo assunto caso seja novidade para você. Na [Umbler](#) foram facilitados esses processos através do deploy via Git e essa é uma das características mais apreciadas pelos clientes.

Referências

A cada capítulo, listarei referências onde você pode se aprofundar nos assuntos citados.

Publicação na Umbler

Série de videos sobre desenvolvimento Node.js que termina com a publicação de uma aplicação Node.js na Umbler.

<https://www.youtube.com/watch?v=gq9uGdZCKxI>

Publicação na AWS

Série de artigos sobre publicação da sua aplicação Node.js na Amazon AWS.

<https://www.luiztools.com.br/post/publicando-sua-aplicacao-node-js-no-amazon-lightsail-aws/>

Dúvidas técnicas

Respostas para as 8 dúvidas técnicas mais comuns em Node.js.

<https://www.luiztools.com.br/post/as-8-duvidas-tecnicas-mais-comuns-sobre-nodejs/>

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

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 a web 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 uma série de formas de criar aplicações com esta fantástica 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 ainda mais tutoriais bacanas, sugiro dar uma olhada em meu blog <https://www.luiztools.com.br>.

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!

Caso tenha gostado do material, indique esse livro a um amigo que também deseja aprender a programar para web 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!

Curtiu o Livro?

Por favor, deixe a sua review sincera sobre o livro acessando o link abaixo, isso ajuda muito este humilde autor:

https://www.amazon.com.br/review/create-review/ref=cm_cr_dp_d_wr_but_top?ie=UTF8&channel=glance-detail&asin=B074RCRKSL

Aproveita e me segue nas redes sociais: <https://about.me/luiztools>

Conheça meus outros livros:

[Criando apps para empresas com Android](#)
[Scrum e Métodos Ágeis: Um Guia Prático](#)
[Agile Coaching: Um Guia Prático](#)
[Java para Iniciantes](#)
[Node.js e Microservices: Um Guia Prático](#)
[Programação Web com Node.js: Edição MySQL](#)
[MongoDB para Iniciantes](#)

Conheça meus cursos online:

[Curso de Scrum e Métodos Ágeis](#)
[Curso de Node.js e MongoDB](#)
[Curso de Web FullStack JS](#)
[Curso de React Native e Google Firebase](#)
[Curso de Gestão de Backlog com Jira](#)

Apêndice 1: Módulos Recomendados

Uma das coisas mais fantásticas do Node.js é o seu ecossistema. O NPM rapidamente cresceu para se tornar o maior repositório de bibliotecas e extensões do mundo e isso em menos de uma década de existência.

Esse capítulo, na verdade, é um índice remissivo para que você encontre facilmente todos os módulos recomendados nos demais capítulos, apenas por uma questão de organização, em ordem alfabética e com uma frase resumindo do que se trata. Tomei a liberdade de incluir também outras bibliotecas notáveis que não foram citadas, mas que acredito serem importantes.

Clique nos links dos nomes para ir ao site de cada módulo.

[Artillery.io](#)

Módulo muito bacana para testes de carga e stress em APIs, não necessariamente Node.

[Babel](#)

Transpilador muito utilizado para permitir a utilização de recursos recentes do EcmaScript mesmo em navegadores mais antigos.

[Chai](#)

Popular biblioteca para uso de TDD e BDD com Node.js.

[DotEnv](#) e [DotEnv-Safe](#)

Módulos para carregamento de variáveis de ambiente de maneira rápida e fácil. A diferença entre os dois pacotes é que a versão ‘safe’ não permite a inicialização do sistema se as variáveis de ambiente necessárias não puderam ser encontradas.

EJS (Embedded JavaScript)

View-engine muito popular para usa com Express visando permitir o uso de HTML e JS para composição das views.

Express

Web framework extremamente popular para Node.js. Se você não conhece, deveria, resolve boa parte das demandas com Node de maneira rápida e fácil.

Express Generator

Um scaffold para ExpressJS que já define uma arquitetura básica mas bem útil para iniciar rapidamente com projetos Node.js.

feature-toggles

Extensão para permitir facilmente fazer...feature-toggles.

Helmet

Módulo de segurança que já blinda a sua aplicação Express quanto aos ataques e brechas mais comuns em web applications.

Hippie

Extensão para permitir o teste de APIs web abstraindo boa parte das complicações.

Jasmine

Popular biblioteca para testes.

Loopback.io

Framework web muito popular para criação de APIs Restful.

Mocha

Popular biblioteca para testes.

Mongoose

ORM bem popular e poderoso, especialmente para uso com MongoDB, até 50% mais veloz do que seu principal concorrente,

Monk.

[Node-Windows](#)

Biblioteca para uso de scripts Node.js como Windows Service em servidores.

[Passport](#)

Autenticação em aplicações Node.js, especialmente APIs.

[PM2](#)

Gerenciador de processos para Node.js, visando recuperação de falhas, monitoramento e muito mais.

[Sequelize](#)

ORM muito popular e indicado para uso de Node.js com bancos relacionais.

[Socket.io](#)

Comunicação ponto-a-ponto para aplicações de mensageria em Node.js.

[StandardJS](#)

Linter JavaScript que determina um guia de estilo uniforme e popular para codificação de projetos JS.

[Supertest](#)

Extensão para permitir o teste de aplicações web abstraindo o front-end.

[Tap](#)

Popular biblioteca para testes.

[Typescript](#)

Um superset que permite escrever em um JavaScript mais ‘bombado’ que depois é compilado para JavaScript tradicional para uso em produção.

Winston

Popular (e poderosa) biblioteca para logging.

