

Primeiros passos com

Node.js



Casa do
Código

JOÃO RUBENS

ISBN

Impresso e PDF: 978-85-5519-285-2

EPUB: 978-85-5519-286-9

MOBI: 978-85-5519-287-6

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Como sempre, em toda a minha trajetória de vida, devo agradecer ao Senhor Jesus pelas graças alcançadas. Sem minha fé em ti, nada disso seria possível.

Agradeço à minha querida esposa, Elisa, que me motivou e incentivou desde o início desta jornada.

Agradeço ao pessoal da Casa do Código, em especial à Vivian Matsui, pelo trabalho de revisão e aconselhamento literário.

Por fim, e não menos importante, agradeço a toda comunidade de desenvolvedores web, que prontamente atendem aos chamados dos seus pares, desenvolvedores, nos diversos fóruns da vida, esclarecendo dúvidas e compartilhando conhecimentos.

SOBRE O AUTOR

Sou fascinado por desenvolvimento desde os tempos de faculdade. Mesmo antes, já tinha tido contato com Visual Basic 6.0 em um curso técnico, desenvolvendo um sistema simples de gestão para imobiliárias ao final do curso.

Cursei Tecnologia em Análise e Desenvolvimento de Sistemas na UNICAMP. Foi lá que conheci C++ e me interessei por Java. Fiz a doidera de integrar dois sistemas, Java e C++, utilizando JNI, como parte do desenvolvimento de minha dissertação de mestrado, que lidava com algoritmos sobre grafos. Não parei de estudar programação desde então.

Mas foi quando começar a lecionar que me deparei com desenvolvimento para web. Foi amor à primeira vista, há mais de 6 anos! Hoje, leciono linguagens de programação no curso técnico, faculdade e pós-graduação. Troquei o Java por outras três linguagens, que me identifico muito: C#, JavaScript (Node.js) e PHP.

Apesar de não ter uma rotina de desenvolvedor *full time*, pois sou professor de carreira, tenho de ler e estudar sobre tecnologia todos os dias, além de criar estratégias que alavanquem o aprendizado dos meus alunos. Nos momentos de lazer, dentre outras atividades, uno minha paixão de desenvolver com a de ensinar, escrevendo livros e apostilas, palestrando e gravando videocasts sobre tecnologia.

Você pode me encontrar no YouTube (procure por Professor Binho),
Facebook

(<https://www.facebook.com/professorbinhoaulas>) e fóruns de discussão. Se algum dia nos encontrarmos, será um prazer trocar ideias e experiências!

SOBRE O LIVRO

Node.js, sem dúvidas, é uma tecnologia que veio para ficar. Na comunidade de desenvolvedores web, todos os dias, nos deparamos com notícias e novas funcionalidades que envolvem Node.js.

O principal objetivo deste livro é apresentar ao leitor conhecimentos teóricos de nível básico e intermediário sobre Node.js, que o habilite a compreender cenários em que essa tecnologia possa ser aplicada, por conta de suas diversas características.

Para isso, o livro traz uma série de exemplos, divididos em capítulos, sobre o uso de Node.js e suas tecnologias associadas à configuração e utilização de um servidor web. Esses conceitos habilitam o leitor a desenvolver aplicações web *server-side* a partir do zero, trazendo os fundamentos da tecnologia como apoio.

Para demonstrar cada característica da plataforma, vou apresentar diversas ferramentas associadas ao desenvolvimento em Node.js. Com elas, é possível alcançar os objetivos secundários deste livro, que são:

- Utilizar o framework de aplicativos web *Express*;
- Utilizar o módulo `nodemon` para *watch* de processos;
- Criar e configurar *Middlewares*;
- Aplicar o `nunjucks` como *template engine* de páginas web;
- Aplicar o módulo `passport` para autenticação de usuários;

- Utilizar sistemas gerenciadores de banco de dados SQL (MySQL) e NoSQL (MongoDB) com Node.js;
- Realizar *deploy* de uma aplicação construída do início em um serviço de *Cloud*.

Os diversos projetos criados no decorrer do livro estão hospedados no *GitHub*. Você pode acessá-los a partir do link <https://github.com/bmarchete/primeiros-passos-nodejs>.

Caso você queira tirar dúvidas ou deixar sugestões e comentários, acesse o fórum da Casa do Código, em <http://forum.casadocodigo.com.br>.

Espero que você usufrua de todo conteúdo preparado neste livro, que foi desenvolvido com muito rigor e carinho!

A quem se destina

Este livro foi escrito para desenvolvedores web que muito ouviram falar sobre Node.js, mas pouco conhecem. O caro leitor precisará ter algum conhecimento de linguagens de programação, preferencialmente, JavaScript. Esse conhecimento facilitará o desenvolvimento das atividades práticas presentes aqui. Se este não for o caso, é possível ir aprendendo conforme se desenvolve a leitura do livro!

É importante também que o leitor conheça os fundamentos de criação de páginas web, como HTML, CSS, além de entender como ideia geral para que serve um framework.

Apesar de, em diversas partes do livro, estarem detalhadas características da comunicação cliente servidor, é interessante que o leitor conheça o funcionamento do protocolo *HTTP*.

Por último, para acompanhar os capítulos que envolvem operações com dados, o leitor deve conhecer o básico sobre sistemas gerenciadores de banco de dados, sintaxe SQL e um pouco sobre tecnologias NoSQL.

Sumário

1 Introdução	1
1.1 Node.js	2
1.2 Considerações	5
2 Instalação	7
2.1 Node.js (obrigatório)	7
2.2 Atom.io	14
2.3 Cmdr	17
2.4 Considerações	18
3 O que é o Node.js	19
3.1 JavaScript Runtime	19
3.2 Chrome's V8 JavaScript engine	21
3.3 Assíncrono orientado a eventos	22
3.4 Considerações	25
4 Iniciando um servidor web	26
4.1 O módulo http	26
4.2 Request e response	30
4.3 Considerações	35

5 Módulos CommonJs	36
5.1 Carregamento de módulos	36
5.2 Considerações	41
6 NPM	42
6.1 Package.json	42
6.2 Lodash	46
6.3 Nodemon module	49
6.4 Considerações	52
7 ECMAScript 2015 – ES6	53
7.1 Considerações	57
8 Web server com Express	59
8.1 Configurando o servidor com Express	60
8.2 Nodemon – watcher da aplicação	63
8.3 Estrutura da aplicação	66
8.4 App.js	70
8.5 Pasta bin, public, routes e views	75
8.6 Considerações	78
9 Nunjucks – template engine	79
9.1 Estruturas condicionais e de repetição	83
9.2 Herança de template	85
9.3 Páginas parciais	89
9.4 Considerações	90
10 Aplicação com MySQL	92
10.1 Exemplos do capítulo	92
10.2 Controle de playlist	93

10.3 Módulos de conexão	93
10.4 Configuração do Knex	94
10.5 Criação da tabela musicas	95
10.6 Criação do módulo de rotas	95
10.7 Desenvolvimento das páginas web	99
10.8 Operações com dados	104
10.9 Templating	110
10.10 Considerações	114
11 Integração com MongoDB	116
11.1 Exemplos do capítulo	117
11.2 Módulos de conexão	117
11.3 Configuração dos módulos	119
11.4 Criação do Schema para Musicas	119
11.5 Operações com dados	121
11.6 Considerações	124
12 Sessions	126
12.1 HTTP stateless	126
12.2 Aplicação com sessões	129
12.3 Configuração do app.js	129
12.4 Definição do arquivo de rotas	130
12.5 Páginas da aplicação	132
12.6 Considerações	133
13 Autenticação com Passport	135
13.1 Middleware de autenticação	135
13.2 Exemplos do capítulo	136

13.3 Arquivo app.js	141
13.4 Arquivo de rotas	142
13.5 Considerações	144
14 Restrição de acesso a páginas	146
14.1 Exemplos do capítulo	146
14.2 Aplicação	147
14.3 Middleware para restrição de acesso	147
14.4 Arquivo de rotas	149
14.5 Considerações	150
15 Aplicação final	151
15.1 Aplicação - Controle de tarefas	151
15.2 Páginas web	152
15.3 Configuração do Express	155
15.4 Módulos adicionais	158
15.5 Configuração de acesso ao banco de dados	161
15.6 Configurações do módulo de autenticação	162
15.7 Configurações das rotas	164
15.8 Configurações das páginas Web	170
15.9 Teste da aplicação	178
15.10 Considerações	179
16 Deploy	180
16.1 Conta no heroku	181
16.2 Setup do ambiente de deploy	183
16.3 Instalando a aplicação	187
16.4 Considerações	191

17 Daqui para a frente...

193

INTRODUÇÃO

Nos arredores das comunidades de desenvolvimento de sistemas, principalmente aquelas relacionadas ao desenvolvimento web e mobile, muito se tem falado sobre aplicações escaláveis, serviços real-time, microsserviços, componentização e segurança nos últimos cinco anos. Mesmo sabendo pouco sobre esses conceitos, todo desenvolvedor de sistemas acaba se deparando com eles no seu dia a dia, cedo ou tarde. Se não é o seu caso, não se preocupe, pois eles serão explorados com mais profundidade ao longo do livro.

Por conta de fatores como alta adesão por parte dos usuários a serviços acessados pelo smartphone, uso intenso de redes sociais e a ascensão de dispositivos IoTs (*Internet of Things*, ou Internet das Coisas), os velhos paradigmas de desenvolvimento de software tiveram de mudar. Também, as grandes áreas do desenvolvimento, como front-end, back-end e infraestrutura, dividem hoje fatias iguais de importância no mercado de software.

Pensar em desenvolver software hoje vai muito além da escolha de uma linguagem que seja de fácil entendimento ou de costume dos programadores. Quando se pensa em software, se imagina uma aplicação como um todo. Isso significa envolver os tipos de usuários que vão consumir a aplicação, qual será a demanda de

dados, além de, por exemplo, qual é a disponibilidade que os servidores deverão apresentar. Todos esses fatores vão impactar diretamente na escolha das tecnologias envolvidas com o processo de desenvolvimento.

Não são poucos os casos de empresas que tiveram de se adaptar, ou até mesmo trocar de tecnologia por conta das características intrínsecas das aplicações que desenvolvem. É possível encontrar, em uma busca rápida pela web, exemplos de empresas que migraram as tecnologias envolvidas com o desenvolvimento e disponibilidade dos seus produtos, por conta dos fatores apresentados. Dentre elas, podem-se citar:

1. **NetFlix:** criação de frameworks próprios para atender suas demandas, como é o caso do Falcor (<https://netflix.github.io/falcor/>), que funciona como uma interface para os dados da aplicação;
2. **Facebook:** criação de uma linguagem nova para atender às necessidades da plataforma (Hack - <http://hacklang.org/>);
3. **Microsoft:** criação de um ecossistema open source para suas plataformas de desenvolvimento de aplicativos, sites e serviços (.NET Core - <https://blogs.msdn.microsoft.com/dotnet/2014/11/12/net-core-is-open-source/>).

Esses são exemplos bem banais do que ocorre diariamente com empresas que desenvolvem sistemas. Convido você a buscar mais exemplos para entender como está o mercado de software atualmente.

1.1 NODE.JS

O Node.js é uma ferramenta que surgiu nos moldes deste cenário, e ganhou espaço por conta de diversas características apresentadas neste livro. É comum se perguntar o porquê de se adotar uma nova tecnologia, como o Node.js, pois essa adoção envolve treinamento de pessoal, maturidade da ferramenta, vantagens e desvantagens.

Apesar disso, nem sempre esse processo é opcional. Existem tecnologias que estão muito adiante das demais concorrentes em casos específicos. Apenas isso já justifica a migração de sistemas inteiros e horas de capacitação de pessoal, pois o serviço entregue no final é sempre o maior objetivo.

Dito isso, apresento aqui alguns casos em que a adoção do Node.js, por parte de empresas de tecnologia, foi muito além do que meros caprichos:

NASA

Engenheiros da NASA utilizaram Node.js para criar uma API que integra três bases de dados em um só, baseado em um serviço de *Cloud*. Isso permitiu que dados gerados, como por exemplo, em missões extraveiculares, fossem acessados de maneira muito mais eficaz.

Se você quiser saber mais sobre o uso de Node.js na *Nasa*, acesse o link https://nodejs.org/static/documents/casestudies/Node_CaseStudy_Nasa_FNL.pdf.

GoDADDY

A GoDaddy migrou seus projetos front-end e back-end para Node.js, deixando a tecnologia .NET por conta da comunidade que trabalha nos módulos oferecidos pelo sistema de a linguagem ser mais “vibrante”. Isso aponta uma das principais vantagens do Node.js: seu sistema de módulos comunitário.

Fonte: https://www.youtube.com/watch?v=dvMq4gqBleo&index=2&list=PLfMzBWSH11xYNeUJXap4NTMX3TFtH-_0z.

NETFLIX

A Netflix vem usando um sistema baseado em JVM (*Java Virtual Machine*) com uma camada de acesso desenvolvida com Node.js e Docker, por fatores de “escalabilidade horizontal de acesso”.

Fonte: <https://www.youtube.com/watch?v=p74282nDMX8&feature=youtu.be&t=12m11s>.

Na contramão do que muitas empresas de desenvolvimento apostaram (por se tratar de uma tecnologia fundamentada na linguagem JavaScript), Node.js já tem maturidade suficiente para encarar os grandes desafios de desenvolvimento de sistemas atuais.

Ele vem se provando ser uma plataforma de alta eficiência em diversos casos, como por exemplo servidores web.

Em um estudo que compara diversas características de Node.js e C# (veja detalhes em <https://gist.github.com/bmarchete/eef728e4af1654c7053f302e3c677e28>), como simplicidade no desenvolvimento e performance, foi possível observar em um dos testes que um sistema desenvolvido em Node.js supera um desenvolvido em C# na aceitação do número de requisições para uma leitura de dados vinda por *post*, ao passo que leva menos tempo por requisição, como mostra a figura a seguir:

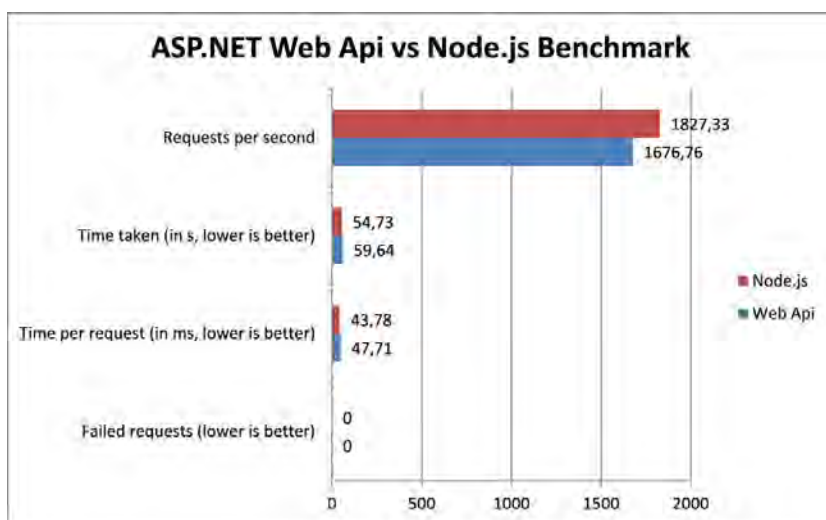


Figura 1.1: Teste de performance entre sistema Node.js e C# Web API. Fonte: <http://mikaeltkosinen.net/post/asp-net-web-api-vs-node-js-benchmark-take-2>

1.2 CONSIDERAÇÕES

São diversos os motivos para a adoção de Node.js no

desenvolvimento de sistemas. Entretanto, os fatores como performance e simplicidade no desenvolvimento fizeram com que conhecer essa tecnologia se estabelecesse como habilidade indispensável para desenvolvedores *front-end* e *back-end*.

INSTALAÇÃO

Durante os exemplos apresentados neste livro, serão usadas apenas ferramentas gratuitas. Algumas delas serão obrigatórias para se conseguir executar os comandos apresentados, outras serão de livre escolha.

Este capítulo apresenta um tutorial para instalação de todas elas em ambiente Windows. O usuário de Linux ou Mac pode acessar o site oficial de instalação e seguir os passos, que são bem semelhantes.

2.1 NODE.JS (OBRIGATÓRIO)

Para desenvolver sistemas baseados em Node.js, é preciso instalar um pacote de ferramentas e bibliotecas disponíveis em <https://nodejs.org>. Para realizar a instalação, proceda da seguinte maneira:

1. Acesse o site <https://nodejs.org>. Em seguida, clique sobre o link com a descrição de *recommended for most users*, ou se preferir, *lastest features*. Neste livro, usaremos o segundo.

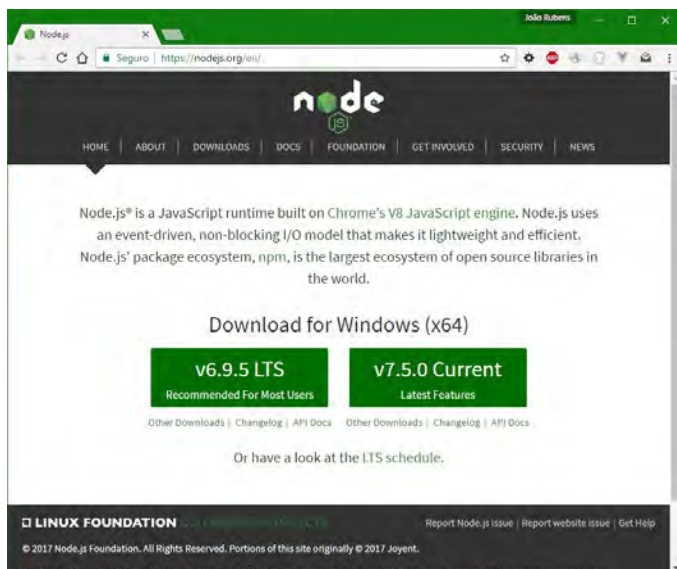


Figura 2.1: Site oficial do Node.js

2. Após feito o download do instalador, abra o executável.

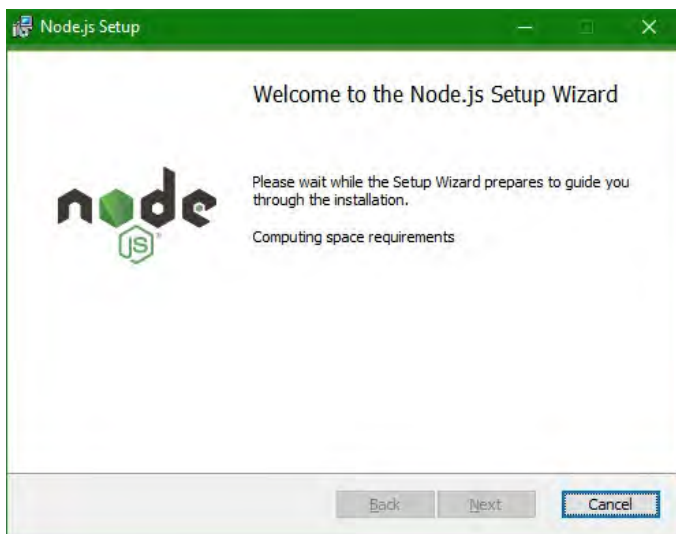


Figura 2.2: Início da instalação do Node.js

3. Clique em **Next** .
4. Em seguida, aceite os termos de condição de uso e clique em **Next** .

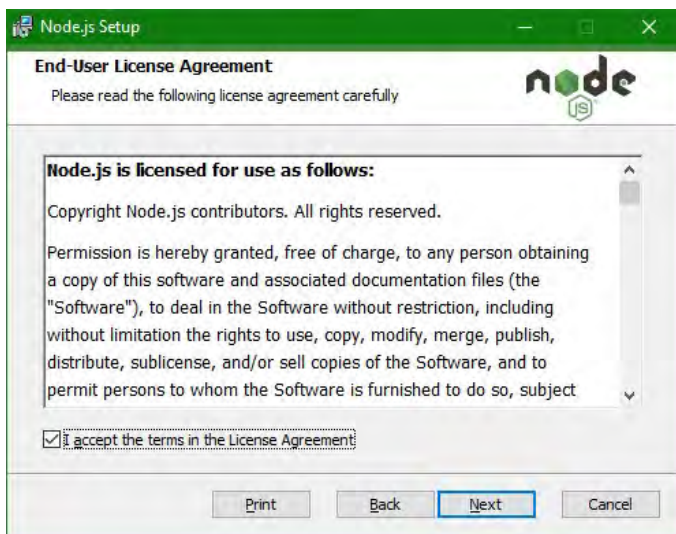


Figura 2.3: Termos de instalação

5. Deixe o caminho de instalação padrão e clique em **Next** .

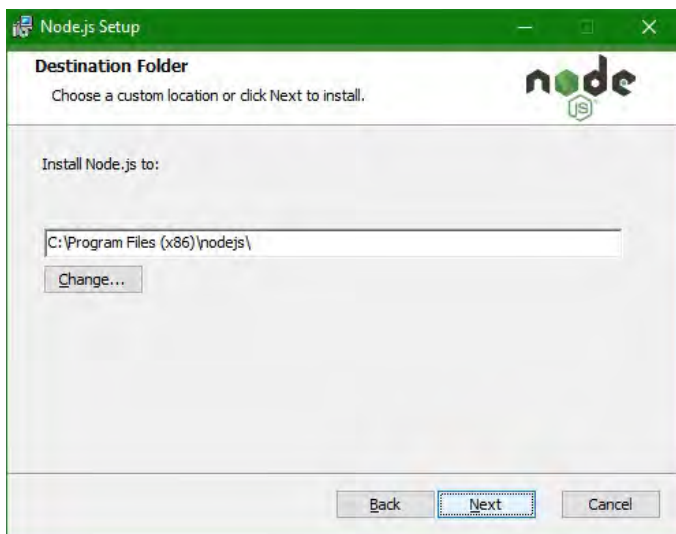


Figura 2.4: Escolha o caminho para instalação

6. Deixe todas as opções ativadas.

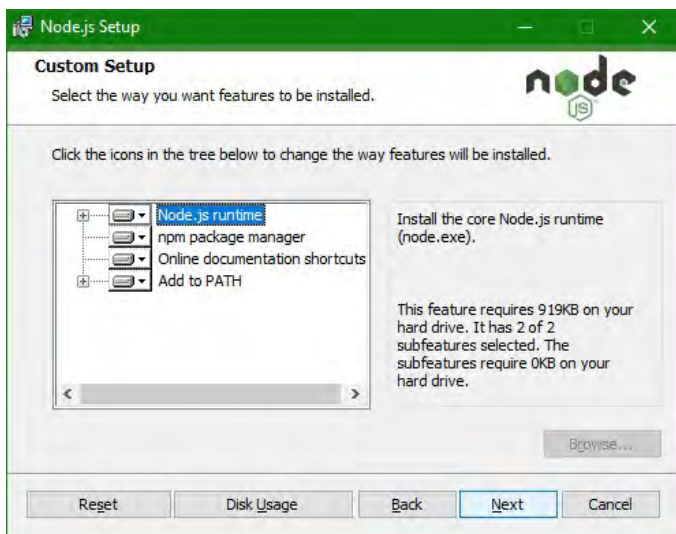


Figura 2.5: Opções de instalação

7. Por fim, clique em `Install` .

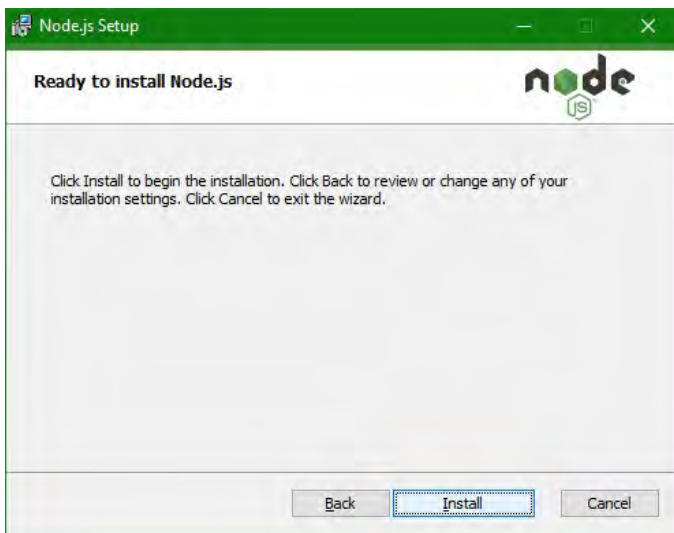


Figura 2.6: Finalizando a instalação

Esta instalação cria uma variável de ambiente e permite acessar os comandos via terminal. O terminal será muito utilizado nos exemplos do livro, portanto, verifique se é possível acessar as ferramentas do Node.js a partir dele. Para isso, proceda da seguinte forma:

1. Abra o prompt de comandos segurando as teclas `Windows+R`. Em seguida, digite `cmd` e aperte `Enter`.

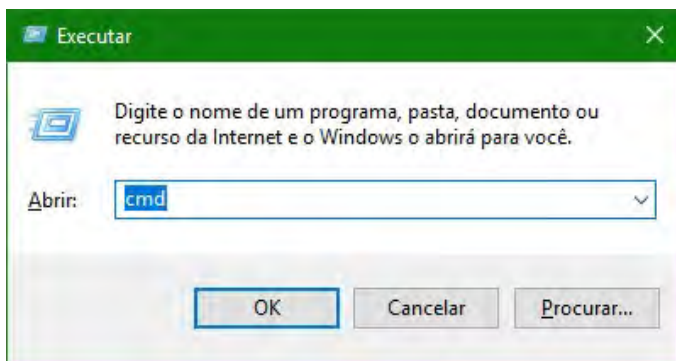


Figura 2.7: Caixa de execução de comandos

2. No terminal, digite o comando `node -v`.

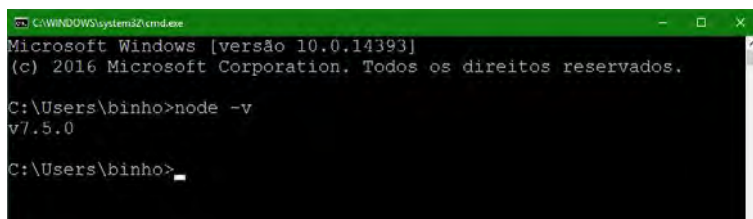


Figura 2.8: Terminal para teste do Node.js

A mensagem que aparece indica o correto funcionamento do Node.js, com sua respectiva versão. No momento da escrita deste livro, a versão final era a **v7.5.0**.

2.2 ATOM.IO

O Atom é o ambiente de desenvolvimento utilizado neste livro. Além de gratuito, ele conta com uma diversidade de plug-ins para personalizá-lo, uma interface customizável e uma ótima

documentação.

Existem outras boas escolhas, como o *WebStorm* (<https://www.jetbrains.com/webstorm/>), o *Visual Studio Code* (<https://code.visualstudio.com/>) ou o *Sublime* (<https://www.sublimetext.com/>). A seguir, segue um tutorial de instalação do Atom, caso ele seja sua escolha para seguir os exemplos do livro:

1. Entre no site <https://atom.io/>, e clique em Download Windows Installer .



Figura 2.9: Site oficial do Atom

2. Execute o arquivo de instalação e aguarde o carregamento.



Figura 2.10: Instalação do Atom

Uma vez instalado, já é possível criar novos projetos no editor Atom.

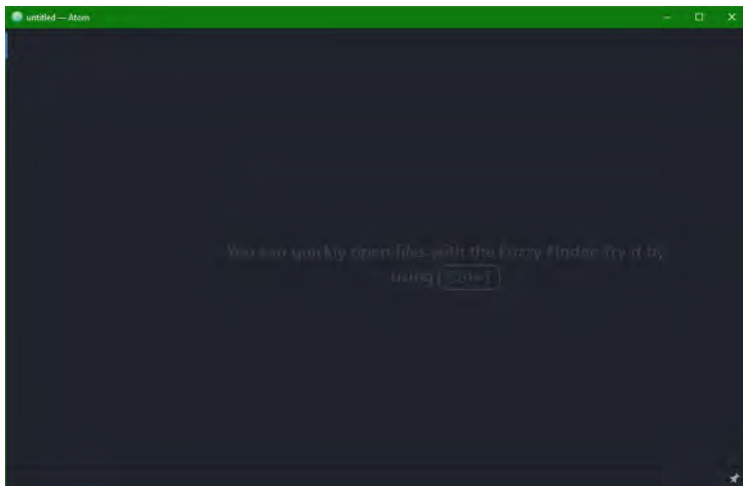


Figura 2.11: Interface do editor Atom

2.3 CMDER

Neste livro, será usado um prompt de comandos não nativo do Windows, que pode ser baixado em <http://cmdr.net/>. Para instalar, proceda da seguinte forma:

1. Acesse o site <http://cmdr.net/>. Role a página até encontrar a seção de download e clique no link Download mini .

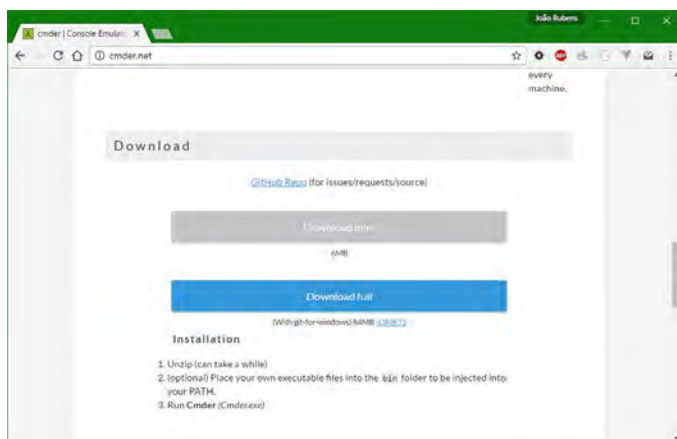


Figura 2.12: Site do Cmdr

2. Descompacte o arquivo em uma pasta de sua escolha. Para executar, clique duas vezes sobre o arquivo Cmdr.exe .

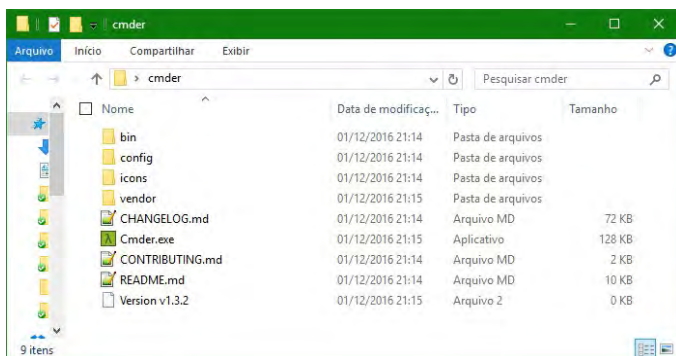


Figura 2.13: Arquivos do Cmder descompactados

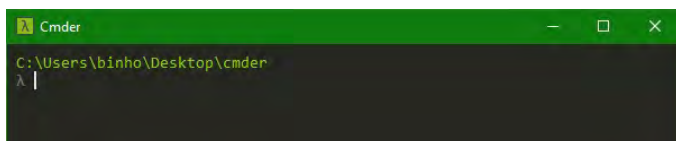


Figura 2.14: Tela de prompt do Cmder

2.4 CONSIDERAÇÕES

Com todas as ferramentas de edição de código e manipulação do Node.js, será possível executarmos todos os exemplos das funcionalidades da tecnologia apresentadas no livro, a partir do próximo capítulo.

O QUE É O NODE.JS

Antes de entrar em detalhes sobre a tecnologia, a partir deste ponto, sempre que usar *Node*, me refiro ao *Node.js*.

O Node, pela definição oficial (<https://nodejs.org/en/about/>), é um *runtime*, ou em uma tradução livre, um **interpretador** de *JavaScript* assíncrono orientado a eventos, desenvolvido para lidar com aplicações de rede escaláveis. Isso significa que o Node fica na espera de eventos serem disparados para executar uma ação.

Para entender melhor, é preciso ir por partes.

3.1 JAVASCRIPT RUNTIME

O Node tem a capacidade de interpretar código JavaScript, assim como um navegador o faz. Quando um navegador recebe um comando escrito em JavaScript, ele o interpreta, como que convertendo para uma linguagem de *máquina*, e em seguida executa as instruções que aquele comando fornece.

Quando o navegador encontra o comando `alert('Mensagem')` em uma tag de script HTML, por exemplo, ele é capaz de mostrar um *pop-up* com uma mensagem. A interpretação do comando e seguinte execução são realizadas após

o carregamento do arquivo ou página em que se encontra o comando.

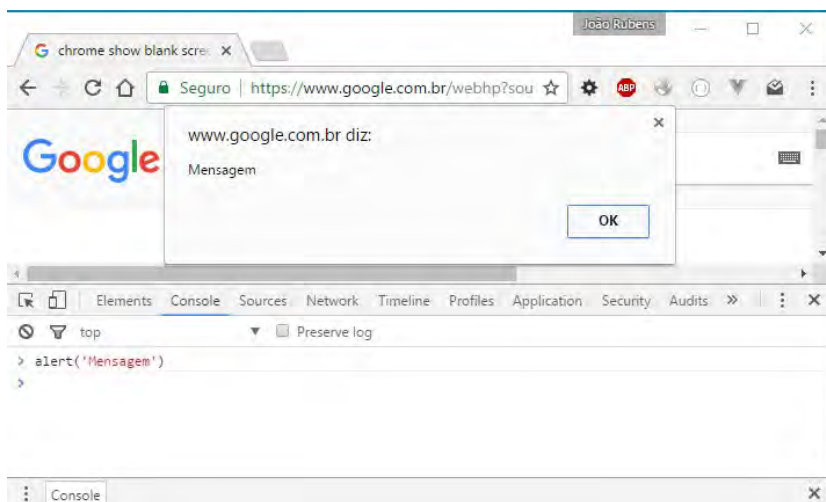


Figura 3.1: Exemplo de navegador executando JavaScript

O Node faz algo bem parecido, porém, sem a necessidade de um navegador ativo. Para ver como isso funciona, proceda da seguinte maneira:

1. Abra o terminal. Em seguida execute o comando `node` .

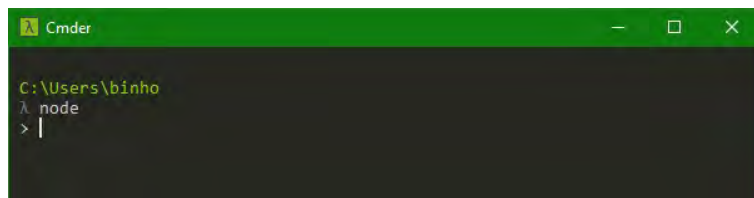


Figura 3.2: Terminal com comando node

2. Em seguida, digite os comandos conforme a figura a seguir.

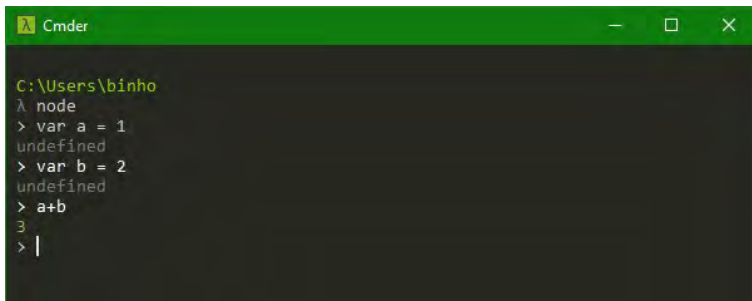
A screenshot of a Windows Command Prompt window titled 'Cmder'. The window has a green title bar with standard minimize, maximize, and close buttons. The background is dark gray. The text inside shows the command prompt at 'C:\Users\binho'. The user has entered 'λ node' and the prompt has changed to '>'. They then entered 'var a = 1', and the prompt returned 'undefined'. Next, they entered 'var b = 2', and the prompt again returned 'undefined'. Finally, they entered 'a+b', and the prompt returned '3'. The cursor is now on a new line after the prompt '> |'.

Figura 3.3: Execução de código JavaScript no terminal

A partir da figura apresentada, é possível observar a execução de um programa *JavaScript*, no qual são definidas duas variáveis e, em seguida, apresentado o resultado da soma entre elas.

Após o comando `node`, iniciou-se um novo processo que engloba um interpretador JavaScript e um *CLI* (*Command-line Interface*). Um *CLI* é um prompt de comando que permite ao usuário inserir comandos. Tudo que ocorre nesse processo, em termos de comandos JavaScript, é interpretado e executado pelo *runtime* do Node. Costuma-se pensar que o próprio Node faz isso, pois uma de suas definições é "ser um *runtime*".

Para que seja possível todo esse processo ocorrer fora de um navegador, o Node utiliza uma ferramenta chamada *Chrome's V8 JavaScript engine*.

3.2 CHROME'S V8 JAVASCRIPT ENGINE

O *Chrome's V8 JavaScript engine* é um sistema que trata de uma maneira bem específica a execução de programas JavaScript. Ele compila e executa o código JavaScript em vez de apenas

interpretá-lo. Além disso, ele faz o gerenciamento da alocação de memória de maneira muito eficiente.

A partir destas premissas, o V8 consegue aumentar significativamente a velocidade com que programas desenvolvidos em JavaScript executam. Por ser um sistema independente do browser, ele pode ser usado como ferramenta por várias plataformas, como fazem o Google Chrome e o Node.js. O Node, em sua instalação, já traz consigo uma versão do V8.

3.3 ASSÍNCRONO ORIENTADO A EVENTOS

Um dos motivos pelos quais o Node se diferencia das outras plataformas de programação, como *Java*, *PHP* e *C#*, é que ele não trabalha com *multithreads*. Isso significa que o Node não é capaz de iniciar novas *threads* que executam tarefas em paralelo, como fazem essas outras linguagens. Por isso, cada processo iniciado pelo Node é considerado *monothread*.

Cada requisição feita a um processo executado pelo Node entra em uma fila de espera para ser processada, e não pode ser definida para uma nova *thread*. A princípio, isso soa estranho, pois passa a impressão de que tudo vai ficar mais lento, já que cada requisição é processada em fila.

Entretanto, algumas características fazem com que tudo aconteça de maneira muito rápida. Um sistema *monothread* não possui a necessidade de ficar gerenciando *threads*, evitando o desperdício de processamento e consumo de memória com a criação e manutenção de cada *thread*.

Apesar de ganhar em performance por ser *monothread*, esperar

em uma fila a execução de cada requisição ser concluída faria com que a plataforma fosse inviável. Por isso, o Node funciona de forma não bloqueante (*non blocking*), ligado diretamente ao sistema de *call-backs* da linguagem JavaScript.

Para entender a ideia melhor, pense no seguinte cenário: um programa precisa acessar os dados da conta de uma pessoa e imprimir na tela algumas informações. Além disso, ele deve verificar o status do cliente na agência, carregar o perfil de investimento do cliente e enviar esses dados para o gerente da conta. O algoritmo de execução disso leva, em média, os seguintes tempos de execução:

1. Acessar dados da conta – 40ms
2. Imprimir informações – 10ms
3. Carregar perfil de investimento – 40ms
4. Enviar dados de perfil para gerente – 10ms

Como acessar os dados de uma conta e carregar o perfil de investimento requerem acesso ao banco de dados, eles acabam sendo tarefas mais demoradas. Executando essa sequência de instruções em um sistema bloqueante (programa *PHP*, por exemplo), o total do tempo de execução seria a soma dos tempos, ou seja, **100ms**. Isso ocorre porque cada passo do algoritmo é executado de forma **síncrona**, e um passo **bloqueia** o outro até terminar de ser executado.

Já em um sistema não bloqueante pode-se definir quais tarefas são dependentes ou não de outras. No exemplo anterior, o algoritmo poderia ficar da seguinte forma:

1. Acessar dados da conta – 40ms

- Imprimir informações – 10ms
- 2. Carregar perfil de investimento – 40ms
 - Enviar dados de perfil para gerente – 10ms

Neste caso, o passo número 1 independe do passo número 2. Em um sistema não bloqueante, o total do tempo de execução seria de 50ms, já que o passo dois pode ser executado mesmo se o passo 1 não tiver sido concluído ainda.

Depois que o passo 1 começar a ser executado, o passo 2 também é executado. Como o passo 2 depende do passo 1 para ser executado, ele aguarda os 40ms de execução. Na soma, o passo 1 é concluído em 50ms. Mas de forma **assíncrona**, ou **não bloqueante**, o passo dois também sofre o mesmo processo, e termina em 50ms. Isso significa que, ao fim dos **50ms**, ambos os passos 1 e 2 já finalizaram suas execuções, apresentando-se na metade do tempo de um sistema bloqueante.

Com o Node, isso é possível pelo sistema de *Event Loop*. Cada requisição ao sistema, ou no exemplo anterior, cada passo principal do algoritmo é tratado como um evento. Os eventos são independentes e não bloqueiam a entrada de outros eventos.

O *Event Loop* funciona como um programa ouvinte que fica perguntando para cada evento emitido se ele já concluiu o seu processamento. Quando um evento é concluído, ele dispara uma **função de retorno**, ou um *call-back*, que permite a execução de outras tarefas associadas ou dependentes dele.

O *Event Loop* controla o início e o resultado de cada evento, fazendo com que programas desenvolvidos neste paradigma possam receber dezenas de milhares de requisições e tratá-las de

forma paralela.

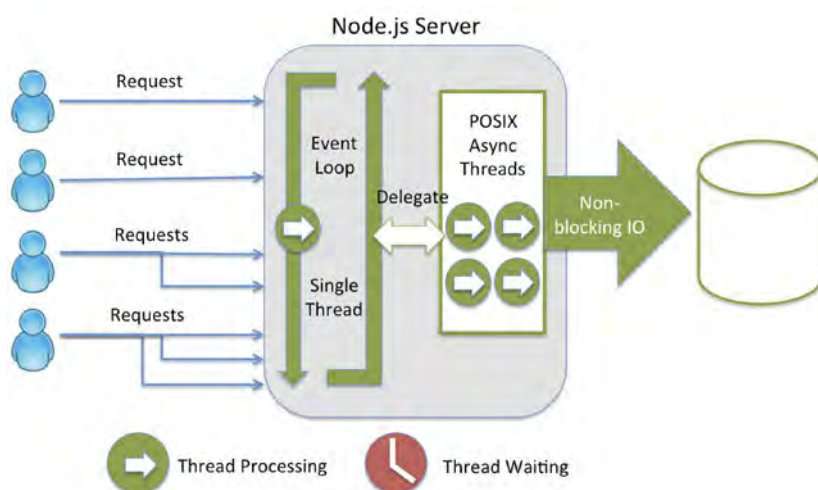


Figura 3.4: Exemplo de funcionamento do Event Loop. Retirado de <https://gist.github.com/ilyaigpetrov/f6df3e6f825ae1b5c7e2>

A grande estratégia do Node para ganhar em desempenho é paralelizar seus recursos em forma de ações não bloqueantes, ao invés de criar diversas threads para executarem tarefas em paralelo, já que o Node é um processo *monothread*.

3.4 CONSIDERAÇÕES

É importante entender o processo de eventos e assincronismo do Node, pois quando se desenvolvem sistemas nesta plataforma, deve-se saber quando optar por uma tarefa síncrona ou assíncrona, pensando sempre na arquitetura da plataforma como um todo.

INICIANDO UM SERVIDOR WEB

Agora que as bases do Node já foram apresentadas, juntamente com suas tecnologias associadas, podemos criar um pequeno servidor de páginas web.

O Node possui muitos módulos que já vem por padrão com a plataforma, e muitos outros desenvolvidos pela comunidade. Vários deles serão apresentados no decorrer do livro. Por ora, basta conhecer aqueles que permitem a criação de um servidor de páginas web!

4.1 O MÓDULO HTTP

O módulo *http* pode ser usado dentro de uma aplicação Node para lidar com as características do protocolo *HTTP*. Ele é considerado um módulo *low-level*, pois permite ao programador lidar com as questões essenciais do protocolo. Isso significa que é possível acessar todas as características do protocolo programaticamente, configurando cada uma delas manualmente. Apesar disso, iniciar um servidor web utilizando esse módulo é muito simples!

É claro que conhecer todos os casos de uso do protocolo requer um pouco mais de estudo, mas vale a pena dar uma olhada na documentação oficial para mais detalhes (atualmente *Node.js* v7.4.0). Para isso, acesse o endereço <https://nodejs.org/api/http.html>.

Outra dica importante para entender melhor o funcionamento deste módulo é conhecer as principais características do protocolo *HTTP*, como *headers*, *content-types*, *verbs*, dentre outros.

Para criar um novo servidor web, é preciso iniciar um novo projeto. Para isso, vamos criar um projeto chamado `node-server`, e um arquivo chamado `server.js` dentro dele.

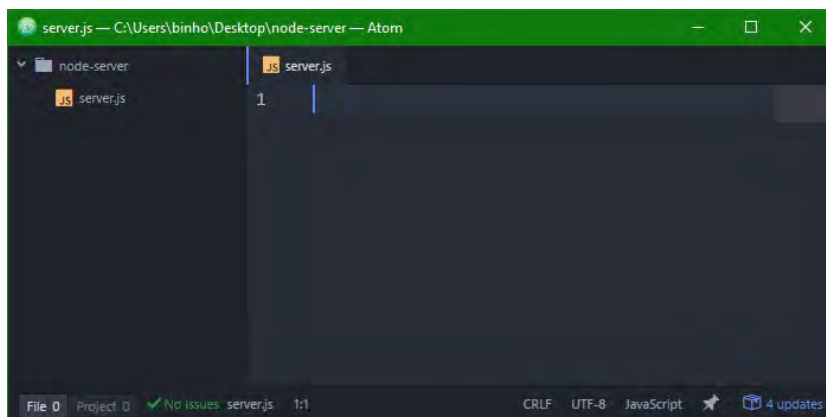


Figura 4.1: Estrutura inicial do projeto

Em seguida, é preciso fazer a requisição do módulo *http* para criar um novo servidor. A função `createServer()` recebe como entrada uma função *callback*, que conterà dois parâmetros inicialmente, com as configurações do servidor. Os parâmetros aqui citados são `request` e `response`, responsáveis por receber

as solicitações de acesso às páginas *web* e de enviar as respostas com as páginas em si.