Tem algum módulo que você conhece e recomenda e que não está nessa lista? Mande para mim pelo contato@luiztools.com.br que incluirei em revisões futuras deste livro!

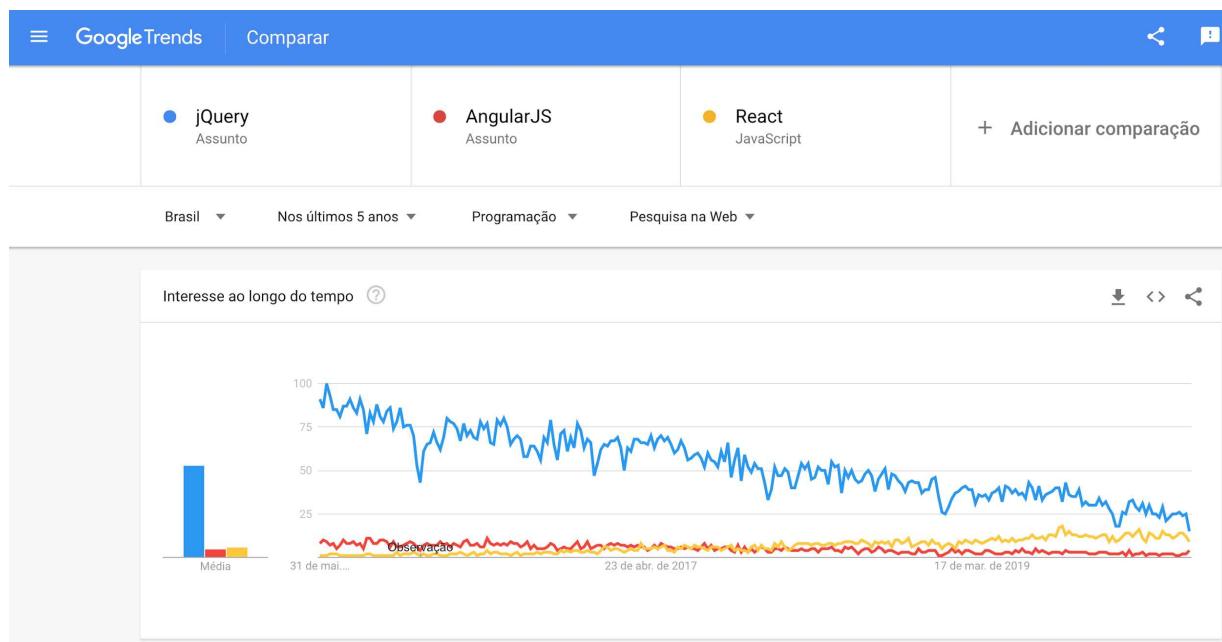
Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>

Apêndice 2: jQuery

“The very existence of libraries affords the best evidence that we may yet have hope for the future of man”
— T.S. Eliot

jQuery é uma biblioteca JavaScript cross-browser desenvolvida para simplificar os scripts client side que interagem com o HTML. Ela foi lançada em dezembro de 2006 no BarCamp de Nova York por John Resig. Usada por cerca de 73% dos 10 milhões sites mais visitados do mundo, jQuery é a mais popular das bibliotecas JavaScript.

Apesar de nos últimos anos ter caído em popularidade, ainda encontra-se acima de outros frameworks JavaScript muito populares atualmente como Angular e React, por isso ainda entendo que vale a pena ser estudado, como mostra o gráfico de popularidade do Google Trends dos últimos 5 anos (em 2020):



JQuery define uma série de seletores para facilitar a busca por componentes dentro do documento, bem como função prontas para manipulá-los, principalmente de maneira assíncrona. Além disso,

uma vez que tornou-se padrão de mercado, existem diversos plug-ins que extendem suas capacidade para proporcionar efeitos cada vez mais incríveis, indo desde máscaras de campos à transições de páginas, slideshows, jogos e muito mais.

Toda a especificação completa, bem como download da biblioteca pode ser feito no site oficial: <http://jquery.com>

Como usar

Primeiramente você terá de entrar no site JQuery.com e baixar a versão mais recente da biblioteca para o seu computador. Dê preferência à versão mais recente (3.5.1 na época que escrevo este livro) e na distribuição "minificada" para ter o máximo de performance e a menor carga possível. A versão que estou usando pode ser baixada neste link:

<https://code.jquery.com/jquery-3.5.1.min.js>

No documento HTML (ou EJS no caso do Node) em que deseja ter acesso aos recursos do JQuery você deve inserir uma tag SCRIPT apontando o atributo src para seu arquivo JS do JQuery, como abaixo (atenção ao nome do arquivo, eu removi a versão).

Código a2.20: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

<script src="js/jquery.min.js"></script>

No exemplo acima o meu arquivo jquery.min.js está dentro da pasta js na raiz do site.

Nota: lembre-se que a tag SCRIPT pode ser usada para referenciar um arquivo JS ou como um contâiner para scripts JS inline. Você pode ter mais de uma tag SCRIPT na sua página HTML, mas não usá-lo de mais de uma maneira (referenciar um arquivo JS e colocar código JS no mesmo SCRIPT, por exemplo).

Seletores

O primeiro e mais fundamental conceito do JQuery é o de seletores. Sempre que queremos lidar com um componente HTML, devemos primeiro carregá-lo em nosso script. Para fazer isso, geralmente os programadores têm de definir um id no elemento e depois carregá-lo usando document.getElementById. No JQuery é muito mais simples e muito mais poderoso ao mesmo tempo.

Considere o seguinte trecho HTML:

Código a2.21: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<!DOCTYPE html>
<html>
  <head><meta charset="utf-8" /></head>
  <body>
    <input type="text" name="nome" id="txtNome" />
  </body>
</html>
```

Em JQuery usamos o ‘\$(filter)’ como função de seleção, sendo que devemos passar por parâmetro o filtro a ser utilizado. Este filtro pode ser (usando o HTML acima como exemplo):

- um id, prefixado com susterido: \$("#txtNome");
- tipo de tag, sem prefixo algum: \$("input");
- um atributo definido após a tag, entre colchetes: \$("input[name='nome']")
- começo de um atributo, usando circunflexo: \$("input[name^='nom']")
- final de um atributo, usando cifrão: \$("input[name\$='ome']")
- um elemento dentro de outro elemento: \$("body > input")
- mais de um elemento ao mesmo tempo: \$("head,body")

Todos os seletores exemplificados acima carregam o elemento input presente no HTML de exemplo.

O uso da função de seleção irá retornar um objeto JS contendo o elemento HTML e todas suas propriedades e funções. Além disso, a busca é feita em todo o documento, logo, se mais de um elemento atender ao filtro, todos serão retornados, motivo pelo qual devemos sempre buscar utilizar filtros específicos, como por id, por exemplo.

Uma vez que selecionamos o elemento que queremos, podemos manipulá-lo usando os atributos e funções nativos do JS ou usando as funções próprias do JQuery, citadas a seguir.

Funções de Elemento

Para usar as funções abaixo, você deve estar manipulando um elemento HTML que acabou de encontrar com o seletor do JQuery.

element.show(), hide() e toggle()

Exibe, esconde ou alterna o elemento HTML (equivalente a display:block e none).

element.html([value])

Obtém ou altera o HTML desse elemento.

element.val([value])

Obtém ou atribui o value deste elemento HTML.

element.text([text])

Obtém ou atribui o texto interno deste elemento HTML.

element.attr(attributeName[, value])

Obtém o valor do atributo especificado por parâmetro ou até mesmo altera seu valor passando um segundo parâmetro.

element.append(html)

Anexa o conteúdo HTML à este elemento HTML. Comumente utilizado para adicionar options em selects dinâmicos.

element.empty()

Elimina todos elementos internos à este.

element.focus()

Define o foco do usuário para o elemento HTML em questão.

element.remove()

Remove este elemento da página.

Funções de Coleções

Para usar as funções abaixo, o seletor JQuery deve ter retornado uma coleção de elementos.

colecao.size()

Obtém a quantidade de elementos retornados pelo seletor.

colecao.each(function(index, value))

Permite iterar sobre uma coleção de elementos, sendo que a cada iteração a função passada por parâmetro receberá o índice e o valor do elemento daquela iteração,

Eventos

Para usar os eventos abaixo, você deve primeiro carregar o elemento (ou documento) ao qual deseja associar o evento. Note que a maioria destes eventos são os mesmos eventos JS que poderíamos configurar no próprio elemento HTML, como vimos no tópico anterior. Use um ou outro, sendo que a proposta abaixo usando JQuery ajuda a tornar o seu HTML mais legível e menor.

\$.ready(function)

O ready é um evento disparado quando a página HTML está pronta para ser manipulada via JS. Ou seja, assim que o carregamento de elementos da página terminar, a função passada por parâmetro ao \$.ready será disparada.

element.click(function) e dblclick(function)

Define uma função que será disparado quando este element for clicado uma ou duas vezes. Mesmo efeito do atributo onclick.

element.change(function)

Define uma função que será disparada quando este element for alterado (selects principalmente). Mesmo efeito do atributo onchange.

element.focus(function) e blur(function)

Define uma função que será disparada quando este elemento ganhar/perder o foco do usuário (cursor). Mesmo efeito dos atributos onfocus e onblur.

element.keypress(function(event)), keydown e keyup

Define funções para manipulação de teclas pressionados quando o foco está sobre o elemento em questão. Mesmo efeito dos atributos onkeypress, onkeydown e onkeyup.

element.on(evento, function)

Adiciona um gatilho no elemento que irá disparar uma função toda vez que o evento especificado acontecer. Ex: click, change, etc.

Qualquer evento pode ser associado desta forma.

Todos os demais eventos podem ser vistos no site oficial:
<https://api.jquery.com/category/events/>

Exercitando

Vamos exercitar o que vimos até aqui de JQuery criando um exercício que mais tarde evoluíremos com outra tecnologia importantíssima chamada Ajax.

Crie uma nova pasta para guardar os arquivos deste projeto que chamaremos de exemplojquery. Nesta pasta, crie uma subpasta chamada js e coloque dentro dela o arquivo do JQuery, que você baixa no site oficial do jquery.com. Na raiz do projeto crie um arquivo HTML padrão chamado index.html, com HEAD e BODY, e com uma tag SCRIPT no BODY referenciando o script JQuery que está na pasta js, como abaixo:

Código a2.22: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<!DOCTYPE html>
<html>
  <head>
    <title>Exemplo JQuery</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Exemplo JQuery</h1>
    <p>Sistema para exemplificar uso de JQuery</p>
    <script src="js/jquery.min.js"></script>
  </body>
</html>
```

Nota: é possível na tag SCRIPT referenciar o arquivo JQuery diretamente do site oficial, passando o caminho completo do mesmo. Ex: <http://jquery.com/jquery-3.5.1.min.js>.