Dessa forma, precisamos inserir o seguinte código no arquivo `server.js` :

```
const http = require('http');

const server = http.createServer((request, response)=>{
  console.log('Servidor acessado!');
});
```

Agora falta definir em qual porta o servidor de aplicações *web* vai funcionar. Isso é feito a partir da função `listen()` , indicando o número da porta que deve estar disponível no momento, é claro!

```
const http = require('http');

const server = http.createServer((request, response)=>{
  console.log('Servidor acessado!');
});

server.listen(3000);
```

Uma mensagem será emitida no console do servidor, indicando que ele recebeu uma requisição naquela porta, toda vez que alguém tentar acessar o servidor via protocolo *HTTP* na porta indicada.

O resultado final da aplicação é apresentado na figura a seguir:



Figura 4.2: Código de um servidor web

Vamos testar se nossa aplicação está funcionando? Para a realização de um teste, é preciso invocar a aplicação, a partir do console com o comando `node` (como visto no capítulo anterior). Para isso, abra o console, acesse a pasta `node-server` e digite o comando: `node server.js`

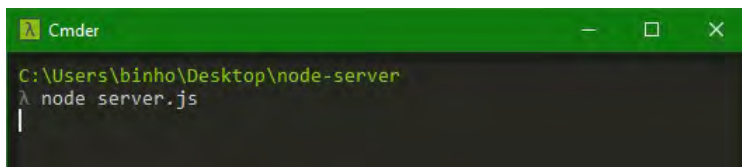


Figura 4.3: Invocando a aplicação

Em seguida, é preciso abrir um navegador, para que seja possível acessar a página a partir do endereço `localhost:3000`.

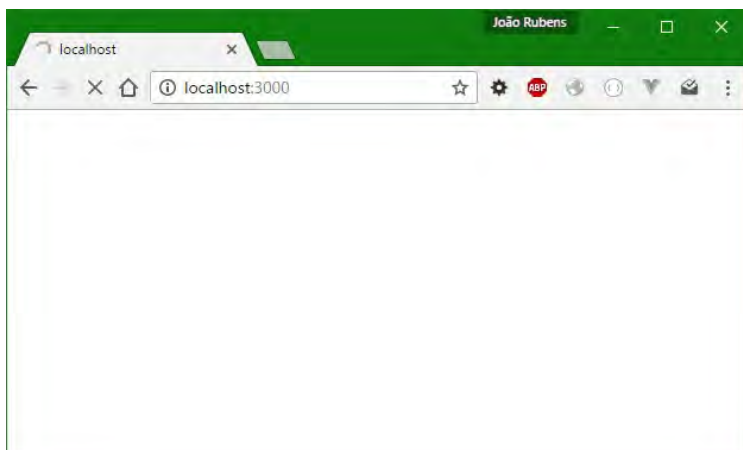


Figura 4.4: Navegador acessando endereço do servidor web

Observe que a página nunca carrega! Isso acontece porque não foi enviado nenhum retorno para o navegador quando foi feita a solicitação. O navegador fica esperando o *timeout* do sistema por conta disso. Se você realizar vários *reloads*, vai poder ver o que acontece no console.

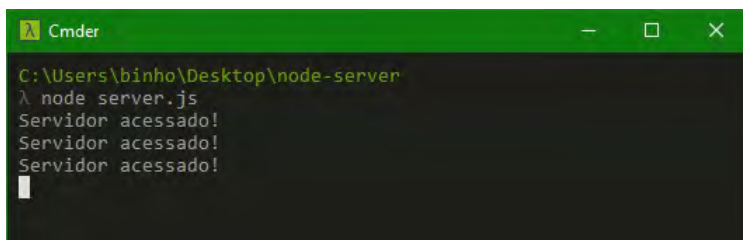


Figura 4.5: Console com mensagens de acesso ao servidor

4.2 REQUEST E RESPONSE

Fazer com que o navegador fique aguardando a resposta do servidor, por um tempo indefinido, não parece ser uma solução

agradável! Podemos melhorar esse projeto!

Como visto anteriormente, a função `http.createServer()` recebe uma função *callback* com dois objetos: `request` e `response`. Esses dois objetos são responsáveis por tratar dos dados oriundos da chamada recebida pelo servidor a partir do navegador, ou qualquer outro programa que faça requisições *HTTP*. Eles também devolvem os dados processados pelo servidor, sejam páginas *HTML*, arquivos *JSON* etc.

No exemplo a seguir, foram adicionadas algumas linhas de código à aplicação, que permitem a visualização de dados oriundos de uma requisição ao servidor.

```
const http = require('http');

const server = http.createServer((request, response)=>{

  const headers = request.headers;
  const method = request.method;
  const url = request.url;

  console.log("Headers");
  console.log(headers);
  console.log("Method: " + method);
  console.log("URL: " + url);

});

server.listen(3000);
```

Observe a criação de `const headers`, `const method` e `const url`. Essas constantes recebem as propriedades da requisição, como por exemplo, o `user-agent` e `cookies` (`request.headers`), o método de acesso da conexão (`request.method`) e o caminho do endereço acessado (`request.url`). Para visualizar os dados recebidos a cada nova

requisição, usamos aqui o comando `console.log()` , que apresenta as informações no console do servidor iniciado.

Ao iniciar a aplicação, realizando uma requisição de um navegador Chrome, é possível obter os seguintes dados no prompt de comando:

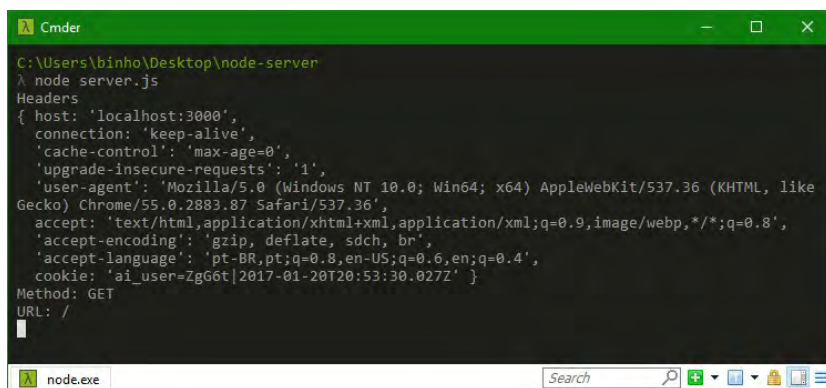


Figura 4.6: Dados do objeto request

É possível observar que a requisição foi feita ao servidor a partir do endereço raiz, representado por um `/` na impressão da URL, e que o método de acesso foi o `GET` .

Agora que já conhecemos melhor o funcionamento do objeto `request` , podemos utilizar o objeto `response` para enviar uma mensagem. Vamos configurar a aplicação para receber uma requisição *HTTP* e retornar a mensagem *Olá Mundo* em formato de *HTML*, caso a requisição atenda à URL `/teste` .

```
const http = require('http');

const server = http.createServer((request, response)=>{

  if (request.url == '/teste') {
```

```
response.setHeader('Content-type', 'text/html');
response.end('<h1>Olá Mundo</h1>')
}

});

server.listen(3000);
```

A estrutura `if` permite que apenas a URL `localhost:3000/teste` receba a resposta do servidor. A função `setHeader()` configura o servidor para que a resposta seja devolvida no formato de *HTML*. Esse cabeçalho é interpretado pelo navegador no momento que faz o render da página.

CONTENT-TYPE

Uma lista de content-types e suas especificações pode ser consultada em:
<https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>.

A função `response.end()` devolve à requisição o texto especificado. Veja no navegador o resultado para esta URL:

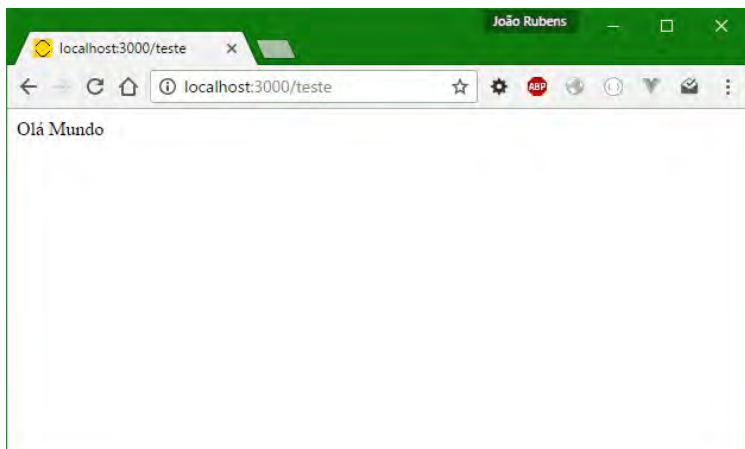


Figura 4.7: Navegador exibindo resposta do servidor

Também é possível enviar documentos no formato *JSON* nas respostas de requisições, de uma maneira rápida. Caso queira realizar um exemplo, refatore a aplicação com o código a seguir:

```
if (request.url == '/json') {  
  response.setHeader('Content-type', 'application/json');  
  const data = {  
    id: 1,  
    name: "user"  
  };  
  response.end(JSON.stringify(data));  
}
```

Neste exemplo, a partir do acesso ao endereço `localhost:3000/json`, é configurada uma resposta do tipo `application/json`. Em seguida, é criado um novo objeto JavaScript `const data`, que é convertido para uma String *JSON* com a função `JSON.stringify()`. Ao acessar a URL pelo navegador, é possível obter a seguinte resposta:

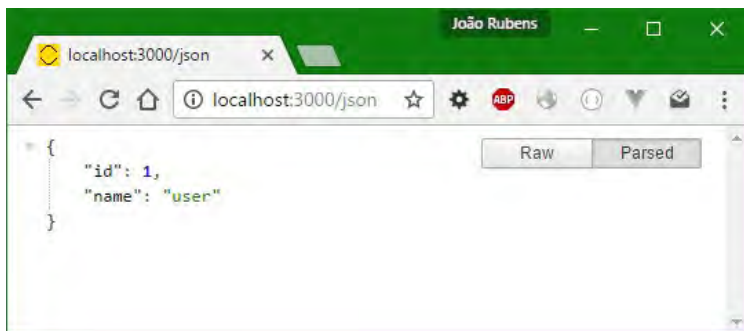


Figura 4.8: Navegador com resposta no formato json

4.3 CONSIDERAÇÕES

Neste capítulo, foram apresentados os rudimentos da programação de um servidor web utilizando *Node.js*. O módulo `http` disponibilizado pela tecnologia permite iniciar um servidor web, definir uma porta de escuta, configurar cabeçalhos de resposta, receber dados do cliente (nos exemplos, um navegador), além de outras funcionalidades não exploradas ainda. Com o uso de apenas um módulo, um servidor web é capaz de lidar com quase todas as questões envolvidas com os objetos de `request` e `response`.

MÓDULOS COMMONJS

Programas modulares estão na moda, e não é à toa. A capacidade de desenvolver pedaços de softwares que resolvem problemas específicos e plugá-los de forma transparente facilita muito o dia a dia de quem desenvolve grandes projetos, e precisa estar focado na regra de negócio do seu produto em vez de se preocupar com detalhes de implementações de soluções tão comuns no mercado.

Nativamente, *JavaScript* não possui um sistema de carregamento de módulos. Na verdade, esse sistema já foi implementado na linguagem na versão *ECMAScript 6*, mas essa implementação ainda não é totalmente suportada pelos navegadores e pelo Node.

5.1 CARREGAMENTO DE MÓDULOS

Para resolver o problema de carregamento de módulos, a comunidade de desenvolvedores criou alguns padrões. O Node.js adotou o padrão conhecido como *CommonJs*.

Uma das características que torna esse padrão ideal para o Node é o carregamento síncrono de módulos, já que ele vai acontecer apenas quando o servidor iniciar. Vamos desenvolver

um exemplo para entender como funciona a criação de um módulo!

Começamos criando um novo projeto chamado `modules-ex1`. Em seguida, criamos um arquivo chamado `server.js`, que conterá a inicialização do nosso servidor. Criamos também um arquivo chamado `operations.js`, para desenvolvermos nosso primeiro módulo.

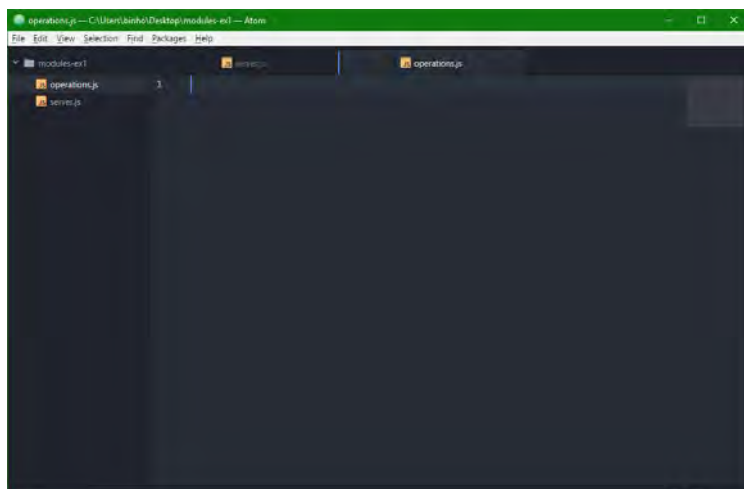


Figura 5.1: Início de um projeto para carregar módulos

Agora, dentro do arquivo `operations.js`, vamos desenvolver o módulo, inserindo o seguinte código:

```
module.exports = {  
  var1: 1,  
  var2: 2  
};
```

O código anterior define um novo módulo a ser retornado quando o arquivo é carregado. O módulo foi definido como um

objeto *JavaScript*, e por isso pode ser acessado como tal.

Para ver se o módulo está funcionando corretamente, é preciso importá-lo no local em que será utilizado. Por isso, acesse o arquivo `server.js` e insira o seguinte código:

```
const operations = require('./operations');

console.log(operations.var1);
console.log(operations.var2);
```

Desse modo, será realizada a importação do módulo, e os dados do objeto *JavaScript* definidos no módulo serão impressos no console. É importante destacar que os símbolos `./` que precedem o nome do arquivo `operations` indicam ao servidor Node que o arquivo se encontra naquela mesma pasta na qual ele está sendo chamado. Se estes símbolos fossem omitidos, o Node procuraria o arquivo na pasta padrão de módulos, que veremos mais adiante.

Para ver o resultado da aplicação funcionando, digite o comando `node server.js` no console dentro da pasta raiz do projeto.

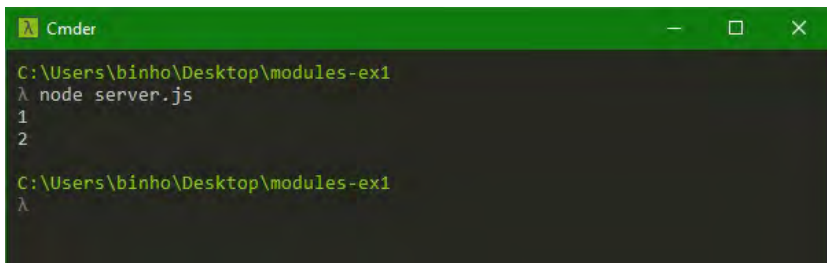


Figura 5.2: Execução de programa que carrega um módulo

Um modulo pode exportar também uma função, que, por sua

vez, pode ter *n* funções dentro de si, além de objetos JavaScript.

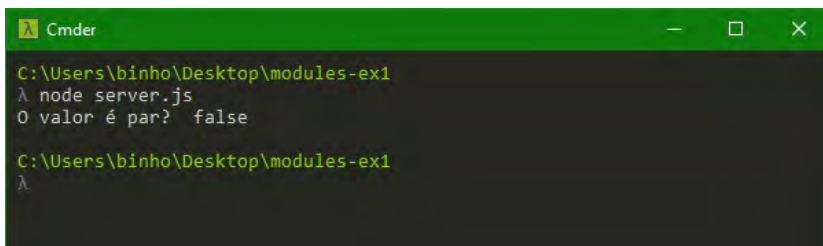
Para entender melhor, vamos continuar adicionando algumas funções no nosso projeto. Precisamos de um novo arquivo, que chamaremos de `funcoes.js`, em que vamos criar uma função que verifica se um número é par ou ímpar. A função deve retornar um objeto *JavaScript*, contendo uma mensagem e um valor verdadeiro ou falso para a execução.

```
module.exports = (x) => {  
  
  var par = () => x % 2 == 0 ? true: false;  
  
  return {  
    msg: 'O valor é par? ',  
    value: par()  
  };  
  
};
```

Falta apenas ajustar o arquivo `server.js` para utilizar a função criada no arquivo `funcoes.js`. Vamos usar a função com o parâmetro de valor **1**, e imprimir o resultado no console da aplicação.

```
const funcoes = require('./funcoes');  
  
const resultado = funcoes(1);  
console.log(resultado.msg, resultado.value);
```

Se você quiser testar a aplicação, execute no console o comando `node server.js`. O resultado deve ser semelhante ao da figura a seguir.



```
Cmder
C:\Users\binho\Desktop\modules-ex1
λ node server.js
O valor é par? false

C:\Users\binho\Desktop\modules-ex1
λ
```

Figura 5.3: Resultado da execução de server.js

Uma terceira opção de exportar módulos é utilizar um único arquivo para exportar diferentes objetos ou funções. Para entender melhor, crie um outro arquivo no projeto chamado `funcoes2.js`. Neste arquivo, vamos exportar duas outras funções: uma que soma dois valores e outra que subtrai dois valores. O arquivo `funcoes2.js` usa uma notação que permite exportar vários módulos.

```
exports.adicao = (x,y) => x + y;
exports.subtracao = (x,y) => x - y;
```

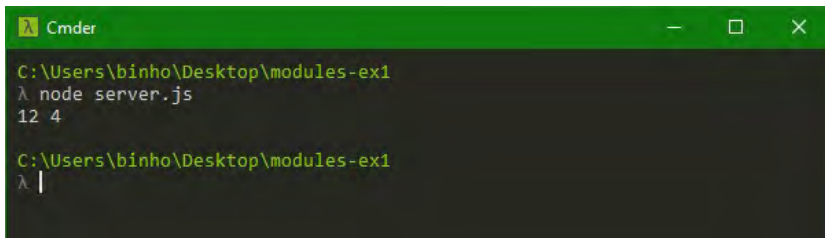
Desta forma, é possível configurar o arquivo `server.js` para usar cada uma das funções separadamente.

```
const funcoes2 = require('./funcoes2');

const add = funcoes2.adicao(5,7);
const sub = funcoes2.subtracao(7,3);

console.log(add, sub);
```

Para ver o resultado, execute no console o comando `node server.js`.



```
Cmder
C:\Users\binho\Desktop\modules-ex1
λ node server.js
12 4

C:\Users\binho\Desktop\modules-ex1
λ |
```

Figura 5.4: Execução de múltiplos módulos em um arquivo

Observe que o arquivo `server.js` apresenta a execução de cada uma das funções. O ponto (`.`) que distingue qual das funções vai ser executada define a notação correta para esse procedimento.

Todas as formas de se criar módulos são válidas e amplamente utilizadas. O uso da notação `module.exports` é mais comumente encontrado, pois ela permite encapsular objetos *JavaScript* e funções enquanto em um único ponto de entrada.

5.2 CONSIDERAÇÕES

Entender como funciona o sistema de modularização, conhecido como *CommonJS* adotado pelo Node, dá ao desenvolver toda flexibilidade disponível na plataforma, além de facilitar o entendimento do uso de módulos de terceiros, assunto de próximos capítulos.

NPM

Quando o Node.js é instalado, ele traz consigo um gerenciador de módulos, o *NPM (Node Package Manager)*. Este permite compartilhar e reutilizar módulos desenvolvidos por terceiros a partir de uma API simples, acessada por um *CLI*.

No momento da escrita deste livro, existiam mais de **385.000 módulos** no repositório do NPM. Para acompanhar os números de perto deste e de outros gerenciadores de módulos (ou *packages*), acesse <http://www.modulecounts.com/>.

Neste livro, serão vistas as principais rotinas de uso do NPM para gerenciar projetos desenvolvidos com Node. Partindo do princípio que o NPM trabalha com importação de pacotes de um repositório, é preciso conhecer os principais comandos para executar tarefas de rotina, como por exemplo, instalar um pacote.

Para entender como tudo funciona, vamos fazer a importação e utilização de alguns pacotes (conhecidos como *modules* ou *packages*).

6.1 PACKAGE.JSON

Antes de utilizar o NPM, pode-se criar um arquivo com o

nome de `package.json` dentro do projeto. Esse arquivo será responsável por conter informações técnicas sobre o projeto em questão, informações sobre comandos de execução e depuração do projeto e sobre os módulos dos quais o projeto depende para funcionar.

Esse arquivo não é obrigatório para se trabalhar com pacotes. Entretanto, é muito interessante possuir um no projeto, pois é ele quem vai conter as principais configurações de execução.

Ao levar o projeto de um lugar para o outro, não é preciso transportar junto todos os módulos de terceiros para o correto funcionamento da aplicação, pois a qualquer momento é possível restaurar os módulos a partir de um comando do NPM, que vai acessar o `package.json` para saber quais módulos carregar.

Para simplificar o entendimento, vamos inicializar um arquivo `package.json`. Começaremos criando uma nova pasta chamada `npm-modules`.

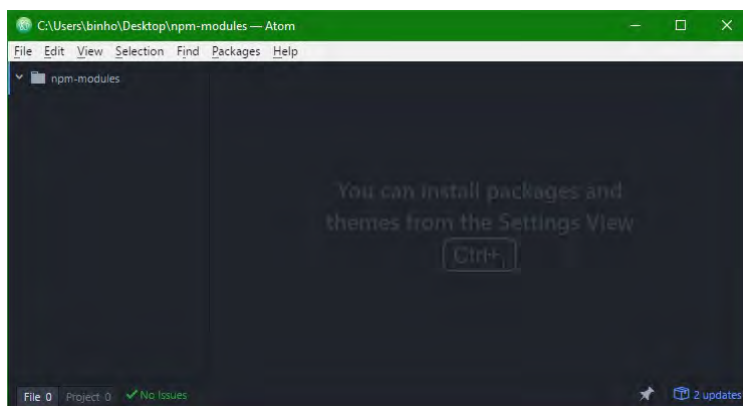
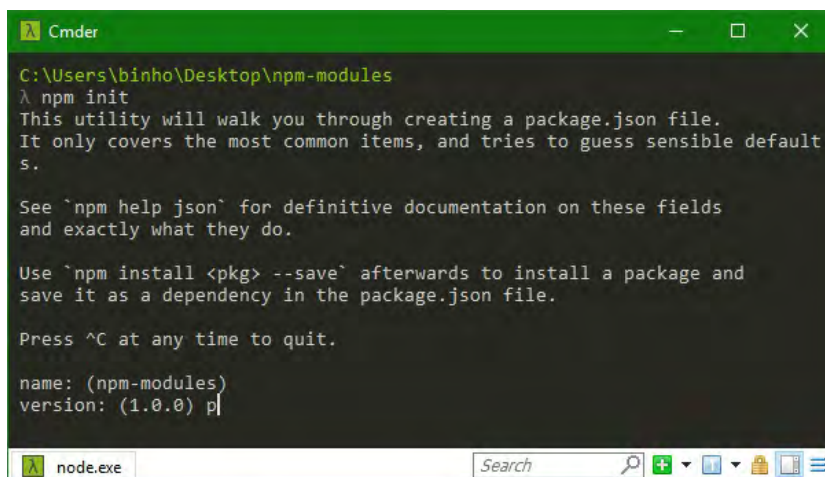


Figura 6.1: Novo projeto npm modules

Agora vamos configurar o nosso projeto Node. No terminal, dentro da pasta do projeto, digite o comando `npm init`. Isso fará com que o arquivo `package.json` seja criado, e iniciará um assistente de configuração.



```
Cmder
C:\Users\binho\Desktop\npm-modules
λ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible default
s.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

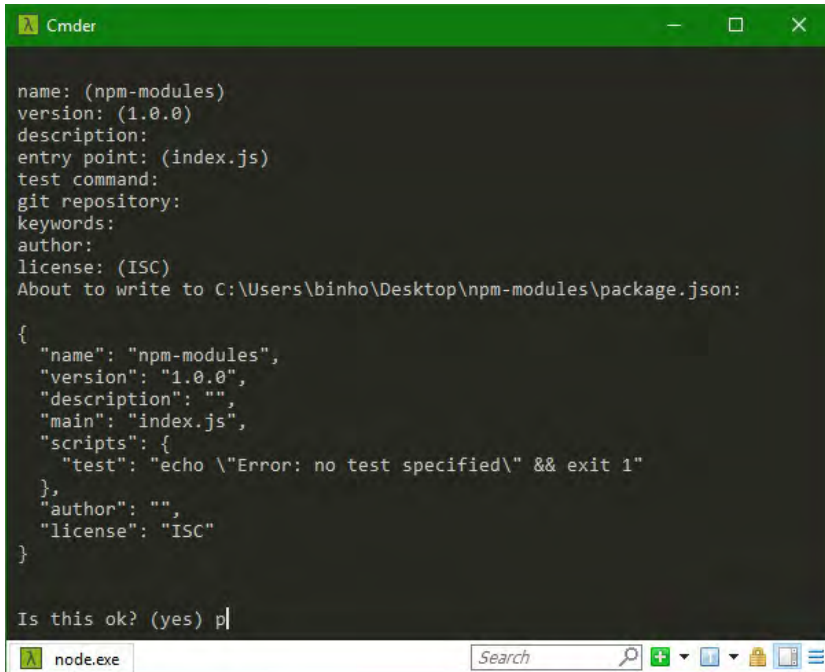
Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.

name: (npm-modules)
version: (1.0.0) p|
```

Figura 6.2: Comando `npm init`

O assistente apresentará perguntas sobre a versão e nome do projeto, dados do autor e descrição do projeto. Vamos pular essas etapas apertando `enter`, pois elas não são essenciais para o funcionamento do projeto.



```
name: (npm-modules)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\binho\Desktop\npm-modules\package.json:

{
  "name": "npm-modules",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this ok? (yes) y|
```

Figura 6.3: Configuração do npm init

Para concluir, digitamos `y` (*yes*).

Você pode navegar até a pasta do projeto no editor para ver o arquivo criado.

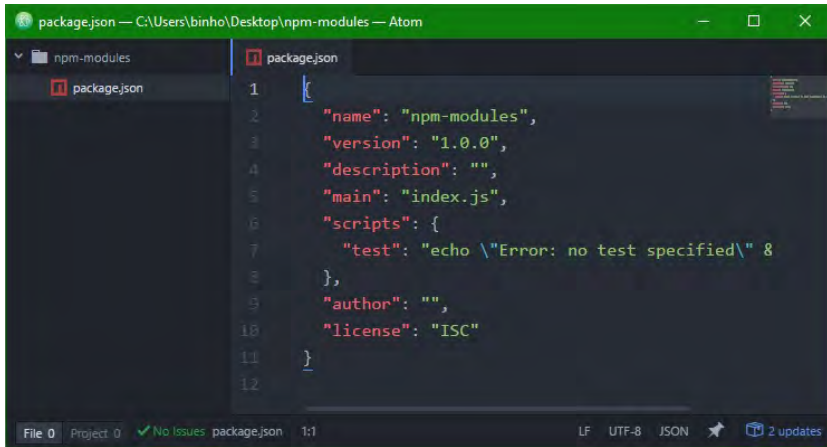


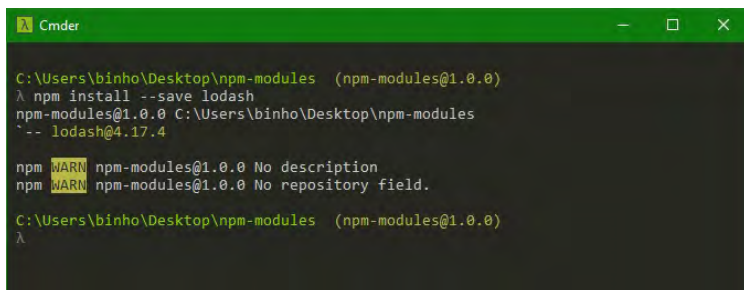
Figura 6.4: Estrutura do arquivo package.json

Agora já é possível carregar módulos, guardando as informações deles no `package.json`.

6.2 LODASH

Como exemplo de módulo, vamos conhecer o **Lodash**. O Lodash é uma biblioteca bem comum de funções utilitárias para *JavaScript*.

Para realizarmos um teste, precisamos adicionar o `lodash` como módulo no nosso projeto. Isso se faz a partir do comando: `npm install --save lodash`. Este comando instrui o NPM a fazer o download de todos arquivos que compõe o módulo `lodash`. Caso o `lodash` seja dependente de outros módulos de terceiros, esses também serão importados recursivamente.



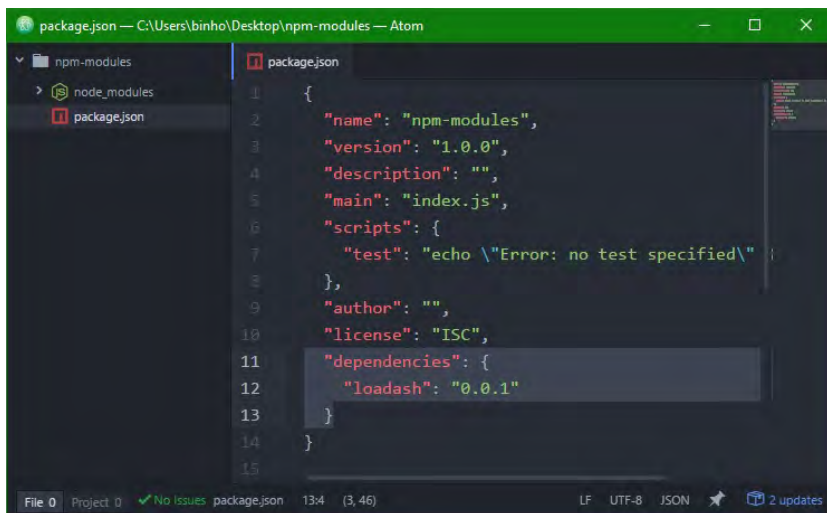
```
C:\Users\binho\Desktop\npm-modules (npm-modules@1.0.0)
λ npm install --save lodash
npm-modules@1.0.0 C:\Users\binho\Desktop\npm-modules
`-- lodash@4.17.4

npm WARN npm-modules@1.0.0 No description
npm WARN npm-modules@1.0.0 No repository field.

C:\Users\binho\Desktop\npm-modules (npm-modules@1.0.0)
λ
```

Figura 6.5: Importação do módulo lodash

Observe que o comando de importação recebeu uma flag `--save`, que indica que, além de importar para o projeto o novo módulo, o NPM deve salvar a entrada do projeto no arquivo `package.json`, como visto na figura seguinte.



```
package.json — C:\Users\binho\Desktop\npm-modules — Atom
package.json
1 {
2   "name": "npm-modules",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\"",
8   },
9   "author": "",
10  "license": "ISC",
11  "dependencies": {
12    "lodash": "0.0.1"
13  }
14 }
15
```

Figura 6.6: Arquivo package.json atualizado

O nome `lodash` agora aparece em uma nova seção (`dependencies`), seguido da versão do módulo. O NPM trabalha

com um sistema de versionamento bem simples, que você pode analisar com mais calma, visitando o site <https://docs.npmjs.com/files/package.json>.

Outro ponto a destacar neste procedimento foi a criação de uma pasta chamada `node_modules`. Esta é a pasta padrão para qual o NPM vai transferir todos os módulos importados pelo comando `npm install`.

Agora que já importamos o `lodash`, podemos começar a usá-lo. Vamos precisar de um arquivo `index.js` para realizarmos alguns testes. Vamos usar uma função do `lodash` que quebra em partes um array, permitindo, por exemplo, personalizar sua impressão.

Usando a função `forEach()` do *JavaScript*, faremos com que o programa imprima os valores do array, separando-os em partes de três em três:

```
const _ = require('lodash');

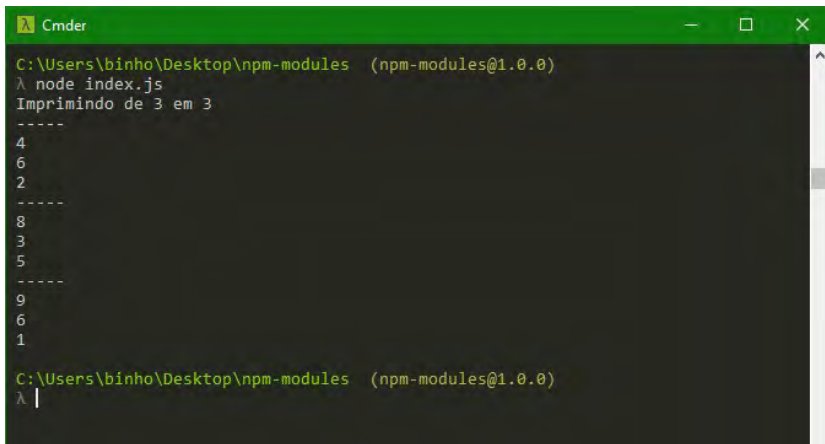
const array = [4,6,2,8,3,5,9,6,1];

console.log('Imprimindo de 3 em 3');

_.chunk(array,3).forEach((item)=>{

    console.log('-----');
    item.forEach((i)=>{
        console.log(i);
    })
});
```

Precisamos ver o resultado para nos certificarmos de que tudo ocorreu bem. Para isso, digitamos o comando `node index.js` no terminal.



```
C:\Users\binho\Desktop\npm-modules (npm-modules@1.0.0)
λ node index.js
Imprimindo de 3 em 3
-----
4
6
2
-----
8
3
5
-----
9
6
1
C:\Users\binho\Desktop\npm-modules (npm-modules@1.0.0)
λ |
```

Figura 6.7: Resultado do projeto com lodash

Com o `lodash`, é possível remover valores falsos e nulos de um array (função `compact`), remover uma lista de valores de um determinado array (função `pull`), recortar trechos de um array (função `take`) etc.

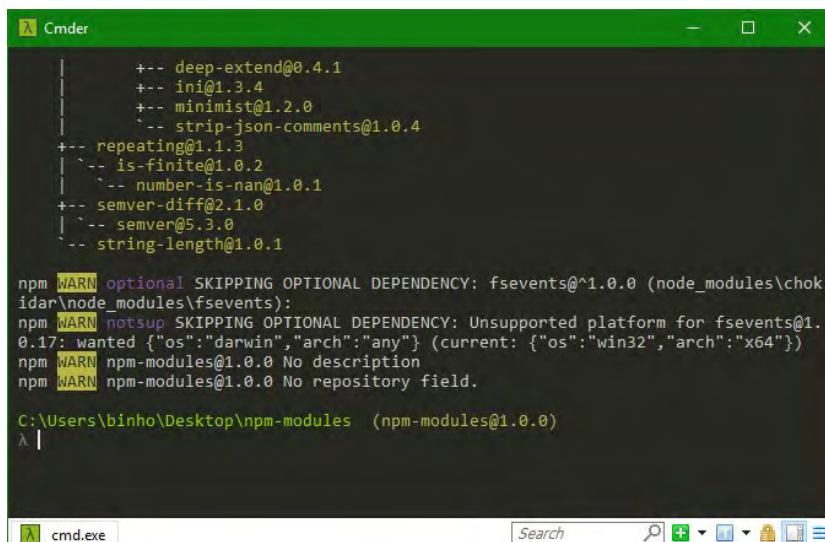
Para conhecer melhor as funções utilitárias do `lodash`, acesse <https://lodash.com/docs/4.17.4>.

6.3 NODEMON MODULE

Outro módulo muito comum entre os desenvolvedores de Node.js é o `nodemon`. Ele funciona como um *watcher*, isto é, ele fica observando toda vez que um arquivo do seu projeto é alterado. Quando ocorre uma alteração de arquivo, o `nodemon` reinicia o processo do Node e a aplicação está pronta novamente, evitando ter de ficar executando todo momento o comando `node`.

Vamos inserir o `nodemon` no projeto corrente, e realizar

alguns testes. No terminal, precisamos baixar o pacote do nodemon e adicionar sua entrada no arquivo `package.json`, a partir do comando `npm install --save nodemon`.



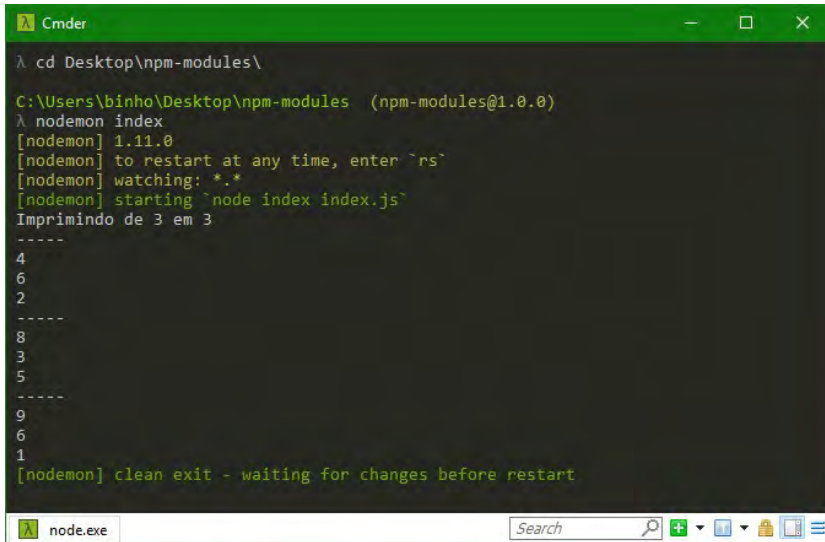
```
C:\Users\b\ino\Desktop\npm-modules (npm-modules@1.0.0)
λ npm install --save nodemon
+-- deep-extend@0.4.1
+-- ini@1.3.4
+-- minimist@1.2.0
+-- strip-json-comments@1.0.4
+-- repeating@1.1.3
+-- is-finite@1.0.2
+-- number-is-nan@1.0.1
+-- semver-diff@2.1.0
+-- semver@5.3.0
+-- string-length@1.0.1

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@^1.0.0 (node_modules\chokidar\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.0.17: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
npm WARN npm-modules@1.0.0 No description
npm WARN npm-modules@1.0.0 No repository field.

C:\Users\b\ino\Desktop\npm-modules (npm-modules@1.0.0)
λ
```

Figura 6.8: Instalação do nodemon

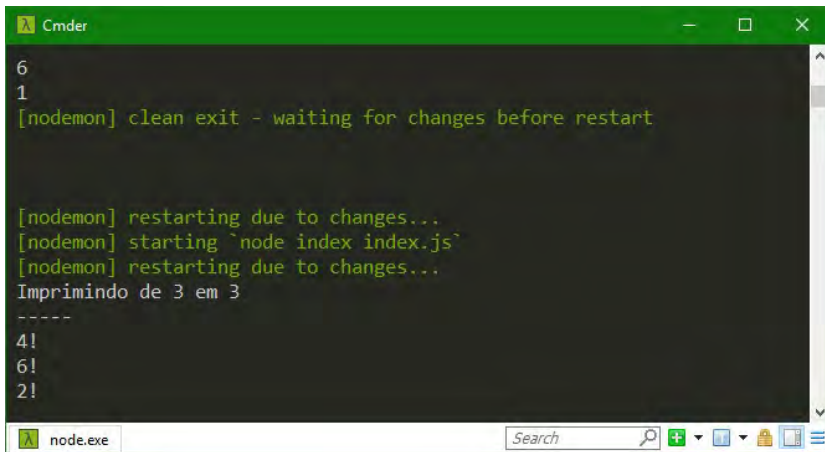
Em seguida, usamos o comando `nodemon index` para executar o módulo, fazendo com que o recarregamento da aplicação aconteça a cada alteração salva nos arquivos.



```
λ cd Desktop\npm-modules\
C:\Users\binho\Desktop\npm-modules (npm-modules@1.0.0)
λ nodemon index
[nodemon] 1.11.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index index.js`
Imprimindo de 3 em 3
-----
4
6
2
-----
8
3
5
-----
9
6
1
[nodemon] clean exit - waiting for changes before restart
```

Figura 6.9: Programa funcionando com nodemon

Observe que, após a execução do arquivo, uma mensagem do tipo *waiting for changes...* indica que, a cada alteração nos arquivos, o programa é executado novamente. Vamos alterar a linha de impressão de dados no console para `console.log(i + "!");`. Observe o resultado no console da aplicação, pois o `nodemon` reinicia o processo e executa a tarefa.



```
6
1
[nodemon] clean exit - waiting for changes before restart

[nodemon] restarting due to changes...
[nodemon] starting `node index index.js`
[nodemon] restarting due to changes...
Imprimindo de 3 em 3
-----
4!
6!
2!
```

Figura 6.10: Nodemon reiniciando processo após salvamento de arquivo

6.4 CONSIDERAÇÕES

Existem diversos comandos e opções trazidas junto com o NPM. Porém, aqui foram apresentadas as principais delas. No decorrer do livro, outras serão apresentadas conforme surgem as necessidades.

O sistema de *packages* (*modules*) é um dos grandes diferenciais da plataforma que envolve o Node.js. O site <https://www.npmjs.com/> permite buscar rapidamente por novos módulos, além de consultar a documentação de uso de cada um deles.

ECMAScript 2015 – ES6

O *ECMAScript 2015* (ou *ES6*, como é mais conhecido) é o padrão de linguagem de programação implementado pela nova versão do *JavaScript*. Ele adiciona uma série de características ao *JavaScript* que o tornam mais eficiente em sua execução, além de mais flexível em sua codificação.

Porém, nem todas as características que este novo padrão implementa estão presentes nas implementações do Node.js para a linguagem. Até os próprios navegadores ainda não implementaram todas as novidades. Isso significa que alguns recursos da nova versão podem não funcionar nos navegadores, ou no servidor, quando usamos Node.js.

O uso de definições de variáveis (`const` , `let` , no lugar de `var`) e o uso de `arrow functions` são exemplos de novas funcionalidades já implementadas no Node. Mas o uso de `import` para carregar módulos, por exemplo, até a data de escrita deste livro, ainda não era suportado. Você pode acompanhar o andamento das implementações do *ES6* em <http://node.green/>, bem como o planejamento para as próximas versões do Node.js.

Isso significa que, para poder usar todos recursos do *ES6*, é necessário utilizar um módulo de **transpiler**. Um transpiler é um

modulo que transcreve o código-fonte de uma aplicação para uma versão qualquer definida em suas configurações. Desse modo, é possível escrever código-fonte com todas as características do *ES6*, e ainda assim permitir que ele funcione em todos os navegadores, ou no caso, no Node.js.

Um dos transpilers mais conhecidos do mercado para essa tarefa é o **Babel** (<https://babeljs.io/>). O próprio site já apresenta alguns casos de uso que mostram como ele funciona.

Vamos criar um novo projeto para entender como usar o **Babel** junto com o Node, para desenvolver aplicações no padrão *ES6*. Precisamos criar uma nova pasta, que chamaremos de `es6` e, dentro dela, um novo arquivo para configurar nosso servidor, chamado `server.js`.