Agora coloque duas DIVs no BODY deste HTML, uma com id 'divCadastro' e outra com id 'divListagem'. A ideia aqui é construir

um único HTML que contenha tanto um formulário de cadastro quanto a listagem dos clientes cadastrados. Tornaremos essa tela dinâmica apenas usando JQuery, sem qualquer tipo de backend ou banco de dados.

Código a2.23: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div id="divCadastro">
  <h2>Cadastro</h2>
</div>
<div id="divListagem">
  <h2>Listagem</h2>
</div>
```

Note que se você abrir este arquivo HTML no seu navegador verá as duas DIVs visíveis, mas não é o que queremos, certo? De alguma forma temos que garantir que apenas uma delas esteja visível de cada vez.

Para fazer isso, vamos adicionar botões de navegação em ambas DIVs, como abaixo, visando ocultar/exibir as DIVs de acordo com cada botão pressionado:

Código a2.24: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<div id="divCadastro">
  <h2>Cadastro</h2>
  <input type="button" id="btnListar" value="Listar">
</div>
<div id="divListagem">
  <h2>Listagem</h2>
  <input type="button" id="btnCadastrar" value="Cadastrar">
</div>
```

Mas e como podemos programar esses botões? Uma das maneiras é com JQuery, que é o que queremos exercitar aqui.

Para não ficarmos escrevendo JavaScript no meio do HTML, o que é considerado uma má prática, crie um arquivo scripts.js na sua subpasta js e refencie-o na index.html logo abaixo do JQuery:

Código a2.25: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<script src="js/jquery.min.js"></script>
<script src="js/scripts.js"></script>
```

Nota: a tag SCRIPT exige que ela seja fechada com uma /SCRIPT, jamais fechada nela mesma. Se você não respeitar essa regra, sua página não irá funcionar.

A ordem que carregamos os arquivos JS importa pois a página HTML é lida de cima para baixa, e como vamos usar recursos do JQuery nesse arquivos scripts.js, ele deve vir após.

Agora abra seu scripts.js e vamos definir o evento ready do JQuery, que é disparado assim que os componentes da página estão todos prontos para serem manipulados:

Código a2.26: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
$(document).ready(function(){
})
```

Note que usamos o seletor sobre o objeto document, que representa a página HTML atual, e sobre ele definimos o evento ready que irá disparar a função de callback (vazia no momento) assim que a página estiver pronta para manipulação.

O que vamos fazer aqui? Primeiro, esconder a divListagem, pois queremos que o HTML inicie exibindo somente a divCadastro. Fazemos isso facilmente em JQuery selecionando o elemento que queremos esconder e depois chamando a função hide() nele, que serve para esconder elementos HTML:

Código a2.27: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
$(document).ready(function(){
    $('#divListagem').hide();
})
```

Implementando este código você notará que quando abrir seu arquivo index.html no navegador, ele exibirá somente uma das DIVs, mas que o botão dela não funciona ainda.

Para fazê-lo funcionar é simples, precisamos manipular o evento click deles! Ainda dentro do evento ready do seu scripts.js, inclua os seguintes eventos nos botões (note que sempre começamos selecionando o componente que vamos manipular):

Código a2.28: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
$(document).ready(function(){
    $('#divListagem').hide();

    $('#btnListar').click(function(){
        $('#divListagem').show();
        $('#divCadastro').hide();
    })

    $('#btnCadastrar').click(function(){
        $('#divListagem').hide();
        $('#divCadastro').show();
    })
})
```

```
    })  
})
```

Teste agora e verá que é perfeitamente possível manipular a exibição das divs usando os botões.

Não sei o que você achou do código acima, mas uma coisa muito bacana do JQuery é que existem muitas formas de otimizar os códigos JS que escrevemos com ele. Por exemplo, o mesmo funcionamento do clique dos botões pode ser otimizado para isso aqui:

Código a2.29: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
$(document).ready(function(){  
    $('#divListagem').hide();  
  
    $('#btnListar,#btnCadastrar').click(function(){  
        $('#divListagem,#divCadastro').toggle();  
    })  
})
```

Sim, o código acima, que é bem menor, faz a mesma coisa do anterior. Isso porque o recurso de seletores do JQuery é muito poderoso e permite que utilizemos mais de um filtro, separados por vírgulas e combinados durante a seleção. Sendo assim, quando chamo a função click para definir o callback desse evento, ele aplica-se a todos elementos retornados pelo seletor 'composto'.

Mas isso não resolve todo o mistério desse código menor que faz a mesma coisa, certo? Como ficou a questão do hide x show? A função toggle aplica um 'hide' em quem 'está show' e vice-versa, ou seja, oculta se estiver visível, e exibe se estiver oculto.

Esse é o poder do JQuery!

Com esses conceitos e conhecimentos dominados, podemos avançar no nosso exercícios. Agora vamos criar um formulário de cadastro de cliente (nome, idade e UF) na primeira div e uma tabela de clientes na segunda. Sim, já fizemos algo parecido antes, você pode aproveitar a 'base', mas será ligeiramente diferente desta vez.

Dentro da divCadastro, segue o formulário:

Código a2.30: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<form action="" method=""><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>
    <input type="button" id="btnListar" value="Listar" | <input
    type="submit" value="Salvar" />
</p>
</form>
```

Tomei a liberdade de trocar o btnListar de lugar, colocando-o dentro do form também. Note que este form não possui action nem method,

pois não vamos enviá-lo ao servidor, vamos fazer coisas bem mais legais. :)

Já a divListagem, segue o HTML também:

Código a2.31: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<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>Ações</td>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td colspan="4">Nenhum cliente cadastrado.</td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <td colspan="4">
        <input type="button" id="btnCadastrar" value="Cadastrar">
      </td>
    </tr>
  </tfoot>
</table>
```

Como ainda não vimos como estilizar nossas páginas HTML, ignore os atributos style que usei e não se importe se a aparência de tudo ficar muito feia, como nas imagens abaixo:

Exemplo JQuery

Sistema para exemplificar uso de JQuery

Cadastro

Nome:

Idade:

UF: RS

|

E nessa outra aqui:

Exemplo JQuery

Sistema para exemplificar uso de JQuery

Listagem

Nome	Idade	UF	Ações
------	-------	----	-------

Nenhum cliente cadastrado.

Agora vamos programar o nosso formulário usando JQuery!

Como não temos back-end nesse exercício, quando o FORM for submetido, vamos capturar os dados enviados usando JQuery e usá-los para preencher a tabela com a lista de clientes. Claro, esse armazenamento dos cadastros será efêmero, e toda vez que atualizarmos essa página no navegador ela virá zerada.

No entanto, servirá para nosso intuito de exercitar o que vimos de JQuery. Como já estamos fazendo, aliás!

Para programar a submissão do form, vamos novamente abrir nosso scripts.js e inserir novos códigos dentro do evento ready do document (logo abaixo daqueles scripts de clique dos botões das divs).

Código a2.32: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
$('form').submit(function(event){  
    const data = $(this).serializeArray();  
  
    let linha = " ";  
    data.forEach(item => linha += '<td>' + item.value + '</td>');  
  
    if($('table > tbody > tr > td').length === 1)//se tem apenas uma  
    TD, é a default  
    $('table > tbody').empty();  
  
    $('table > tbody').append('<tr>' + linha + '<td><input  
    type="button" value="X" /></td></tr>');  
    $('#divListagem,#divCadastro').toggle();  
    event.preventDefault();  
})
```

Esse código ficou extenso, mas vou explicar.

Primeiro, o evento submit do form é disparado toda vez que o form é submetido, o que no nosso caso acontece apenas quando o input de submit é pressionado. Para evitar o refresh na página, comportamento natural de submissões de formulário, coloquei um event.preventDefault() ao final da função de callback, para cancelar a "submissão de verdade".

Para pegar os dados que foram submetidos temos duas opções: serialize() e serializeArray() no form que foi submetido. O serialize() retorna uma string no formato x-www-form-urlencoded (chave1=valor1&chave2=valor2, etc) enquanto que serializeArray() retorna um array de objetos JSON name/value, que foi o que optei.

Usei um forEach no array de name/value para montar o HTML da linha que precisa ser inserida na tabela. Mas antes de inserir, fiz um teste para ver quantas TDs (células) existiam dentro da tr de tbody. Se existir só uma, é porque é aquela default (não existem cadastros...) e devemos removê-la com a função empty(), que zera o conteúdo de um elemento.

Para finalizar, usei a função append no tbody para anexar a nova linha e troquei a visibilidade das divs para que seja possível ver o cliente recém cadastrado. Note que em diversos momentos usei seletores com o formato 'element > child', onde consigo descer níveis dentro da hierarquia de elementos HTML.

Se você testar agora, verá que funciona perfeitamente.

Mas...e aquele botão de exclusão que eu coloquei na última coluna da TD e que não funciona ainda?

Adicione um último código dentro do 'ready', que fará com que todos os botões de exclusão da tabela removam a própria linha onde estão (adicionei uma confirmação JavaScript também, só pra evitar cliques acidentais):

Código a2.33: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
$('table').on('click','input[value="X"]', function(){
    if(confirm('Tem certeza que deseja excluir este cliente?'))
        $(this).closest('tr').remove();
})
```

Basicamente selecionei a única tabela que temos e usei o 'on' para fazer um binding de 'click' real-time, diferente da função click() que só funciona com elementos que já existam no documento HTML. O 'on' deve ser aplicado sobre a tabela e espera o evento, o seletor do componente que vai receber este evento e a função do evento.

Atenção: como os botões de excluir são adicionados depois que a tabela já existia, não funciona deixar um código de click pronto para eles em seu arquivo JS, por isso que usei o 'on' aqui, que é aplicado mesmo nas mudanças futuras de elementos na página.

Falando da função dentro do click, ela pega o botão que foi clicado e procura a TR mais próxima dele (closest), removendo-a (remove). Bem simples.

Exemplo JQuery

Sistema para exemplificar uso de JQuery

Listagem

Nome	Idade	UF	Ações
Luiz	29	RS	<input type="button" value="x"/>

O funcionamento é muito legal na minha opinião. Uma aplicação simples e responsiva para o usuário, muito rápida e eficiente, sem aqueles refreshs de tela, redirecionamentos, etc.

Mas como podemos manter esse mesmo nível de experiência agradável tendo um back-end com banco de dados e tudo mais?

Com Ajax.

Ajax com jQuery

Uma biblioteca que dá um bom endereçamento ao problema do Ajax e que tem grande aceitação no mercado são as funções de Ajax do jQuery, que iremos conhecer a seguir.

Como pode presumir, para utilizar-se dos recursos de Ajax presentes no framework jQuery, primeiro teremos de referenciá-lo em nosso código HTML, como já fizemos em lições anteriores. Para isso usamos a tag script, como abaixo:

Código a2.34: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
<script src="js/jquery.min.js"></script>
```

Uma vez com o JQuery carregado na página, temos acesso a uma série de funções que nos permitem fazer chamadas assíncronas, entre elas, a mais importante que é a função

```
$.ajax(url[, settings]);
```