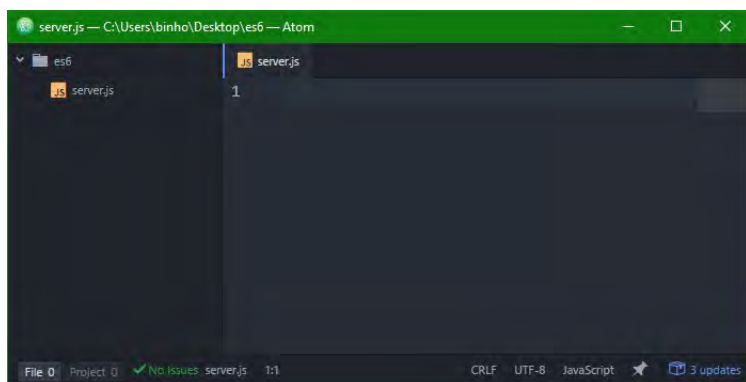
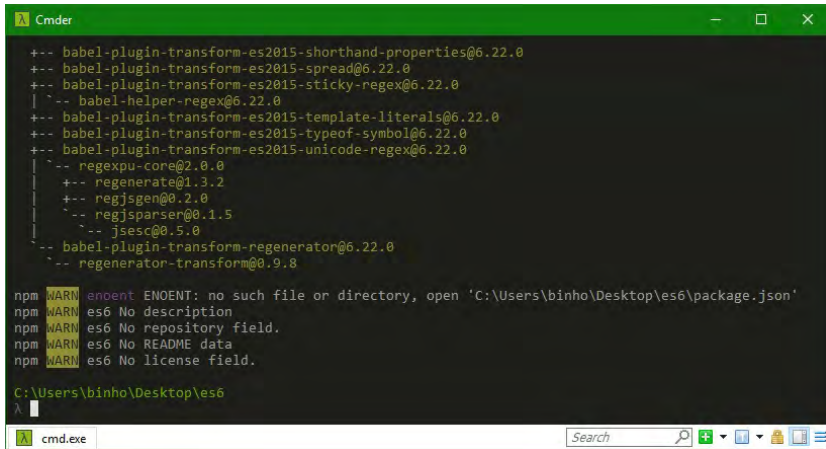


Figura 7.1: Criação de um novo projeto

Para utilizarmos o `babel` em parceria com o Node, importamos alguns módulos de configuração, a partir do comando: `npm install babel-register babel-preset-es2015`.



```
cmd.exe
+-- babel-plugin-transform-es2015-shorthand-properties@6.22.0
+-- babel-plugin-transform-es2015-spread@6.22.0
+-- babel-plugin-transform-es2015-sticky-regex@6.22.0
|   -- babel-helper-regex@6.22.0
|       -- babel-plugin-transform-es2015-template-literals@6.22.0
+-- babel-plugin-transform-es2015-typeof-symbol@6.22.0
+-- babel-plugin-transform-es2015-unicode-regex@6.22.0
|   -- regenerator-core@2.0.0
|       +-- regenerate@1.3.2
|       +-- regjsgen@0.2.0
|       -- regjsparser@0.1.5
|           -- jsc@0.5.0
+-- babel-plugin-transform-regenerator@6.22.0
    -- regenerator-transform@0.9.8

npm WARN ENOENT: no such file or directory, open 'C:\Users\binho\Desktop\es6\package.json'
npm WARN es6 No description
npm WARN es6 No repository field.
npm WARN es6 No README data
npm WARN es6 No license field.

C:\Users\binho\Desktop\es6
>
```

Figura 7.2: Instalação dos módulos babel

Vamos começar a testar a aplicação criando dois módulos exemplos (`mod1()` e `mod2()`) em um novo arquivo, que chamaremos de `modulos.js` . Nele, usaremos a nova notação do ES6 para configuração de módulos (`export function`).

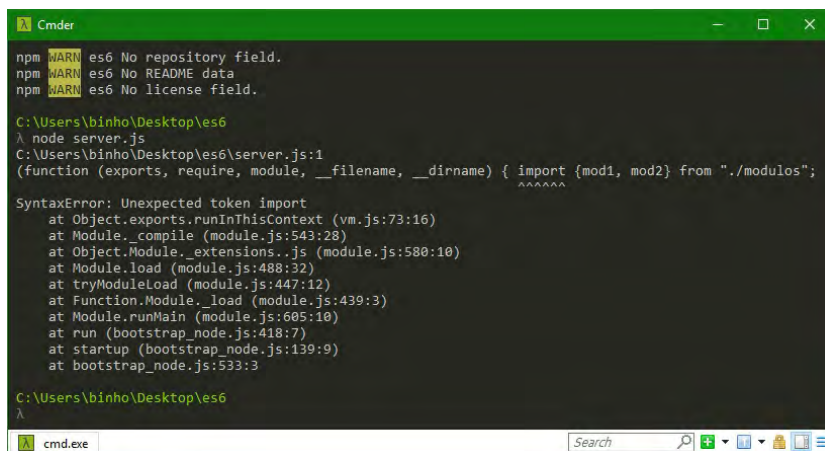
```
export function mod1() {
  console.log('Módulo 1 carregado');
}
export function mod2() {
  console.log('Módulo 2 carregado');
}
```

Agora, para importarmos e executarmos os módulos criados dentro do arquivo `server.js` , vamos usar a notação `import` . Nesta notação, é possível definir os nomes dos módulos que serão importados, como mostra o trecho de código a seguir:

```
import {mod1, mod2} from "./modulos";

mod1();
mod2();
```

Se o programa for executado da maneira que está, ocorrerá um erro, pois o node não conhece a sintaxe `import`, como mostra a figura a seguir.



```
cmd
npm WARN es6 No repository field.
npm WARN es6 No README data
npm WARN es6 No license field.

C:\Users\binho\Desktop\es6
λ node server.js
C:\Users\binho\Desktop\es6>server.js:1
(function (exports, require, module, __filename, __dirname) { import {mod1, mod2} from "../modulos";
                                                                    ^^^^^^
SyntaxError: Unexpected token import
    at Object.exports.runInThisContext (vm.js:73:16)
    at Module._compile (module.js:543:28)
    at Object.Module._extensions..js (module.js:580:10)
    at Module.load (module.js:488:32)
    at tryModuleLoad (module.js:447:12)
    at Function.Module._load (module.js:439:3)
    at Module.runMain (module.js:605:10)
    at run (bootstrap_node.js:418:7)
    at startup (bootstrap_node.js:139:9)
    at bootstrap_node.js:533:3

C:\Users\binho\Desktop\es6
λ
```

Figura 7.3: Erro na execução do servidor

Por isso precisamos configurar os módulos do `babel`. Desta forma, criamos inicialmente um arquivo chamado `.babelrc`, que conterà as configurações de transcrição da linguagem *JavaScript*. Dentro do arquivo, insira o seguinte trecho de código:

```
{
  "presets": ["es2015"]
}
```

Essa configuração garante que o código escrito em *ECMAScript 2015* seja transformado em *JavaScript* padrão, garantindo sua execução pelo servidor Node.

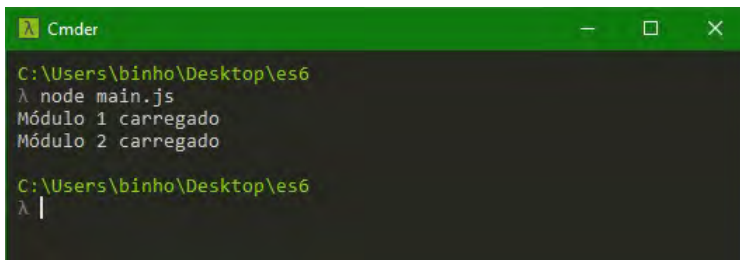
Agora, precisamos garantir que os módulos do `babel` estejam carregados antes da execução do arquivo `server.js`. Para isso, vamos criar um novo arquivo chamado `main.js`, e inserir o

código de importação `babel` e do arquivo `server.js` :

```
require("babel-register");  
require("./server");
```

Esse será o novo arquivo de execução do Node, pois ele permite carregar o módulo de transpiler do `babel` , e em seguida executar o arquivo `server.js`.

Vamos finalizar a configuração acessando o terminal e digitando o comando `node main.js` , para carregarmos a aplicação. Observe que o Node consegue, desta forma, executar normalmente nosso servidor.



```
Cmder  
C:\Users\binho\Desktop\es6  
λ node main.js  
Módulo 1 carregado  
Módulo 2 carregado  
C:\Users\binho\Desktop\es6  
λ |
```

Figura 7.4: Execução de programa com código ES6

7.1 CONSIDERAÇÕES

Com o advento do *ES6*, diversas tecnologias começam a se adaptar para suportar todos os recursos da especificação, enquanto que a comunidade se adapta em adotar os novos padrões da linguagem. Apesar disso, é comum encontrar diversos desenvolvedores em Node.js utilizando apenas as características já implementadas, por conta de evitar os processos de *transpiler*.

Neste livro, serão sempre usadas as sintaxes suportadas por padrão no Node.js. Apesar disso, é possível que você queira manter

o seu código atualizado com as novas funcionalidades da linguagem *JavaScript*, utilizando um *transpiler* (como o *Babel*), sem interferir diretamente na execução dos exemplos aqui apresentados.

WEB SERVER COM EXPRESS

Anteriormente, um servidor web foi criado para responder chamadas na porta 3000, utilizando-se do módulo `http`, nativo do Node.js. Desta mesma forma, é possível criar um servidor web completo para tratar de questões específicas, como permitir o retorno de páginas web estáticas (ou dinâmicas), lidar com autenticação e autorização, acessar banco de dados, fazer cache de sessões etc.

Todas essas questões podem ser resolvidas por um bom programador *JavaScript*, com conhecimentos avançados de web. Mas a ideia por trás da tecnologia Node é, em contrapartida, utilizar módulos que resolvem cada um destes problemas, acoplando-os à aplicação que está sendo desenvolvida.

Deste modo, o programador não se preocupa em resolver problemas que fogem da lógica do negócio de sua aplicação, ao passo que consegue acessar o código-fonte de cada módulo para entender seu funcionamento, ou acessar suas documentações, geralmente disponíveis em um repositório do GitHub.

Um dos módulos mais famosos para o desenvolvimento de

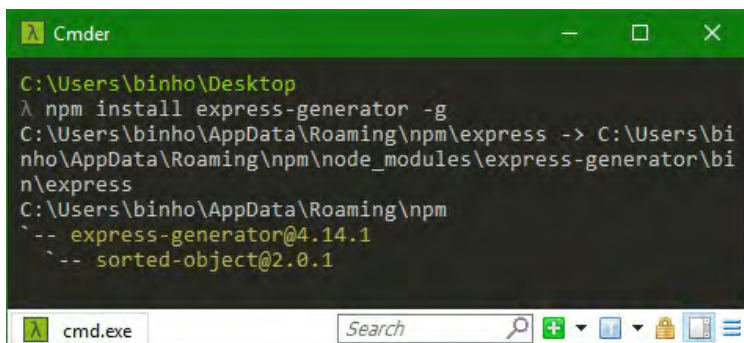
servidores web é o *Express*. Ele trata de uma maneira simplista o sistema de rotas das páginas (o conceito de rotas será melhor apresentado mais adiante), e possui diversos módulos que podem ser acoplados em si conforme a necessidade da aplicação.

Neste capítulo, vamos entender melhor o funcionamento do *Express*, criando um servidor web.

8.1 CONFIGURANDO O SERVIDOR COM EXPRESS

O *Express* pode ser instalado a partir de um gerador, que cria rapidamente uma estrutura de servidor web básica, baixando diversos módulos e definindo a configuração deles. Vamos usar esse gerador a fim de simplificar a criação e a explicação de um servidor web com o *Express*. Depois disso, vou detalhar cada parte da aplicação gerada, para que possamos entender o seu funcionamento como um todo.

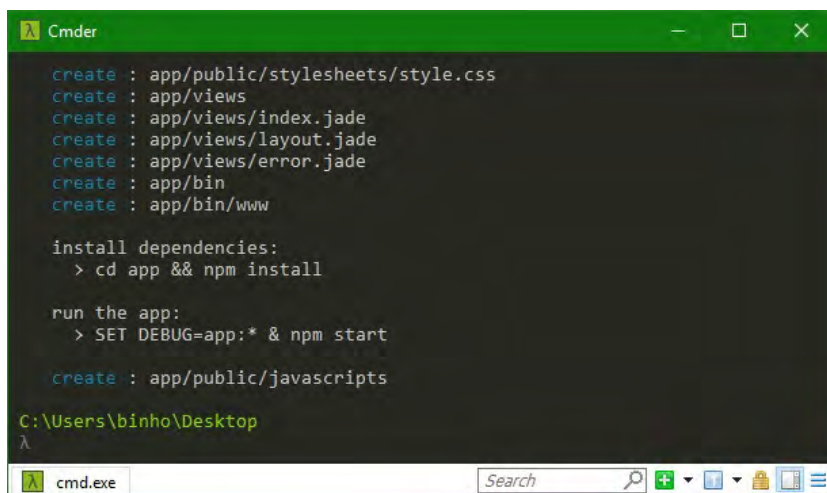
Precisamos, inicialmente, instalar o gerador do *Express*, que é um módulo. No terminal, o comando `npm install express-generator -g` faz essa instalação. Observe que desta vez utilizamos uma flag `-g`, que indica que o módulo gerador será instalado globalmente. Quando isso ocorre, o módulo pode ser acessado de um repositório local (o Node.js cria uma pasta para guardar esses módulos), sem a necessidade de se fazer o download para cada novo projeto.



```
C:\Users\binho\Desktop
λ npm install express-generator -g
C:\Users\binho\AppData\Roaming\npm\express -> C:\Users\binho\AppData\Roaming\npm\node_modules\express-generator\bin\express
C:\Users\binho\AppData\Roaming\npm
^-- express-generator@4.14.1
^-- sorted-object@2.0.1
```

Figura 8.1: Instalação do gerador do Express

Em seguida, precisamos acessar o diretório no qual queremos criar nosso servidor web, e digitar o comando `express app`. Neste caso, `app` representa o nome do diretório principal que será criado pelo gerador.



```
create : app/public/stylesheets/style.css
create : app/views
create : app/views/index.jade
create : app/views/layout.jade
create : app/views/error.jade
create : app/bin
create : app/bin/www

install dependencies:
> cd app && npm install

run the app:
> SET DEBUG=app:* & npm start

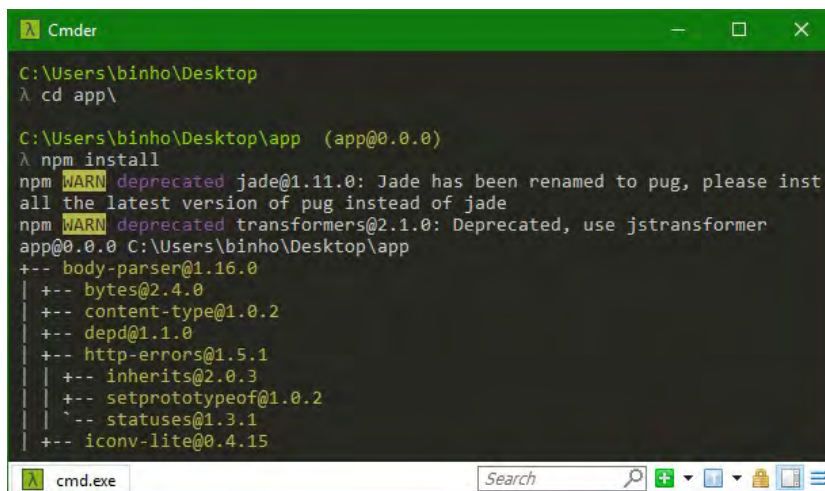
create : app/public/javascripts

C:\Users\binho\Desktop
λ
```

Figura 8.2: Criação de uma aplicação com gerador Express

Agora, nos resta instalar todas as dependências usadas pela

aplicação gerada. Você pode observar que a indicação das dependências encontra-se no arquivo `package.json`, mas a pasta `node_modules` ainda não existe. Por isso, precisamos executar o comando `npm install`, que fará o download de todas as dependências do projeto.



```
C:\Users\binho\Desktop
λ cd app\

C:\Users\binho\Desktop\app (app@0.0.0)
λ npm install
npm WARN deprecated jade@1.11.0: Jade has been renamed to pug, please inst
all the latest version of pug instead of jade
npm WARN deprecated transformers@2.1.0: Deprecated, use jstransformer
app@0.0.0 C:\Users\binho\Desktop\app
+-- body-parser@1.16.0
| +-- bytes@2.4.0
| +-- content-type@1.0.2
| +-- depd@1.1.0
| +-- http-errors@1.5.1
| | +-- inherits@2.0.3
| | +-- setprototypeof@1.0.2
| | ^-- statuses@1.3.1
| +-- iconv-lite@0.4.15
```

Figura 8.3: Instalando módulos da aplicação

Neste momento, nossa aplicação já está pronta para ser executada. Vamos testar executando o comando `npm start`. Note que não inserimos o comando `node`, e mesmo assim a aplicação funcionou!

Isso ocorre porque o comando `npm start` executa um *script* pré-configurado no arquivo `package.json`. O *script* faz com que o comando `node ./bin/www` seja executado, já que o arquivo de entrada da aplicação criada está na pasta `bin`.

Vamos ver o resultado no navegador, digitando o endereço

localhost:3000 .

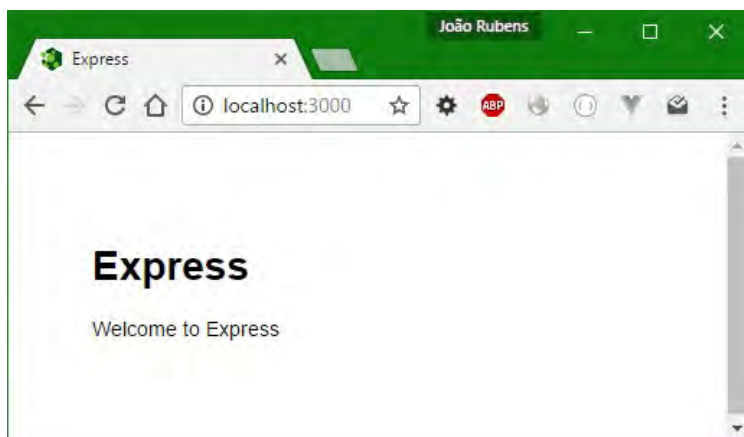


Figura 8.4: Teste do servidor Express

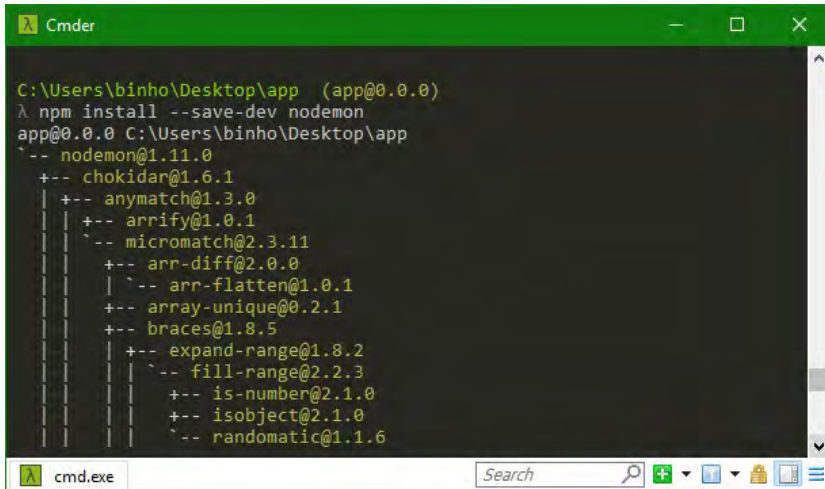
A mensagem apresentada no navegador indica que o servidor já está funcionando, e que todas as dependências do projeto gerado estão configuradas!

8.2 NODEMON – WATCHER DA APLICAÇÃO

Vamos começar a personalizar a nossa aplicação adicionando o módulo `nodemon` (visto anteriormente), para evitar o transtorno de precisar recarregar o servidor toda vez que uma alteração é feita. Caso sua aplicação estiver sendo executada, termine a execução (no Windows, `ctrl+c`).

Agora precisamos, ainda dentro do terminal, inserir o comando que instala o `nodemon` no projeto: `npm install --save-dev nodemon` . A flag `--save-dev` permite adicionar a entrada do módulo no arquivo `package.json` , em uma seção

separada de dependências. Em um ambiente de produção, o nodemon não precisa ser executado, portanto, os pacotes da seção `devDependencies` não serão carregados.



```
C:\Users\binho\Desktop\app (app@0.0.0)
λ npm install --save-dev nodemon
app@0.0.0 C:\Users\binho\Desktop\app
-- nodemon@1.11.0
+-- chokidar@1.6.1
| +-- anymatch@1.3.0
| | +-- arrify@1.0.1
| | `-- micromatch@2.3.11
| |   +-- arr-diff@2.0.0
| |   | `-- arr-flatten@1.0.1
| |   +-- array-unique@0.2.1
| |   +-- braces@1.8.5
| |   | +-- expand-range@1.8.2
| |   | | `-- fill-range@2.2.3
| |   | |   +-- is-number@2.1.0
| |   | |   +-- isobject@2.1.0
| |   | |   `-- randomatic@1.1.6
```

Figura 8.5: Instalação do módulo nodemon

Outra alteração que vamos fazer é a mudança do arquivo de entrada da aplicação para o diretório principal com o nome de `server.js`, já que essa é a configuração mais comumente usada pela comunidade. Portanto, abra o arquivo `package.json`, e troque o comando dentro da entrada `script > start` pelo comando `node ./server`.

Isso fará com que o arquivo `server.js` se torne a porta de entrada da aplicação. Em seguida, vamos retirar o arquivo `www` que está dentro da pasta `bin`, e colocá-lo na pasta raiz do projeto, renomeando-o para `server.js`.

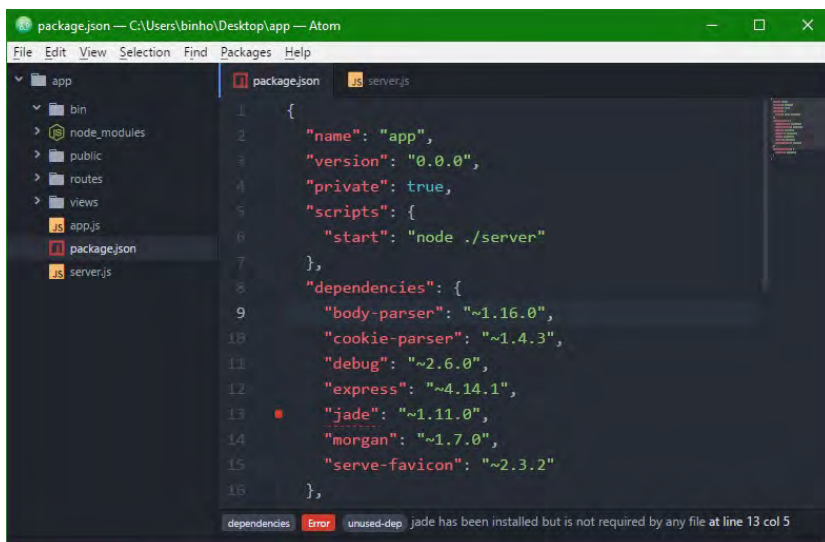


Figura 8.6: Alterações no caminho padrão do arquivo de entrada do servidor web

Para concluir esta etapa, precisamos modificar a primeira linha do arquivo `server.js` de `var app = require('../app');` para `var app = require('./app');`. Isso é necessário porque o arquivo `server.js` agora se encontra no diretório principal.

Agora já é possível verificar o funcionamento do servidor novamente, desta vez, dentro da pasta do projeto no terminal, digitando apenas o comando `nodemon`.



Figura 8.7: Servidor executando a partir de nodemon

8.3 ESTRUTURA DA APLICAÇÃO

Como visto nas seções anteriores, o gerador do *Express* traz uma série de recursos já configurados para um servidor web executar. É claro que, conforme os requisitos da aplicação são implementados, novas adições ou remoções de recursos serão realizadas. Mas o projeto gerado já é um excelente ponto de partida.

Para ficar claro tudo o que aconteceu com a geração do projeto, vamos dar uma olhada nos arquivos criados, começando pelo arquivo que acabamos de modificar, o `server.js`. O arquivo começa com o carregamento de uma série de módulos que configuram a aplicação. O primeiro é onde se encontram as especificações do servidor web que vai ser criado (`var app = require('./app')`).

O segundo permite visualizar erros de uma forma mais fácil no console (`var debug = require('debug')('app:server')`). Ele faz com que os erros apareçam com cores diferentes, facilitando assim o rastreamento da pilha de erros. O próximo (`var http = require('http')`) faz a requisição do módulo que permite a criação de uma aplicação na porta especificada, esperando por requisições HTTP.

A variável `port` recebe a configuração da porta de execução, que será selecionada a partir da variável de ambiente `PORT` ou, caso não haja uma configuração para ela, a porta padrão **3000** (`var port = normalizePort(process.env.PORT || '3000')`).

O comando `app.set('port', port)` configura a aplicação

com a porta escolhida. Esse processo é interessante para um ambiente de produção, em que se deseja configurar uma porta a partir de uma variável de *runtime*, e não gerar mudanças no código da aplicação. Para saber mais sobre como funcionam as variáveis de ambiente do Node.js, acesse: https://nodejs.org/api/process.html#process_process_env.

O próximo passo utiliza o módulo `http` para criar um servidor com as configurações do módulo `app` (`var server = http.createServer(app)`), que é onde está definida a lógica da aplicação, vista mais adiante. Em seguida, o servidor é iniciado, escutando na porta definida anteriormente (`server.listen(port)`), como visto no capítulo *Iniciando um servidor web*.

Em seguida, a aplicação recebe a configuração de uma função (`server.on('error', onError)`) para tratar de erros que ocorram na aplicação (função definida mais adiante). Esse erro não serve de nada para o usuário da aplicação, mas funciona como um erro de depuração do lado do servidor.

A instrução `server.on('listening', onListening)` configura a aplicação para executar ações quando um evento de *listening* é disparado para o servidor `http`. Esse evento ocorre toda vez que o servidor é acionado, como por exemplo um navegador solicitando uma página.

A função `normalizePort()` garante que os dados que definem a porta da aplicação sejam válidos. Para isso, ela retorna um valor qualquer, caso este valor não possa ser convertido em um número inteiro, ou um número inteiro que corresponde ao valor recebido pela função. Se o valor for um número inteiro negativo, a

função retorna `false` , indicando que a porta não é válida.

```
function normalizePort(val) {
  var port = parseInt(val, 10);
  if (isNaN(port)) {
    return val;
  }
  if (port >= 0) {
    return port;
  }
  return false;
}
```

A função `onError(error)` que é executada em `server.on('error', onError)` é acionada toda vez que ocorre um erro na aplicação. Um acesso à página errada não é um erro de aplicação. Um erro de sistema pode, por exemplo, gerar um erro de aplicação (estouro de memória, acesso a base de dados inválido etc.).

Nessa função, os tipos de erros *EACCES* e *EADDRINUSE* são tratados para aparecerem formatados no console da aplicação. Eles ocorrem quando o usuário do sistema não tem permissões de iniciar o servidor, ou se um servidor está tentando ser iniciado em uma porta inválida ou já utilizada. Para realizar um teste simples destes erros, basta abrir dois terminais e tentar iniciar dois servidores na mesma porta.

```
function onError(error) {
  if (error.syscall !== 'listen') {
    throw error;
  }
  var bind = typeof port === 'string'
    ? 'Pipe ' + port
    : 'Port ' + port;
  switch (error.code) {
    case 'EACCES':
      console.error(bind + ' requires elevated privileges');
      process.exit(1);
```

```

        break;
    case 'EADDRINUSE':
        console.error(bind + ' is already in use');
        process.exit(1);
        break;
    default:
        throw error;
    }
}

```

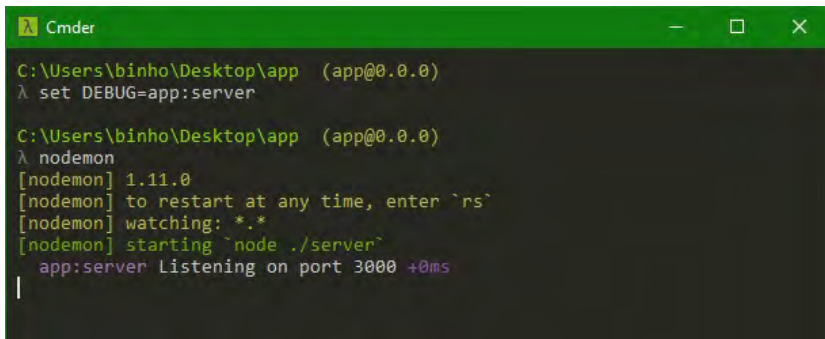
A função `onListening()` faz com que, quando um evento de `http` ocorrer, sejam ativadas as mensagens de *debug* formatadas para aparecerem no console, facilitando a depuração da aplicação. Para que isso ocorra, é necessário definir uma variável de ambiente, indicando que o modo de *debug* está ativado.

```

function onListening() {
    var addr = server.address();
    var bind = typeof addr === 'string'
        ? 'pipe ' + addr
        : 'port ' + addr.port;
    debug('Listening on ' + bind);
}

```

Vamos executar o modo de debug na prática! Utilizando um terminal no Windows (essa configuração varia em sistemas operacionais diferentes), precisamos inicialmente acessar a pasta da aplicação. Em seguida, inserimos o comando `set DEBUG=app:server`, que vai configurar a variável ambiente `DEBUG` para acionar o recurso de depuração apresentado na aplicação. Para realizarmos o teste, digitamos o comando `nodemon`.



```
C:\Users\binho\Desktop\app (app@0.0.0)
λ set DEBUG=app:server

C:\Users\binho\Desktop\app (app@0.0.0)
λ nodemon
[nodemon] 1.11.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node ./server`
app:server Listening on port 3000 +0ms
|
```

Figura 8.8: Configurando servidor para modo debug

Neste momento, a aplicação pode apresentar todas as mensagens de erros associadas ao módulo `debug` para facilitar sua depuração. Para encerrar esse modo, basta fechar e abrir o terminal novamente.

8.4 APP.JS

No arquivo `server.js`, estão todas as configurações que detalham como nossa aplicação deve se comportar como serviço, ou seja, quais opções de acesso ela disponibiliza como servidor *http*. Já no arquivo `app.js`, se encontram todos os módulos que, juntamente com o *Express*, farão o servidor responder de acordo com a lógica que a aplicação necessita.

O módulo carregado na variável `express` é o módulo central da aplicação. É ele quem define o comportamento das rotas da aplicação, configura o uso de *cookies* e *sessões*, define os *assets* que podem ser retornados pelo servidor (como arquivos `js` e `css`), dentre outras configurações (`var express = require('express')`).

Para realizar todas essas configurações, o módulo *express* utiliza as funções `set` e `use` para acoplar outros módulos, que apresentarão as regras de funcionamento de cada uma destas configurações. Por isso o *Express* tem suas vantagens, já que fornece um ambiente modular que permite que diversos desenvolvedores implementem suas configurações.

O gerador do *Express* já traz consigo várias implementações que, juntas, permitem a criação de um servidor *http* funcional. Vamos entender o básico desses módulos auxiliares.

O módulo *path* (`var path = require('path')`) define o caminho principal da aplicação, permitindo o uso de caminhos relativos, tanto em sistemas Windows quanto em POSIX. A instrução `var favicon = require('serve-favicon')` permite inserir um arquivo de ícone da página padrão para o servidor.

O módulo *Morgan* (`var logger = require('morgan')`) permite fazer log de requisições *http*. Esse recurso é muito interessante quando se deseja rastrear as ações de um usuário visitando as páginas web, por exemplo.

A instrução `var cookieParser = require('cookie-parser')` habilita o servidor a trabalhar com *cookies* para armazenar valores. Os *cookies* são muito importantes para manter dados em uma comunicação cliente-servidor, como é o caso, por exemplo, de um sistema de autenticação.

Para que o *Express* seja capaz de receber requisições *http* que trazem dados de um formulário, ele recebe as instruções desta configuração com o módulo *body-parser* (`var bodyParser = require('body-parser')`).

O *Express* implementa um sofisticado sistema de rotas, que consegue diferenciar o tipo de acesso ao servidor (`get` , `post` , `put` , `delete` etc.), além de fornecer objetos que tratem dos dados da requisição e da resposta para chamadas *http*. Uma técnica muito comum em aplicações *Express* consiste em separar as rotas da aplicação em arquivos diferentes. As instruções `var index = require('./routes/index')` e `var users = require('./routes/users')` apontam para os arquivos de definição destas rotas.

Ao findar a seção de carregamento de módulos, o comando `var app = express()` chama a função de execução do *Express*. Então, a partir da variável associada a esta função (`app`), é possível associar cada módulo importado ao *Express*, definindo a lógica da aplicação.

Por padrão, o gerador do *Express* traz consigo a configuração de um sistema de *template engine* conhecido como *Jade* (mais detalhes sobre *template engines* serão vistos adiante). Os comandos `app.set('views', path.join(__dirname, 'views'))` e `app.set('view engine', 'jade')` dizem ao *Express* em que pasta encontrar as páginas HTML (`views`) e qual o *template engine* adotado.

A	linha	comentada
(<code>app.use(favicon(path.join(__dirname,</code>	<code>'public',</code>
	<code>'favicon.ico'))</code>	<code>))</code>

permite definir o local em que está o arquivo de *favicon*. Caso exista um definido, deve-se remover o comentário desta linha para que este apareça adequadamente.

A instrução `app.use(logger('dev'))` configura o módulo de `logger` para seguir o padrão `dev`. O módulo `logger` (que

no caso utiliza o `morgan`) vem com algumas pré-configurações, e `dev` representa uma delas, em que é possível distinguir por cores os eventos que ocorrem ao acessar o servidor a partir de uma chamada *http*. Caso essa instrução seja removida, os *logs* de acesso *http* não aparecerão mais no terminal do servidor.

Os comandos `app.use(bodyParser.json())` e `app.use(bodyParser.urlencoded({ extended: false }))` configuram a aplicação para receber requisições de formulários *HTML* e arquivos *JSON*. Caso você deseje entender melhor os parâmetros do módulo `body-parser`, você pode consultar a documentação oficial em <https://github.com/expressjs/body-parser>.

A instrução `app.use(cookieParser())` configura o *Express* para inserir no objeto `req` (objeto criado nas rotas, quando acontece uma requisição) os dados de *cookies*.

O *Express* precisa ser configurado para fazer a entrega de arquivos estáticos, como arquivos de folha de estilo, imagens e *script*. Desta forma, o comando `app.use(express.static(path.join(__dirname, 'public')))` habilita o *Express* para reconhecer o diretório em que se encontram esses arquivos (no caso, o diretório `public`), e os envia em uma solicitação *http*.

Com os módulos que definem as rotas da aplicação carregados, eles podem ser acoplados ao *Express*, a partir das funções `app.use('/', index)` e `app.use('/users', users)`. Todas as rotas que tenham relação com o caminho principal da aplicação (`/`) serão tratadas pelo módulo `index`, e aquelas que tenham relação com o caminho `/users`, pelo módulo `users`.

Caso a rota solicitada por um cliente não seja válida, é possível definir uma função que devolva um erro de acesso. Nesta aplicação, um objeto `Error` carrega uma mensagem e um código de erro, que serão repassados para uma rota que trata deste erro em específico.

```
app.use(function(req, res, next) {  
  var err = new Error('Not Found');  
  err.status = 404;  
  next(err);  
});
```

A rota que lida com erros de requisições *http* recebe o parâmetro `err`, que garante que essa rota seja invocada quando ocorre algum erro. A função de tratamento de erros definida aqui renderiza o arquivo `error` (definição feita pelo *template engine*), que tem acesso à variável `err`, para apresentar os dados em uma página web. Esse processo ajuda a entender, diretamente no navegador, o que pode ter ocorrido.

```
app.use(function(err, req, res, next) {  
  res.locals.message = err.message;  
  res.locals.error = req.app.get('env') === 'development' ? err :  
  {};  
  res.status(err.status || 500);  
  res.render('error');  
});
```

Por fim, todas as configurações realizadas no *Express* (mantidas na variável `app`) são definidas como um módulo, que pode ser reutilizado em outras partes da aplicação. Nesta, o módulo `http`, definido no arquivo `server.js`, usa o módulo `app` para configurar o servidor, fazendo com que o *Express* lide com as questões da lógica da aplicação.

8.5 PASTA BIN, PUBLIC, ROUTES E VIEWS

Na estrutura de diretórios, existem outras três pastas (a `bin` deve ser excluída, pois removemos anteriormente o arquivo presente nela). A pasta `public` deve conter os *assets* da aplicação (CSS, *JavaScript*, imagens).

A pasta `routes` está definida como padrão para armazenar os arquivos de definições de rotas, e a pasta `views`, que contém todos os arquivos de definição de páginas web da aplicação. Esses arquivos armazenam as estruturas *HTML* do projeto, e podem ser tratados por diversos *template engines*, assunto de próximos capítulos do livro.

Routes

O sistema de rotas do *Express* define o que será executado para cada chamada *http*. Uma rota define as regras de acesso da aplicação, a partir de uma *URI (Uniform Resource Identifier)*. É a partir dela, por exemplo, que é definido o conteúdo de retorno do servidor para o endereço `localhost:3000/user`.

O sistema de rotas do *Express* permite definir várias configurações, mas dentre as mais importantes estão:

- **Verbo de acesso da aplicação:** verifica se a chamada para a rota é do tipo `GET`, `POST`, `PUT`, `PATCH`, `DELETE` etc.
- **Parâmetros da rota:** permite adicionar parâmetros (variáveis) à rota, trazendo flexibilidade no controle do acesso. Para a rota `localhost:3000/user/1`, o

último parâmetro pode ser definido como o `id` do usuário, e o resultado do retorno do servidor pode ser alterado por conta desse parâmetro.

- **Middlewares:** é possível adicionar chamadas a *middlewares* quando é criada uma rota. Mais detalhes sobre *middlewares* serão vistos mais adiante neste livro.
- **Objetos `request` , `response` e `next` :** a partir da configuração da rota, é possível ter acesso aos objetos de dados para a requisição ao servidor e retorno de dados (`request` , `response`). Além desses, o objeto `next` permite que a aplicação Express continue procurando por novas rotas de acesso em vez de retornar a chamada para o cliente.

Um módulo interno do Express permite configurar as rotas de uma maneira bem simples. Observe o código apresentado no arquivo `index.js` , que está dentro da pasta `routes` :

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

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

module.exports = router;
```

A partir da variável `router` , é possível definir quais rotas estarão disponíveis para o acesso da aplicação. Em seguida, é criada uma rota para a aplicação.

A função que define uma rota começa com o verbo de acesso

(no caso `get`), que recebe dois parâmetros. O primeiro é o padrão da rota, que conterà o caminho da *URI*, junto com os parâmetros necessários, além de uma função de *callback*, que recebe como parâmetros os objetos `request` , `response` e `next` (o último é opcional), e define, em seu corpo, as ações a serem executadas quando essa rota é acionada.

Por padrão, o gerador do *Express* ainda utiliza a notação de funções tradicional. Para seguir o padrão estabelecido neste livro, elas devem ser trocadas pela notação `arrow` . Como exemplo, a rota apresentada anteriormente se apresentará da seguinte forma, após a troca:

```
router.get('/', (req, res, next) => {  
  ...  
});
```

O código `module.exports = router` permite que todas as rotas estabelecidas no arquivo sejam exportadas como um módulo. Dessa forma, vários arquivos podem ser usados para definir rotas, facilitando a organização do projeto.

A aplicação gerada ainda vem com outro exemplo de arquivos de rotas, chamado `users.js` , dentro da pasta `routes` .

Vale lembrar de que o endereço completo das rotas definidas nos arquivos foi apresentado no arquivo `app.js` . É lá que os módulos de rotas criados nos arquivos `index.js` e `users.js` são carregados. Assim, o primeiro parâmetro da função define um prefixo para cada rota criada nos arquivos.

Views

A pasta `views` armazena toda a hierarquia de páginas web,

como visto anteriormente. O *Express* tem a flexibilidade de aceitar diversas tecnologias de *template engines*, que são facilitadores de criação de páginas dinâmicas. Eles trazem diversos recursos, como sintaxe simplificada para impressão de variáveis, estruturas de repetição e herança de páginas.

Neste livro, o *template engine* conhecido como *Nunjucks* (<https://mozilla.github.io/nunjucks/>) será utilizado como o padrão. Veremos todos detalhes necessários sobre *template engines* no capítulo que trataremos sobre o *Nunjucks*.

8.6 CONSIDERAÇÕES

O *Express* é um dos frameworks web mais usado pela comunidade Node, e traz consigo um excelente gerador de aplicações. A estrutura funcional do projeto permite que novas funcionalidades sejam inseridas ou removidas de uma maneira simples.

No geral, basta conhecer o módulo em questão e seu funcionamento, para acoplá-lo na aplicação. A flexibilidade e simplicidade do *Express* são suas características marcantes, e fazem com que ele seja um dos preferidos dos desenvolvedores.

NUNJUCKS – TEMPLATE ENGINE

Um *template engine* é uma ferramenta que auxilia o desenvolvedor a criar páginas web dinâmicas, pois traz consigo uma série de funcionalidades de manipulação destas páginas. A partir de um *template engine*, é possível, dentro de uma página *html*, executar estruturas de controle, condição, manipulação de variáveis etc. Desta forma, podemos 'juntar' os dados vindos de uma base qualquer (banco de dados, por exemplo) com uma página *html*, alterando apenas os pontos específicos da página, nos quais os dados devem se encontrar.

É possível configurar o *Express*, quando usamos seu gerador, para trazer por padrão diversos tipos de *template engines* (basta ver a documentação do gerador para saber mais sobre os *template engines* disponíveis). Porém, o que vamos utilizar no contexto das aplicações desenvolvidas neste livro, o *Nunjucks*, precisa ser adicionado manualmente.

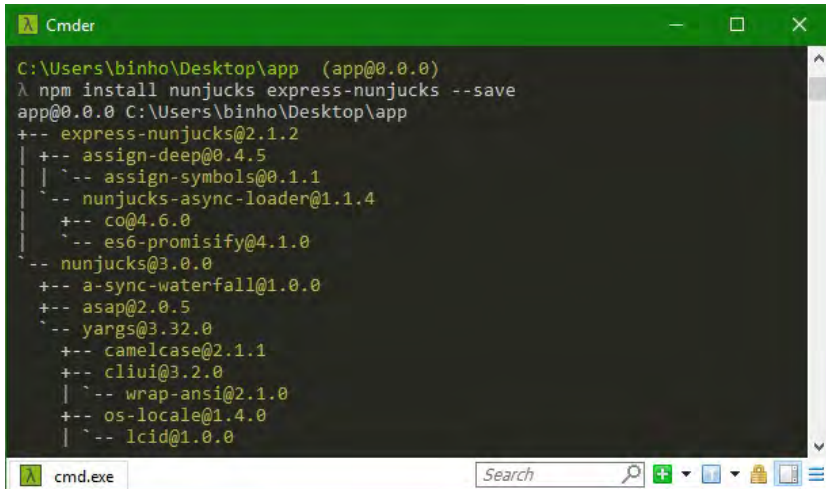
O *Nunjucks* é um *template engine*, atualmente, parte da fundação Mozilla. A partir de um módulo próprio para integrá-lo ao *Express*, podemos manipular todos recursos disponibilizados por ele. A documentação oficial do módulo pode ser encontrada

em <https://www.npmjs.com/package/express-nunjucks>.

O *Nunjucks* foi escolhido para ser o *template engine* apresentado neste livro, porque ele possui uma sintaxe intuitiva, uso de filtros, herança de *template*, e estruturas condicionais e de repetição. Além disso, a configuração dele é bem simples.

Para testarmos o *Nunjucks*, precisamos de uma aplicação *Express*. Vamos reaproveitar aquela que desenvolvemos no capítulo *Web server com Express*. Você pode baixar o projeto completo no repositório do GitHub do livro (<https://github.com/bmarchete/primeiros-passos-nodejs.git>).

Vamos começar instalando os módulos de configuração do *Nunjucks*. Para isso, precisamos abrir o terminal, localizando a pasta em que se encontra o projeto do *Express*. Em seguida, inserimos o comando `npm install nunjucks express-nunjucks -save`. Isso instalará o módulo *nunjucks* e o módulo para *Express*.



```
C:\Users\binho\Desktop\app (app@0.0.0)
λ npm install nunjucks express-nunjucks --save
app@0.0.0 C:\Users\binho\Desktop\app
+-- express-nunjucks@2.1.2
| +-- assign-deep@0.4.5
| | -- assign-symbols@0.1.1
| -- nunjucks-async-loader@1.1.4
|
+-- co@4.6.0
  -- es6-promise@4.1.0
-- nunjucks@3.0.0
+-- a-sync-waterfall@1.0.0
+-- asap@2.0.5
-- yargs@3.32.0
  +-- camelcase@2.1.1
  +-- cliui@3.2.0
  | -- wrap-ansi@2.1.0
  +-- os-locale@1.4.0
  | -- lcid@1.0.0
```

Figura 9.1: Instalação dos módulos do Nunjucks

Agora precisamos acessar o arquivo `app.js`, e importar o módulo do `nunjucks`. Insira a linha de código `var expressNunjucks = require('express-nunjucks')` na seção de importação de módulos (você pode inserir logo após a importação do módulo `body-parser`).

Neste momento, já podemos substituir o *jade* pelo *Nunjucks*. Altere a linha de código que define a `view engine` para `app.set('view engine', 'njk')`, e na linha de baixo, insira o código `var njk = expressNunjucks(app)`. Isso fará com que o *Nunjucks* se integre ao *Express*. Desta forma, podemos utilizar os arquivos de HTML do nosso projeto com a extensão padrão do *Nunjucks* (`.njk`).

Como não vamos mais utilizar os arquivos do *jade*, você pode remover todos eles da pasta `views`.

Para testarmos a configuração que realizamos, vamos criar um novo arquivo chamado `index.njk` na pasta `views`. Vamos criar um documento `html` padrão, que imprime o título fornecido pela rota configurada como a principal do projeto. A rota está configurada para devolver em uma variável `title` a palavra `Express`.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>{{title}}</title>
  </head>
  <body>
    <p>Olá mundo! Trabalhando com {{title}}</p>
  </body>
</html>
```

Com o *Nunjucks*, toda impressão de variável pode ser feita inserindo-se o nome da variável dentro de `{{ }}`. Certifique-se de salvar o arquivo `app.js` antes de executar o comando `nodemon` no terminal. Abrindo o navegador, veremos o resultado no endereço `localhost:3000`.