Esta função recebe geralmente dois parâmetros. O primeiro é a URL para onde essa requisição será enviada. O segundo atributo é um objeto JSON com as configurações da requisição Ajax. O exemplo abaixo mostra um objeto JSON com as principais configurações possíveis para usar como documentação. Nenhuma configuração é obrigatória.

```
var settings = {
    async: true, //diz se a requisição será assíncrona ou não
    contentType: 'application/json', //diz o tipo de conteúdo da requisição
    contents: 'application/json', //diz o tipo de conteúdo da resposta
```

```
data: null, //o corpo da requisição  
method: 'GET', //o método de envio da requisição  
error: function(){ alert('erro'); }, //call-back function de erro  
success: function(data){ alert('sucesso'); }, //callback function  
de sucesso,  
}
```

Assim, podemos fazer uma requisição Ajax a uma API local (como a que criamos no capítulo sobre Web APIs) usando o seguinte código:

Código a2.35: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
$.ajax('/clientes', settings);
```

Caso a requisição tenha sucesso executará a function na setting de success ou caso dê erro, executará a function de error. O objeto data recebido por parâmetro no callback de sucesso é o conteúdo da resposta do servidor, quando houver.

A documentação completa da função Ajax pode ser obtida em: <http://api.jquery.com/jquery.ajax/>

- No exemplo de settings acima foi informado que o tipo de conteúdo da requisição é [application/json], que é um formato muito popular na Internet. O padrão para esta informação é [application/x-www-form-urlencoded] que é aquele formato de chave-valor que é enviado por padrão quando submetemos um form ao servidor. Para que possamos enviar JSON temos que construir os objetos antes de enviá-los, passando o objeto JSON na setting data. Agora para enviarmos campos de formulário em uma requisição Ajax usamos a function serialize() logo após selecionarmos o form que queremos enviar, como a seguir.

Código a2.36: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
var data = $('#formCadastro').serialize();  
settings.data = data;
```

A função `serialize()` vai coletar os names e values dos campos para formar a string de chave-valor necessária. Note que a escolha entre o uso de JSON ou não vai depender das necessidades de front-end do seu projeto, e até o uso de Ajax pode ser considerado dispensável em alguns casos.

Funções Auxiliares

A função Ajax do JQuery, tal qual como demonstrada acima é utilizada quando queremos ter um controle maior das configurações das nossas requisições assíncronas. Quando queremos apenas realizar GETs e POSTs assíncronos de maneira rápida e fácil, existem soluções melhores dentro do próprio JQuery.

```
$.get(url, callback(data));
```

A função `$.get` faz um GET assíncrono no servidor, disparando a function `callback` que você definir como o data retornado pelo servidor como parâmetro. O conteúdo desse data vai depender do que foi enviado pelo servidor, podendo ser texto plano, HTML, XML, JSON, etc.

```
$.post(url, data, callback(data));
```

A função `$.post` faz um POST assíncrono no servidor, disparando a function `callback` que você definir com o data retornado pelo servidor como parâmetro. O conteúdo desse data (e até mesmo sua existência) vai depender do retorno do servidor, uma vez que em um POST nem sempre temos um corpo de retorno.

```
$.getJSON(url, callback(jsonData));
```

Semelhante ao `$.get`, porém aqui o retorno será um array de objetos JSON ou um objeto sozinho, dependendo do seu servidor. Em cima do objeto `jsonData` geralmente aplicamos um `$.each` para realizar algum processamento, como mostrado no exemplo abaixo onde adicionamos `options` em um `select` após pegar os estados do banco de dados.

Código a2.37: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
$.getJSON('/clientes', function(jsonData){  
    jsonData.each(function(key,val){  
        $('#estados').append('<option>' + val + '</option>');  
    });  
})  
  
element.load(url);
```

Semelhante ao `$.get` também, mas aqui o objeto é carregar conteúdo HTML de um fonte no servidor diretamente para dentro (como no `append`) de um elemento selecionado via JQuery anteriormente. Ou seja, primeiro usamos um seletor JQuery para carregar o componente que vai carregar o HTML e depois fazemos um GET no servidor buscando o HTML que queremos inserir em nosso componente, como no exemplo abaixo onde carregamos as `options` de estado de nosso `select` usando HTML fornecido pelo servidor.

Código a2.38: disponível em <https://www.luiztools.com.br/livro-nodejs-fontes>