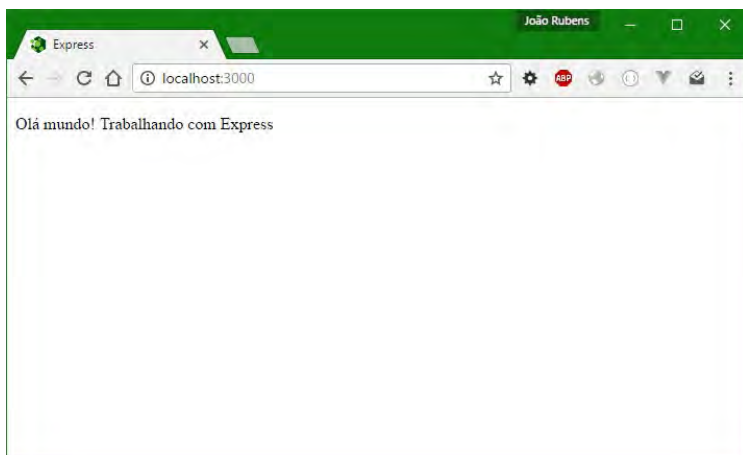


Figura 9.2: Olá mundo com Nunjucks

9.1 ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO

Vamos desenvolver um exemplo para trabalhar com estruturas de repetição e condição. Criaremos uma aplicação que permite ao usuário acessar, por exemplo, a URL *localhost:3000/tabuada/3* (o número 3 define o parâmetro da chamada), para que seja feita a impressão dos valores da tabuada do valor passado como parâmetro, caso este seja um valor par.

Primeiramente é preciso criar uma rota para o acesso da aplicação. Desta forma, vamos abrir o arquivo `index.js` que está dentro da pasta `routes`, no qual definimos as rotas da aplicação. Em seguida, devemos adicionar uma rota que receba um valor como parâmetro, fazer um cálculo para saber se o valor é par, e disponibilizá-lo a uma página que chamaremos de `tabuada.njk`.

O código para essa operação é apresentado a seguir:

```
router.get('/tabuada/:valor', function(req, res, next) {
  var valor = req.params.valor;
  var resultado = (valor % 2) == 0 ? valor : false;
  res.render('tabuada',{numero : resultado});
});
```

Observe que o resultado, caso não seja par, é retornado como `false`. A variável `numero` passada como parâmetro da função `render()` pode ser acessada por uma página `.njk`.

Agora precisamos criar um arquivo dentro da pasta `views` com o nome de `tabuada.njk`. Esse arquivo conterá o código *html* com a impressão da tabuada do número repassado pela rota criada anteriormente.

Com a sintaxe `{% %}`, determinamos as estruturas `if` e `for` para desenvolvermos a página. O trecho `{% if numero %}` determina se a variável `numero` representa algum valor, ou se encontra-se como `false`. Desta forma, imprimimos apenas a tabuada dos números pares. O trecho `{% for i in [0,1,2,3,4,5,6,7,8,9,10] %}` faz uma iteração do número 0 ao 10, garantindo a impressão linha a linha da tabuada. O código apresentado a seguir demonstra a criação da página por completo:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>{{title}}</title>
</head>
<body>
  {% if numero %}
    <h1>Tabuada do {{numero}}</h1>

    {% for i in [0,1,2,3,4,5,6,7,8,9,10] %}
      <p>{{numero}} x {{i}} = {{ numero * i}}</p>
    {% endfor %}
  {% else %}
```

```

    <h1>Números Ímpares são inválidos</h1>
    {% endif %}

</body>
</html>

```

Caso queira encontrar outros exemplos diversos de uso destas estruturas, você pode acessar o site <https://mozilla.github.io/nunjucks/templating>.

Para ver o resultado da aplicação, não se esqueça de reiniciar o servidor. Em seguida, acesse o endereço `localhost:3000/tabuada/8`, e você verá um resultado semelhante ao da figura a seguir:

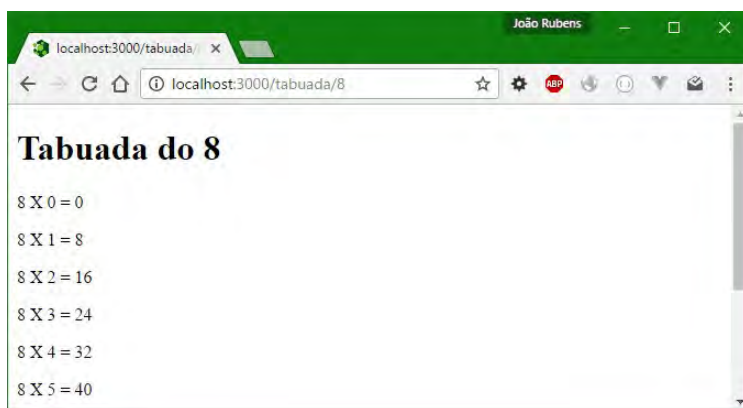


Figura 9.3: Impressão de tabuada na página web

9.2 HERANÇA DE TEMPLATE

O *Nunjucks* permite herança de *template*. Isso significa que é possível definir uma página como *layout* padrão e, a partir dela, definir subpáginas que preencherão certos espaços (conhecidos como blocos). Para exemplificar, vamos criar uma página padrão com menu e rodapé, utilizando o framework *Bootstrap* para

desenvolver o CSS.

Desta forma, crie um arquivo chamado `layout.njk` dentro da pasta `views`. Vamos trazer para dentro dele uma estrutura padrão de uma página com *Bootstrap*, que contém uma barra de navegação superior e uma seção de conteúdo. Um exemplo de código para esta página pode ser conferido a seguir:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title></title>

  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font-awesome/4.4.0/css/font-awesome.min.css">
</head>
<body>

  <nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navbar" aria-expanded="false" aria-controls="navbar">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="#">Nunjucks</a>
      </div>
      <div id="navbar" class="collapse navbar-collapse">
        <ul class="nav navbar-nav">
          <li class="active"><a href="#">Home</a></li>
          <li><a href="#about">About</a></li>
          <li><a href="#contact">Contact</a></li>
```

```

        <li class="dropdown">
            <a href="#" class="dropdown-toggle" data-toggle="drop
down" role="button" aria-haspopup="true" aria-expanded="false">Dr
opdown <span class="caret"></span></a>
            <ul class="dropdown-menu">
                <li><a href="#">Action</a></li>
                <li><a href="#">Another action</a></li>
                <li><a href="#">Something else here</a></li>
                <li role="separator" class="divider"></li>
                <li class="dropdown-header">Nav header</li>
                <li><a href="#">Separated link</a></li>
                <li><a href="#">One more separated link</a></li>
            </ul>
        </li>
    </ul>
</div>
</div>
</nav>

<div class="container">
    <div class="row" style="margin-top: 50px;">
        <div class="col-md-6">
            {% block pagina %} {% endblock %}

        </div>
    </div>
</div>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.
3/jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js
/bootstrap.min.js"></script>
</body>
</html>

```

Esse arquivo pode ser usado como base para as páginas da aplicação, já que todo o seu conteúdo serve apenas de *boilerplate* (código padrão de uma página *html*). A linha com código *Nunjucks* `{% block pagina %} {% endblock %}` indica que é neste ponto em que o conteúdo de uma página qualquer será adicionado. Um documento de *layout* pode conter diversos blocos,

além de aceitar toda estrutura padrão de um documento *Nunjucks* qualquer.

Agora, dentro do arquivo `tabuada.njk`, removemos todo o código referente aos cabeçalhos do *html*, pois a página herdará tudo isso de `layout.njk`. Dentro de um bloco (`{% block pagina %}`), todo o conteúdo inserido é renderizado em uma nova página. O código da página `tabuada.njk` refatorada é apresentado a seguir:

```
{% extends "layout.njk" %}

{% block pagina %}

    {% if numero %}
        <h1>Tabuada do {{numero}}</h1>

        {% for i in [0,1,2,3,4,5,6,7,8,9,10] %}
            <p>{{numero}} x {{i}} = {{ numero * i}}</p>
        {% endfor %}

    {% else %}
        <h1>Números Ímpares são inválidos</h1>
    {% endif %}

{% endblock %}
```

Ao executar novamente a aplicação, é possível visualizar a página de apresentação da tabuada com o novo *template*.

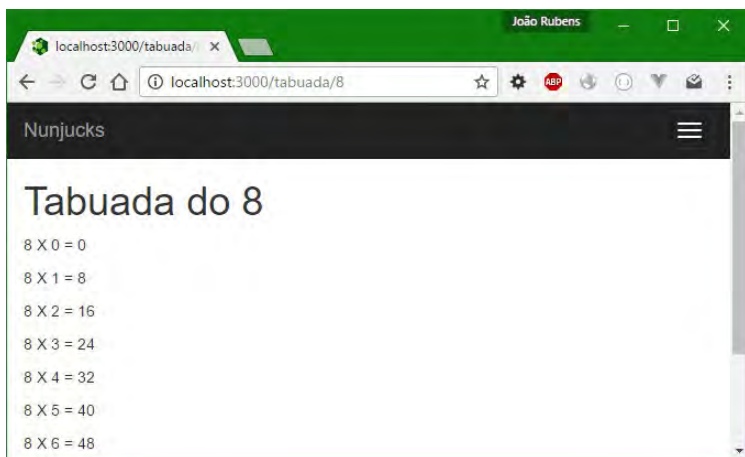


Figura 9.4: Página com herança de layout

9.3 PÁGINAS PARCIAIS

Um outro recurso relevante que o *Nunjucks* também disponibiliza é o uso de páginas parciais. É possível incluir páginas parciais (*sidebars*, por exemplo) em qualquer parte de uma página *Nunjucks*. Para ver como isso funciona, vamos abrir o arquivo `layout.njk` da pasta `views`.

Após a `div` com o código que define o bloco `pagina`, vamos inserir uma `div` que conterá o código `{% include "sidebar.njk" %}`. Esse comando permite ao *Nunjucks* apresentar um arquivo com código *html* no ponto especificado.

```
<div class="col-md-6">
  {% include "sidebar.njk" %}
</div>
```

Agora precisamos criar um novo arquivo chamado `sidebar.njk` na pasta `views`, que conterá o código a ser

renderizado pelo *Nunjucks* no local definido. Vamos criar uma `div` com uma margem e dois elementos dentro: um `h2` e um `p` :

```
<div style="margin: 10px; border: solid 1px black;">
  <h2>Eu sou uma sidebar</h2>
  <p>Sidebar Sidebar Sidebar</p>
</div>
```

O conteúdo do arquivo `sidebar.njk` é todo inserido no ponto de declaração `{% include %}` do arquivo de *layout*. A figura a seguir mostra o resultado do exemplo:

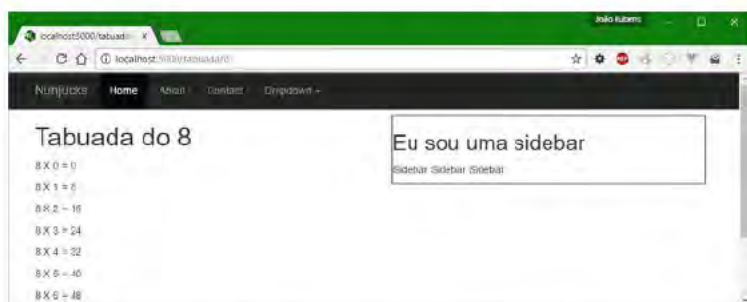


Figura 9.5: Inserção de arquivo parcial

9.4 CONSIDERAÇÕES

Existem diversos template engines que fazem operações semelhantes ao Nunjucks no mercado:

- Hogan (<http://twitter.github.io/hogan.js/>);
- Pug (<https://pugjs.org/api/getting-started.html>);
- Ejs (<http://www.embeddedjs.com/>);
- Twig (<https://www.npmjs.com/package/twig>).

Conhecer as características e performance de cada um facilita a

escolha. Dentre vários motivos, este livro apresenta o Nunjucks por sua sintaxe simplificada e seu sistema de herança de layouts.

APLICAÇÃO COM MYSQL

Dentre diversas outras funcionalidades, um servidor web deve ter a capacidade de se comunicar com um sistema gerenciador de banco de dados para armazenar dados de uma aplicação (SGBD). O uso de SGBDs é uma tarefa rotineira de desenvolvedores web, e a escolha de um em específico acaba envolvendo uma série de questões técnicas.

Por simplificação, neste livro, foram escolhidas duas tecnologias diferentes de banco de dados para a integração com servidores baseados em Node.js. A primeira, baseada no padrão relacional, utiliza o MySQL como SGBD. O MySQL é um SGBD famoso e de fácil instalação.

A segunda, baseia-se no conceito orientado a documentos. Para os exemplos da segunda tecnologia, mais adiante, usaremos o MongoDB.

10.1 EXEMPLOS DO CAPÍTULO

Para realizarmos os exemplos deste capítulo, vamos utilizar o projeto que desenvolvemos no capítulo *Nunjucks template engine*. Você pode baixar o projeto completo no repositório do GitHub do livro (<https://github.com/bmarchete/primeiros-passos-nodejs.git>).

Além disso, é preciso ter o MySQL instalado e rodando na porta padrão (3306).

10.2 CONTROLE DE PLAYLIST

A aplicação exemplo a ser criada permitirá ao usuário inserir, alterar, excluir e consultar músicas. Ao inserir uma música, o usuário digita o nome, artista e total de estrelas da música. O usuário pode também alterar os dados de uma música inserida (o id não pode ser alterado) e excluir qualquer música.

10.3 MÓDULOS DE CONEXÃO

Como quase tudo na comunidade Node, existem diversos módulos que facilitam a integração de aplicações web baseadas em Node com banco de dados. Dentre os mais utilizados, estão o Sequelize e o Knex.

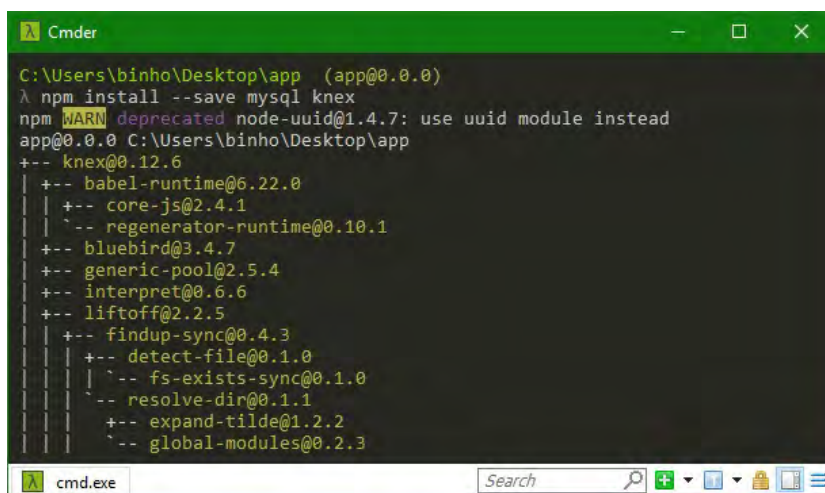
O Sequelize é um módulo que atua como um *ORM (Object Relational Mapping)* e permite, dentre outras operações, gerar as tabelas de um banco de dados a partir de código personalizado. O Knex é um módulo que traz uma série de funções que simplificam operações com dados, evitando o uso direto de SQL. As operações de inserção, consulta, alteração e remoção de dados podem ser usadas a partir de funções que retornam *promises*, o que torna o código mais claro e objetivo.

Ambos se integram facilmente com diversos SGBDs, como o MySQL, MariaDB e SQLite (dentre outros). Também são baseados em sistema de *promises* do JavaScript, deixando o código muito mais legível com relação ao uso de *callbacks*.

Nossa escolha será o Knex, pois o seu uso inicial é bem simplificado, já que não é objetivo deste livro o aprofundamento em conceitos relacionados a operações com dados.

10.4 CONFIGURAÇÃO DO KNEX

Para configurar o Knex, siga as seguintes instruções. Precisamos instalar o módulo na aplicação a partir do comando `npm install --save mysql knex`.



```
C:\Users\binho\Desktop\app (app@0.0.0)
λ npm install --save mysql knex
npm WARN deprecated node-uuid@1.4.7: use uuid module instead
app@0.0.0 C:\Users\binho\Desktop\app
+-- knex@0.12.6
| +-- babel-runtime@6.22.0
| | +-- core-js@2.4.1
| | `-- regenerator-runtime@0.10.1
| +-- bluebird@3.4.7
| +-- generic-pool@2.5.4
| +-- interpret@0.6.6
| +-- liftoff@2.2.5
| | +-- findup-sync@0.4.3
| | | +-- detect-file@0.1.0
| | | | `-- fs-exists-sync@0.1.0
| | | `-- resolve-dir@0.1.1
| | +-- expand-tilde@1.2.2
| | `-- global-modules@0.2.3
```

Figura 10.1: Instalação dos módulos do MySQL e Knex

Na sequência, criaremos um arquivo chamado `db.js` dentro do diretório raiz da aplicação. Nele, será desenvolvido um módulo com as características de conexão do banco de dados, evitando a necessidade de criar essas configurações a cada operação com dados. Caso haja alguma configuração de host, user ou database em seu sistema diferentes das apresentadas aqui, altere o código conforme necessário.

```
var knex = require('knex');

var db = knex({
  client: 'mysql',
  connection: {
    host : '127.0.0.1',
    user : 'root',
    database : 'musics'
  }
});

module.exports = db;
```

10.5 CRIAÇÃO DA TABELA MUSICAS

Agora, precisamos criar uma tabela chamada `musicas` dentro do banco de dados, que vai contar as informações de todas as músicas gerenciadas pela aplicação. Para isso, você pode abrir seu editor de MySQL favorito e inserir o código de criação da tabela música, contendo os campos `id` (chave primária e autoincremento), `nome` e `artista` (tipo texto), e `estrelas` (tipo inteiro).

```
CREATE TABLE `musics`.`musicas` (
  `id` INT NOT NULL AUTO_INCREMENT ,
  `nome` VARCHAR(30) NOT NULL ,
  `artista` VARCHAR(30) NOT NULL ,
  `estrelas` INT NOT NULL ,
  PRIMARY KEY (`id`)
) ENGINE = InnoDB;
```

10.6 CRIAÇÃO DO MÓDULO DE ROTAS

Para atender aos requisitos da aplicação estabelecidos anteriormente, o módulo de rotas deverá apresentar as seguintes configurações:

- Rota de acesso principal da aplicação, em que serão exibidas todas as músicas cadastradas, e as opções de alterar e excluir (rota / do tipo get);
- Rota de formulário de inserção de músicas (rota /add do tipo get);
- Rota de cadastro de músicas, que recebe os dados do cadastro e insere no banco de dados (rota / do tipo post);
- Rota de formulário de edição de uma música (rota /edit/:id do tipo get);
- Rota de alteração dos dados de uma música (rota /edit/:id do tipo put);
- Rota de exclusão de uma música (rota /delete/:id do tipo delete).

Para isso, vamos começar criando um novo arquivo chamado `musicas.js` dentro da pasta `routes`, que será o nosso módulo de rotas. Dentro dele, vamos definir as rotas como apresentadas anteriormente, e carregar módulo `db.js` para que as operações com dados possam ser realizadas.

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

var db = require('../db');

router.get('/', (req, res, next) => {

});

router.get('/add', (req, res, next) => {

});

router.post('/', (req, res, next) => {
```

```
});

router.get('/edit/:id', (req, res, next)=>{

});

router.put('/edit/:id', (req, res, next)=>{

});

router.delete('/delete/:id', (req, res, next)=>{

});

module.exports = router;
```

Configuração

Neste momento, é preciso configurar o nosso novo módulo de rotas na aplicação. Isso é feito no arquivo `app.js`. Vamos substituir a linha de código `var index = require('./routes/index)` por `var index = require('./routes/musicas')`, fazendo com que a aplicação passe a carregar o módulo criado anteriormente.

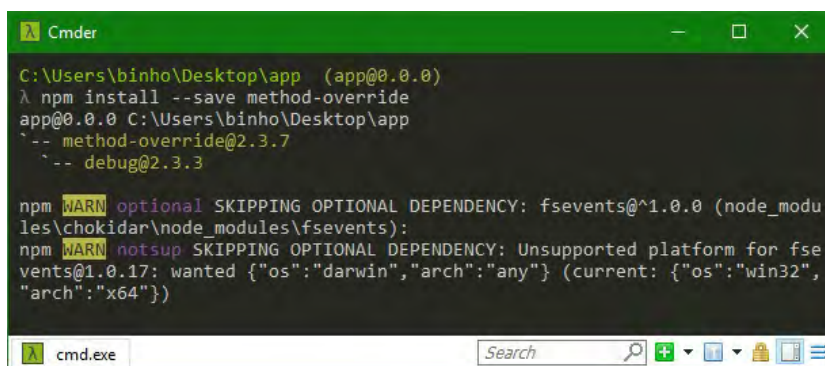
Como este será o único arquivo de rotas que vamos carregar, as referências ao arquivo de rotas `users.js` podem ser removidas também (`var users = require('./routes/users)` e `app.use('/users', users)`). Fechando este procedimento, vamos remover os demais arquivos da pasta `routes`, deixando apenas o arquivo `musicas.js`.

Rotas baseadas em verbos HTTP

Outro ponto importante a destacar do arquivo de rotas criado é o uso dos verbos *HTTP* `put` e `delete`. Como nossa aplicação

receberá dados de um formulário, é preciso configurá-la para realizar uma ‘conversão’ de dados vindos por `post` para as rotas de `put` e `delete`, já que formulários *HTML* não suportam esses verbos.

Essa conversão pode ser feita facilmente a partir da inserção de um módulo apropriado. Usaremos aqui o `method-override`. Por esse motivo, dentro do terminal, na pasta do projeto, vamos inserir o comando `npm install --save method-override`.



```
C:\Users\binho\Desktop\app (app@0.0.0)
λ npm install --save method-override
app@0.0.0 C:\Users\binho\Desktop\app
  -- method-override@2.3.7
  -- debug@2.3.3

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@^1.0.0 (node_modules\chokidar\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.0.17: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
```

Figura 10.2: Adição do modulo `method-override`

Depois, no arquivo `app.js`, importaremos o módulo instalado com o código `var methodOverride = require('method-override');`, o que pode ser feito logo abaixo da linha de importação do arquivo de rotas.

Para configurar o uso do módulo, ainda dentro do arquivo `app.js`, logo após a linha de código `app.use(express.static(path.join(__dirname, 'public')))`, vamos configurar o módulo instalado para que nossa aplicação reconheça, de alguma forma, os métodos `put` e

delete . Essas configurações podem ser encontradas no site com a documentação do módulo (<https://github.com/expressjs/method-override>).

```
app.use(methodOverride(function (req, res) {
  if (req.body && typeof req.body === 'object' && '_method' in req.body) {
    var method = req.body._method
    delete req.body._method
    return method
  }
}));
```

Essas configurações permitem a criação de um campo de entrada em um formulário HTML com o nome `_method` , que indicará o valor `put` ou `delete` para que a conversão seja realizada pelo servidor web no momento do recebimento do formulário. Esse campo, geralmente, deve ficar oculto para o usuário, pois ele é de uso da aplicação.

10.7 DESENVOLVIMENTO DAS PÁGINAS WEB

Pensando em termos de páginas web, nossa aplicação terá 3:

- A página principal, que apresentará os dados de todas as músicas, e uma opção para excluir e alterar dados;
- A página de inserção de novas músicas;
- A página de alteração dos dados de uma música.

Todas as páginas vão herdar o *layout* já estabelecido no arquivo `layout.njk` da pasta `views` . Esse arquivo precisa de algumas alterações.

Na primeira alteração, vamos remover a seção de páginas parciais, excluindo as seguintes linhas de código:

```
<div class="col-md-6">
    {% include "sidebar.njk" %}
</div>
```

Na segunda, vamos refatorar a seção `<nav>`, para que tenha uma barra de menus da aplicação. Nela, vamos inserir uma opção para acessar a página inicial, e uma para acessar o formulário de cadastro de músicas.