```
$('#estados').load('/estados');
```

Ajax na prática com JQuery e Node.js

Note que todas as funções Ajax só fazem sentido quando temos um backend pronto para atendê-las.

Vimos em capítulos anteriores como fazer aplicações web que possuem formulários. Esses formulários quando são submetidos enviam a página inteira HTML para o servidor e exigem a renderização completa do HTML novamente, caracterizado por uma 'piscada' na tela do navegador. Entretanto, nem sempre queremos submeter todos os dados de um formulário ao servidor e muitas vezes nem mesmo queremos enviar a página em si ao servidor, apenas obter dados de lá, por exemplo.

Nestes casos, é útil termos uma Web API no backend que responderá à chamadas Ajax, principalmente aquelas usando o verbo GET, para poder carregar dinamicamente trechos da tela como opções de um select ou divs dinâmicas. Web APIs como essas que estou mencionando já foram ensinadas em capítulo anterior e, se você acompanhou os exercícios práticos, possui uma pronta agora que manipula dados de clientes no banco MongoDB.

Atenção: parto do pressuposto que você possui um banco de dados MongoDB rodando na sua máquina neste exato momento, tal qual foi ensinado no capítulo de Mongo. Também considero que sua WebAPI em Node.js está pronta e rodando em localhost:3000. E por fim, que você fez o exercício prático anterior, de JQuery, e possui a index.html com o formulário de cadastro e listagem pronto.

Voltemos ao projeto exemplojquery, que criamos na seção anterior. Nele, temos apenas uma página index.html que serve para cadastrar e listar clientes, alternando entre divs e com todo comportamento de front-end pronto. O que vamos fazer agora é conectá-la ao nosso back-end Node.js através de requisições HTTP Ajax.

Como mandam as boas práticas, nosso index.html não possui código JavaScript algum, sendo apenas uma view estática. Em nossa pasta js guardamos os arquivos JS que tornam essa página

dinâmica, basicamente a biblioteca JQuery e um arquivo scripts.js com nossos scripts personalizados. Vamos abrir este último.

No scripts.js temos um evento ready do document (nossa página HTML) que define alguns comportamentos e eventos aos componentes do documento usando seletores do JQuery. Essa forma de manipular componentes do HTML como se fosse objetos chama-se DOM ou Document-Object Model.

Um desses eventos é o submit do form, que hoje apenas adiciona uma nova linha na tabela de listagem. Além desse comportamento, queremos que ele poste os dados do formulário para nossa web API, para que o cliente seja salvo no banco também. Antes de aumentarmos a quantidade de código neste submit, vamos organizá-lo melhor.

Crie uma função em scripts.js, fora do ready, para encapsular toda a lógica de adicionar a linha na tabela de listagem, como abaixo:

Código a2.39: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function updateTable(data){
    let linha = "";
    data.forEach(item => linha += '<td>' + item.value + '</td>');

    if($('table > tbody > tr > td').length === 1)//se tem apenas uma TD, é a default
        $('table > tbody').empty();

    $('table > tbody').append('<tr>' + linha + '<td><input type="button" value="X" /></td></tr>');
    $('#divListagem,#divCadastro').toggle();
}
```

Enquanto que nosso evento submit vai ficar assim:

Código a2.40: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
$('form').submit(function(event){  
    const data = $(this).serializeArray();  
  
    updateTable(data);  
  
    event.preventDefault();  
})
```

Agora, vamos criar outra função, que postará via Ajax os dados do formulário, em formato JSON, para a nossa webapi Node.js:

Código a2.41: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
const webApiDomain = 'http://localhost:3000'  
function updateDatabase(data, callback){  
    const json = {};  
    data.forEach(item => json[item['name']] = item['value']);  
  
    $.post(webApiDomain + '/clientes', json, function(result){  
        alert('Cliente cadastrado com sucesso!');  
        callback(data);  
    })  
}
```

Nota: repare como coloquei o domínio da web API em uma constante separada. Isso porque essa informação será usada mais de uma vez e, se um dia você publicar isso em um webhosting, você deverá mudar localhost:3000 para o domínio final da sua API, o que fica muito mais fácil se essa informação estiver em um único lugar.

Essa função pega o array de name-values do formulário e os transforma em um objeto JSON, pois é isso que nossa API espera. Com esse objeto JSON, usamos o \$.post que faz um POST Ajax na URL passada por parâmetro, com o JSON no body e a

function(data) como callback. Bem simples a essa altura do campeonato.

E vamos chamar essa função dentro do evento de submit do form também, ficando assim:

Código a2.42: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
$('form').submit(function(event){  
    const data = $(this).serializeArray();  
    updateDatabase(data, updateTable);  
    event.preventDefault();  
})
```

Nota: repare como passei o updateTable como callback da updateDatabase, para que somente seja atualizado o HTML depois que o servidor indicar que fez a inserção.

Isso teoricamente seria o suficiente para que o cadastro passasse a funcionar, enviando os dados ao back-end. No entanto, a vida não é tão fácil assim. Se você abrir a index.html no navegador e testar o formulário, aparentemente nada acontecerá, mas se usar o F12 do Chrome (Developer Tools), você deve encontrar na aba console um erro como abaixo:

XMLHttpRequest cannot load http://localhost:3000/clientes. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access.

Se a sua página HTML e sua web api estiverem rodando no mesmo projeto, você não terá esse problema. Mas aqui como estamos rodando a index.html em um projeto separado da web API, existe uma regra de segurança dos browsers que proíbe chamadas Ajax de um domínio serem enviadas para outro, a menos que o outro domínio permita isso. Essa regra se chama de Same Origin Policy

(Política da Mesma Origem) e a melhor maneira de driblarmos ela é usando CORS.

Já vimos isso no capítulo 8, retorne lá se não se lembra como habilitar CORS na sua web API Node.

Porém, mesmo com nosso cadastro funcionando (você pode se certificar que realmente salvou os dados acessando o MongoDB via linha de comando), ainda temos o problema da listagem, pois toda vez que carregamos nossa index.html a tabela vem vazia, independente do banco de dados.

Para arrumar isso, vamos criar uma nova função no scripts.js que traz os clientes e os listam na tabela:

Código a2.45: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function loadTable(){
  const tbody = $('table > tbody')
  tbody.empty()
  $.getJSON(webApiDomain + '/clientes', function(data){
    data.forEach(item =>{
      let linha = '<td>' + item.nome + '</td><td>' + item.idade +
      '</td><td>' + item.uf + '</td>'
      tbody.append('<tr>' + linha + '<td><input type="button" value="X" /></td></tr>')
    })
  })
}
```

Essa função limpa o TBODY da TABLE e faz uma requisição GET Ajax já considerando que o retorno será JSON. Na função de callback do getJSON nós iteramos sobre o array JSON retornado do servidor e para cada objeto JSON construímos uma linha que será inserida no TBODY da TABLE, de forma muito semelhante ao que já fizemos anteriormente.

Essa função loadTable deve ser chamada dentro do evento ready do document, logo no início do mesmo, como abaixo:

Código a2.46: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
$(document).ready(function(){  
    loadTable();  
})
```

Agora abra novamente o seu index.html e clique no botão de ver a listagem de clientes. Você vai ver que ela estará carregada!!!

Exemplo JQuery

Sistema para exemplificar uso de JQuery

Listagem

Nome	Idade	UF	Ações
Luiz	29	RS	<input type="button" value="X"/>
Teste	28	RS	<input type="button" value="X"/>
jquery	15	SC	<input type="button" value="X"/>
ajax	12	SC	<input type="button" value="X"/>

Mas nosso trabalho ainda não acabou, afinal, aquele nosso botão de excluir ali da imagem ainda não funcionam 'de verdade'. Hoje eles apenas removem a linha da tabela, mas não o cliente do banco.

Para fazer ele funcionar é um pouco mais complicado pois precisamos ir no banco para remover o cliente daquela linha. O jeito certo de fazer isso é tendo uma informação única, que não se repita entre os clientes, como o `_id` do MongoDB. No entanto, hoje não temos essa informação à nossa disposição.

Na função loadTable() isso é fácil de resolver, vamos modificar levemente ela, colocando essa informação em um data attribute no input de exclusão. Os data attributes são atributos personalizados iniciados com o prefixo 'data-' que a W3C (órgão que mantém os padrões do HTML) recomenda quando precisar de dados personalizados em uma tag HTML.

Assim, queremos algo como

```
<input type="button" value="X" data-  
id="59ab46e433959e2724be2cbc" />
```

E para fazer isso no loadTable, é bem simples (a única linha que mudou foi a de append):

Código a2.47: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function loadTable(){  
    const tbody = $('table > tbody');  
    tbody.empty();  
    $.getJSON(webApiDomain + '/clientes', function(data){  
        data.forEach(item =>{  
            let linha = '<td>' + item.nome + '</td><td>' + item.idade +  
            '</td><td>' + item.uf + '</td>';  
            tbody.append('<tr>' + linha + '<td><input type="button"  
value="X" data-id="' + item._id + '" /></td></tr>');  
        })  
    })  
}
```

Agora quando a tabela for carregada à partir do banco de dados, ela terá o data-id definido no botão de excluir. Para fazer com que a exclusão funcione de verdade, vamos primeiro criar uma função nova no scripts.js, para fazer a requisição Ajax de DELETE. Note que o JQuery não define funções simples como \$.post e \$.getJSON

para os demais verbos HTTP, logo temos de usar a função `$.ajax`, que é mais 'crua' e ao mesmo tempo mais completa:

Código a2.48: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
function deleteCustomer(id, callback){  
    $.ajax({  
        url: webApiDomain + '/clientes/' + id,  
        method: 'DELETE',  
        success: function(result) {  
            alert('Cliente excluído com sucesso!')  
            callback()  
        }  
    })  
}
```

A função `$.ajax` espera apenas um parâmetro que é um JSON de configurações da requisição, como a URL, o method e o callback de sucesso. O resto é auto-explicativo.

E no evento de click dos nossos botões de excluir, modificamos da seguinte forma:

Código a2.49: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
($('table').on('click', 'input[value="X"]', function(){  
    if(confirm('Tem certeza que deseja excluir este cliente?')){  
        const input = $(this);  
        const id = input.attr('data-id');  
        deleteCustomer(id, function(){  
            input.closest('tr').remove();  
        })  
    }  
})
```

Aqui usei a função attr para pegar um atributo do input (data-id) e com esse id consegui fazer o resto que precisava. O código que remove a linha no HTML foi colocado como callback da deleteCustomer, para garantir que só será feito após a exclusão ter sido realizada no banco de dados.

Isso é o suficiente para que a exclusão de clientes funcione. Ou quase. Só funciona para clientes que já estavam cadastrados antes de você abrir o index.html no navegador pois, os clientes cadastrados após isso, não possuem data-id nos inputs de exclusão.

A solução mais rápida para isso é chamar o loadTable() após cada cadastro, para garantir que inclusive vamos pegar clientes cadastrados por outras pessoas (considerando que futuramente esse sistema web seja usado por mais alguém que não apenas você).

Código a2.50: disponível em <https://www.luiztools.com.br/livro-node-fontes>

```
$('form').submit(function(event){  
    const data = $(this).serializeArray();  
    updateDatabase(data, loadTable);  
    $('#divListagem,#divCadastro').toggle();  
    event.preventDefault();  
})
```

Com isso nossa função updateTable perdeu utilidade e você pode removê-la do scripts.js, encerrando aqui nossa prática de Ajax com JQuery, usando Node.js como back-end.

Em nosso próximo capítulo, finalmente vamos estilizar nossas páginas, usando essa mesma index.html como base.

Nota: nos capítulos deste livro lhe mostrei três abordagens completamente diferentes de criação de aplicações web. Uma

usando Node.js como backend e EJS para construção das views.
Outra usando Node.js como backend e JavaScript para construção das views e esta última, onde substitui o JavaScript por jQuery.

Existem outras formas ainda mais profissionais usando frameworks reativos de front-end como ReactJS, VUE.js e Angular, mas isso é assunto para outro livro.

Quer fazer um curso online de Node.js e MongoDB com o autor deste livro? Acesse <https://www.luiztools.com.br/curso-nodejs>