```
<nav class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-
a-toggle="collapse" data-target="#navbar" aria-expanded="false" a
ria-controls="navbar">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="/">Músicas</a>
    </div>
    <div id="navbar" class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <li class="active"><a href="/">Início</a></li>
        <li><a href="/add">Inserir Músicas</a></li>
      </ul>
    </div>
  </div>
</nav>
```

Na terceira, vamos trocar a classe em que está a configuração de bloco de conteúdo para suportar o padrão de doze colunas do *Bootstrap*.

```
<div class="col-md-12">
    {% block pagina %} {% endblock %}
</div>
```

Na quarta, dentro da pasta `views`, vamos criar os arquivos `index.njk` (caso ainda não exista), `add.njk` e `edit.njk`. Os demais arquivos da pasta podem ser removidos (exceto o arquivo `layout.njk`).

Por último, em cada arquivo de *view* que foi criado, vamos iniciar uma estrutura básica, que faz a importação do *layout* padrão e define o bloco de conteúdo.

```
{% extends "layout.njk" %}

{% block pagina %}

    //código da página

{% endblock %}
```

Página principal

Vamos definir na página `index.njk`, um cabeçalho de apresentação, seguido de uma tabela com os dados de todas as músicas. A última coluna da tabela pode apresentar os botões para as ações de editar e excluir. A seguir, está um exemplo de código da página:

```
<h1>Músicas da minha Playlist</h1>

<table class="table table-striped">
  <thead>
    <tr>
      <th>Id</th>
      <th>Nome</th>
      <th>Artista</th>
      <th>Estrelas</th>
      <th>Ações</th>
    </tr>
  </thead>
  <tbody>
    <tr>
```

```

<td>1</td>
<td>Californication</td>
<td>Red Hot Chili Peppers</td>
<td>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
    <span class="glyphicon glyphicon-star"></span>
</td>
<td>
    <a href="#" class="btn btn-primary">Editar</a>
    <a href="#" class="btn btn-danger">Excluir</a>
</td>
</tr>
</tbody>
</table>

```

Neste exemplo, a tabela apresenta apenas um registro estático, mas já configura todo o layout necessário.

Página de inserção de músicas

Na página `add.njk`, vamos definir um cabeçalho de apresentação, seguido de um formulário com campos de inserção de uma nova música. A seguir, você pode ver um exemplo dessas especificações, já com as classes do *Bootstrap* apropriadas para a formatação do formulário.

```

<h1>Cadastro de Músicas</h1>
<div class="col-md-6">
    <form>
        <div class="form-group">
            <label for="nome">Nome da música</label>
            <input type="text" class="form-control" name="nome" id="nome" placeholder="Nome">
        </div>
        <div class="form-group">
            <label for="artista">Nome do artista</label>
            <input type="text" class="form-control" name="artista" id="artista" placeholder="Artista">
        </div>
        <div class="form-group">

```

```

        <label for="estrelas">Total de estrelas</label>
        <input type="text" class="form-control" name="estrelas" id=
"estrelas" placeholder="Estrelas">
    </div>

    <button type="submit" class="btn btn-primary">Cadastrar Músic
</button>
    <a href="#" class="btn btn-default">Voltar</a>
</form>
</div>

```

Lembre-se de que o campo de id não precisa ser inserido, pois será autoincrementado pela aplicação.

Página de edição de músicas

Na página `edit.njk`, precisamos desenvolver um formulário bem parecido com o da página `add.njk`, para que os dados da música sejam alterados.

Vamos inserir um parágrafo indicando o `id` da música que está sendo alterada no momento. A seguir, você pode conferir um exemplo deste formulário:

```

<h1>Alteração de Música</h1>
<div class="col-md-6">
    <form>
        <div class="form-group">
            <label>Id</label>
            <p class="form-control" id="nome">2</p>
        </div>
        <div class="form-group">
            <label for="nome">Nome da música</label>
            <input type="text" class="form-control" name="nome" id="nom
e" value="Music">
        </div>
        <div class="form-group">
            <label for="artista">Nome do artista</label>
            <input type="text" class="form-control" name="artista" id="
artista" value="Some">

```

```

</div>
<div class="form-group">
  <label for="estrelas">Total de estrelas</label>
  <input type="text" class="form-control" name="estrelas" id=
"estrelas" value="2">
</div>

  <button type="submit" class="btn btn-primary">Alterar Música<
button>
  <a href="#" class="btn btn-default">Voltar</a>
</form>
</div>

```

10.8 OPERAÇÕES COM DADOS

Dentro do arquivo `musicas.njk`, definiremos as operações de manipulação dos dados da aplicação. Alguns desenvolvedores da comunidade preferem criar módulos que chamam de controladores para realizar esses procedimentos, mas eles não alteram o funcionamento da aplicação. Para fins de conceituação, usaremos o próprio arquivo de rotas para desenvolver as operações com dados.

Consulta de dados

Como visto anteriormente, a página inicial precisa receber da aplicação todas as músicas cadastradas no banco de dados. Neste momento, seria interessante que você usasse um programa qualquer de acesso ao *MySQL* para inserir algumas linhas de dados, a fim de realizarmos alguns testes.

Dentro do arquivo de rotas criados anteriormente, na rota que define o retorno de dados para a página principal, a aplicação precisa recuperar os dados do banco de dados e associá-los à página `index.njk`. O código *SQL* que faz esta consulta é `SELECT`

* FROM musicas . Mas com a utilização do Knex , esse processo não envolve o uso de SQL. O código de consulta passa a ser:

```
db("musicas").then((musicas)=>{  
  
});
```

Esse método utiliza *promises* do *JavaScript* para armazenar, na variável `musicas` , o resultado da consulta de todas as músicas com o módulo `db` previamente incluído.

Caso ocorra algum erro na execução, é possível passar um segundo argumento à função `then()` , que funciona como um `catch` . O objeto `next` pode ser passado para que a requisição continue para a rota que trata de erros definida no arquivo `app.js` , como visto no capítulo *Web server com Express*. O código passa a ser o seguinte:

```
db("musicas").then((musicas)=>{  
  
}, next);
```

Caso a operação de consulta dos dados ocorra bem, os dados são associados à página `index` , como mostra o código final para a rota:

```
router.get('/', (req, res, next) => {  
  db("musicas").then((musicas)=>{  
    res.render('index', {  
      musicas: musicas  
    });  
  }, next);  
});
```

Inserção de dados

Agora vamos configurar a rota responsável pela entrega da

página que contém o formulário de cadastro. Dentro do arquivo de rotas, alteramos o conteúdo, fazendo com que uma página seja renderizada na resposta.

```
router.get('/add', (req, res, next) => {  
  res.render('add');  
});
```

Isso faz com que o endereço `localhost:3000/add` já devolva o formulário de cadastro, como mostra a figura a seguir:

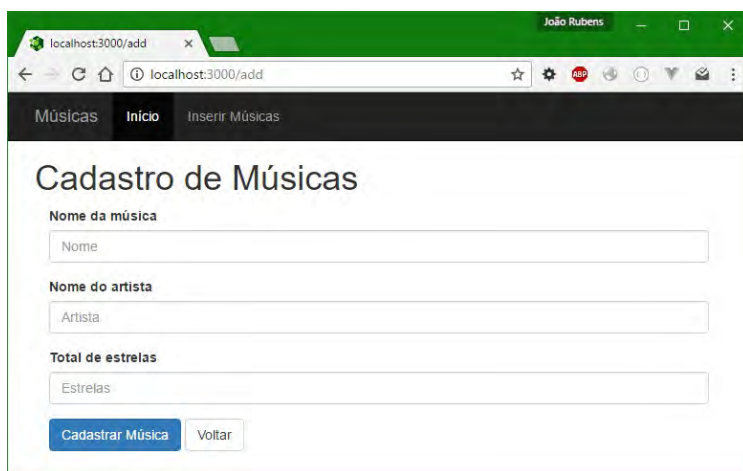
A screenshot of a web browser window. The address bar shows 'localhost:3000/add'. The browser has a green title bar with the name 'João Rubens'. The page has a dark navigation bar with links 'Músicas', 'Início', and 'Inserir Músicas'. The main content area is titled 'Cadastro de Músicas' and contains three input fields: 'Nome da música' (with placeholder 'Nome'), 'Nome do artista' (with placeholder 'Artista'), and 'Total de estrelas' (with placeholder 'Estrelas'). At the bottom are two buttons: 'Cadastrar Música' (blue) and 'Voltar' (white).

Figura 10.3: Formulário de cadastro de músicas

Agora falta definir a ação de inserir registros no banco de dados. Para isso, os dados chegam a partir de uma submissão do formulário utilizando o método `post`, e são inseridos no banco de dados, a partir da função `insert` do *Knex*.

A função `insert`, permite repassar os dados diretamente ao banco de dados quando eles possuem os mesmos nomes definidos no formulário que os enviou. Esse processo simplifica a passagem

de dados e enxuga o código. Caso seja necessário, a função `insert` também permite a inserção de uma coleção de dados de uma vez só.

O código seguinte apresenta o fluxo de inserção de dados na aplicação com o *Knex*:

```
router.post('/', (req, res, next) => {
  db("musicas").insert(req.body).then((ids) => {
    res.redirect('/');
  }, next)
});
```

A variável `ids` armazena o(s) id(s) criado(s) pela inserção. Essa variável pode ser utilizada para uma futura mensagem de inserção realizada. No momento, a aplicação é apenas redirecionada para a rota principal, caso a inserção ocorra normalmente.

Edição de dados

Agora faremos a configuração das rotas responsáveis pela entrega da página que contém o formulário de edição, e pelo processamento da edição.

Na primeira, uma constante `{id}` permite extrair o valor do `id`, que vem a partir dos parâmetros do formulário enviado com os dados para edição. Em seguida, a função `where()` do *Knex* realiza uma consulta ao banco, buscando o primeiro registro que encontrar (`first()`). Caso não seja encontrado nenhum registro que corresponda ao `id` recebido, é devolvido um código *HTTP* de solicitação imprópria (`return res.send(400)`). Do contrário, a página de edição é renderizada com os dados da música encontrada.

```

router.get('/edit/:id', (req, res, next)=>{
  const {id} = req.params;

  db("musicas")
    .where("id", id)
    .first()
    .then((musica) => {
      if (!musica) {
        return res.send(400);
      }

      res.render("edit.njk", {
        musica: musica
      });
    }, next);
});

```

Quando o formulário de edição for preenchido e submetido, a rota de processamento da edição será acionada. Semelhantemente à rota de edição anterior, é feita uma busca pelos dados da música que contém o id de alteração. A grande diferença está no uso da função `update(req.body)` em vez da função `first()`, já que neste momento os dados serão alterados.

Vale lembrar de que esta rota é acionada pelo verbo `put`, e não por `get`. Quando o resultado de uma alteração termina em zero, significa que não foi realizada nenhuma alteração, e o código *HTTP* é retornado.

```

router.put('/edit/:id', (req, res, next)=>{
  const {id} = req.params;

  db("musicas")
    .where('id', id)
    .update(req.body)
    .then((result) => {
      if (result === 0) {
        return res.send(400);
      }
    })

```

```

    res.redirect('/');
  }, next)
});

```

Exclusão de dados

Por último, faremos a configuração da rota responsável pela exclusão de um registro de música na tabela. Neste exemplo, ao clicar em excluir, não será feita nenhuma solicitação de confirmação, fazendo com que a rota de exclusão seja acionada imediatamente.

Para melhorar essa dinâmica, o leitor pode optar por adicionar uma rota que apresente um formulário com a confirmação da exclusão, ou simplesmente um código JavaScript na página `index` solicitando a confirmação. Observe que o processo é bem semelhante ao processo de atualização. A diferença está no uso da função `delete()` no lugar da função `update()`.

```

router.delete('/delete/:id', (req, res, next) => {
  const {id} = req.params;

  db("musicas")
    .where('id', id)
    .delete()
    .then((result) => {
      if (result === 0) {
        return res.send(400);
      }
      res.redirect('/');
    }, next)
});

```

Feitas as alterações no arquivo de rotas, a aplicação está pronta para lidar com os dados e ações vindas das páginas web.

10.9 TEMPLATING

O termo *templating* vem da necessidade de adicionar código interpretado pelo servidor nas páginas web, para torná-las dinâmicas. Isso significa que as páginas apresentarão informações que se alteram, dependendo do estado dos dados no servidor.

Como visto anteriormente, o *Nunjucks* é um *template engine*, e fornece uma série de funcionalidades para criar páginas dinâmicas, a partir da manipulação dos dados, que nesta aplicação, são incorporadas às páginas (a partir do arquivo de rotas), no momento da renderização.

Página principal

A página principal deve imprimir os dados vindos do banco de dados. Dentro da tabela incluída na página, é preciso iterar pelos dados e fazer as impressões nas devidas colunas. Vamos inserir um bloco `{% for %}` para realizar essa iteração.

Dentro deste bloco, vamos imprimir um ícone da biblioteca de ícones do *Bootstrap* em formato de estrela, para representar o total de estrelas. A estrutura `{% for %}` mais interna executa n vezes, em que n corresponde ao número de estrelas dado pela variável `musica.estrelas`. Assim, n ícones são impressos na página, pois correspondem ao desenho de uma estrela.

```
<tbody>
  {% for musica in musicas %}
    <tr>
      <td>{{musica.id}}</td>
      <td>{{musica.nome}}</td>
      <td>{{musica.artista}}</td>
      <td>
        {% for i in range(0, musica.estrelas) %}
```

```

        <span class="glyphicon glyphicon-star"></span>

        {% endfor %}

    </td>
    <td>
        <a href="#" class="btn btn-primary">Editar</a>
        <a href="#" class="btn btn-danger">Excluir</a>
    </td>
</tr>
{% endfor %}

</tbody>

```

Uma última alteração precisa ser feita nos links das ações. O link editar deve redirecionar a aplicação para o recurso `/edit/id` a partir do método `get`. Já o link de exclusão deve redirecionar a aplicação para `/delete/id`, a partir do método `delete`.

```

<td>
    <a href="/edit/{{musica.id}}" class="btn btn-primary">Editar</a>

    <button class="btn btn-danger"
        onclick="event.preventDefault();
        document.getElementById('logout-form-{{musica.id}}').submit()
    ;">Excluir
    </button>
</td>

<form id="logout-form-{{musica.id}}" action="/delete/{{musica.id}}
" method="POST" style="display: none;">
    <input type="hidden" name="_method" value="DELETE" />
</form>

```

O link para a página de edição pode ser feito diretamente a partir da impressão da variável `musica.id`. Já o link para a rota de exclusão precisa ativar um formulário que envie, a partir do método `post`, um campo chamado `_method`, com o valor `delete`, para que o *Express* entenda que a rota desejada é acessada pelo verbo *HTTP delete* (configuração realizada

anteriormente para o módulo `method-override`). O formulário deve ficar oculto na página, pois não representa nada em termos de ação para o usuário da aplicação.

Após as alterações realizadas, a página de consultas está totalmente funcional. Acessando a página principal da aplicação, é possível visualizar o resultado da consulta:



Id	Nome	Artista	Estrelas	Ações
1	Californication	Red Hot Chili Peppers	★★★★★	<button>Editar</button> <button>Excluir</button>
2	Fly me to the moon	Frank Sinatra	★★★★	<button>Editar</button> <button>Excluir</button>

Figura 10.4: Dados apresentados na página inicial

Página de inserção

Na página `add.njk`, o formulário precisa enviar os dados pelo método `post` para a rota principal. Para isso, basta que mudemos a linha de código `<form>` para `<form action="/" method="post">`. Deste modo finalizamos o processo de inserção.

Página de edição

Para que o formulário de edição envie os dados para a rota correta, precisamos alterar a tag `<form>`, fazendo com que o

formulário chame a rota de edição, repassando a ela o `id` da música a ser alterada.

Outra alteração importante é a adição de um campo `<input>`, que permitirá ao servidor interpretar essa chamada como um envio do tipo `put` (e não `post`), correspondendo à configuração da rota de edição.

```
<form action="/edit/{{musica.id}}" method="post">
  <input type="hidden" name="_method" value="PUT">
```

A última alteração no formulário será realizada na propriedade `value` dos campos do formulário. Essa propriedade deve apresentar os dados da música a ser alterada no momento, e não dados estáticos. Para isso, usamos a sintaxe de impressão de variáveis do *Nunjucks*, identificando o nome de cada propriedade a ser impressa nos campos.

```
<input type="text" class="form-control" name="nome" id="nome" value="{{musica.nome}}">
<input type="text" class="form-control" name="artista" id="artista" value="{{musica.artista}}">
<input type="text" class="form-control" name="estrelas" id="estrelas" value="{{musica.estrelas}}">
```

A figura a seguir representa um exemplo de página de edição funcional:

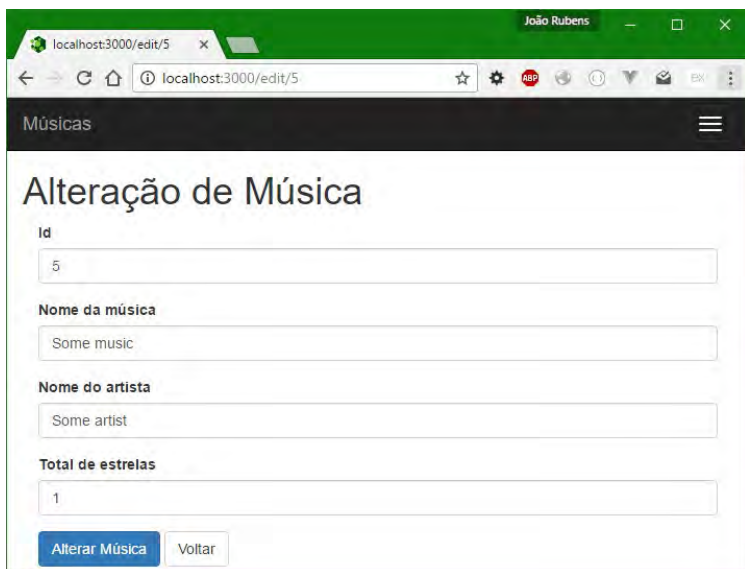


Figura 10.5: Página de alteração de músicas funcional

10.10 CONSIDERAÇÕES

Ao fim de todas as etapas deste capítulo, é possível executar a aplicação, que fará a manipulação correta dos dados, utilizando o MySQL. A manipulação de dados baseados em SGBDs relacionais com o módulo aqui apresentado não só simplifica diversos processos, como garante eficiência no desenvolvimento da aplicação apresentada.

Diversos outros módulos podem fornecer proteção contra ataques que envolvem banco de dados (como SQL injection), migrations, criação dinâmica de tabelas etc. Com este exemplo em mãos, o leitor pode utilizar os diversos fóruns mantidos pela comunidade, o GitHub, ou até mesmo o próprio site de repositórios do npm (já citado anteriormente) para descobrir esses

módulos e suas características.

INTEGRAÇÃO COM MONGODB

No capítulo *Integração com MySQL*, vimos a simplicidade de manipular registros vindos de um banco de dados MySQL. Da mesma forma, a comunidade Node desenvolveu diversos módulos para manipular outros tipos de banco de dados, os chamados *NoSQL* (*Not Only SQL*).

Essas tecnologias de banco de dados diferem no uso de *SQL* e do *modelo relacional* para manipular os registros. Dentre as características mais comumente encontradas em *NoSQLs*, estão:

- Não aplicam schema;
- Modelo de dados flexível;
- Projetados para lidar com clusters distribuídos.

O MongoDB é um banco de dados *NoSQL* orientado a documentos. Isso significa que, em vez de trabalhar com estruturas no formato de tabelas, ele utiliza conjuntos de documentos (no formato *JSON*) para definição dos dados. Isso traz muita flexibilidade no momento da definição dos dados, pois não é preciso definir colunas para armazenar os dados.

Muito poderia ser falado sobre o MongoDB, mas entender

como ele funciona e quais aplicações ele atende melhor que um banco *SQL* foge do escopo deste livro. O intuito aqui é apresentar como integrar o MongoDB ao Node.js.

Se você se interessa por saber mais sobre o MongoDB, recomendo os cursos gratuitos da **MongoDB University** (<https://university.mongodb.com/>).

11.1 EXEMPLOS DO CAPÍTULO

Para realizarmos os exemplos deste capítulo, usaremos o projeto que desenvolvemos no capítulo *Aplicação com MySQL*. Você pode baixar o projeto completo no repositório do GitHub do livro (<https://github.com/bmarchete/primeiros-passos-nodejs.git>).

Além disso, você deverá ter o MongoDB instalado e rodando na porta padrão (27017).

11.2 MÓDULOS DE CONEXÃO

Para fazer com que uma aplicação Node se conecte ao MongoDB, é preciso baixar o driver de conexão. Ele dará suporte às operações com dados, fornecendo uma camada de abstração.

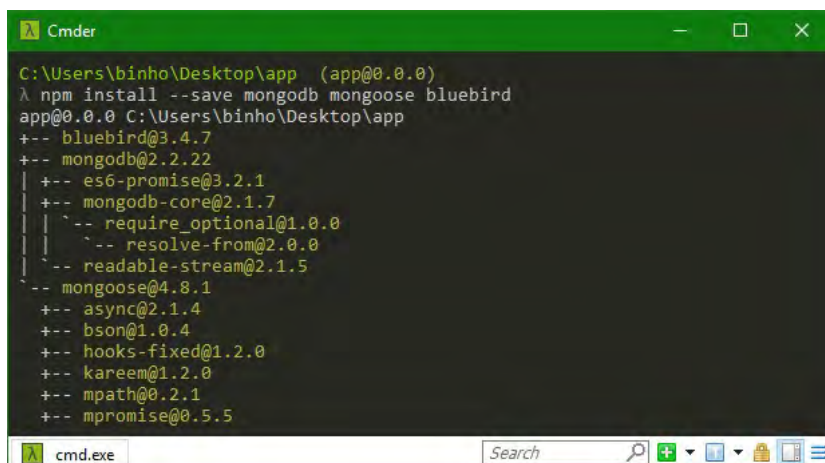
Para mais detalhes não cobertos neste livro sobre o uso do driver, acesse <https://mongodb.github.io/node-mongodb-native/>.

Apesar de suficiente para realizar as transações com registros

(CRUD), no momento da escrita deste livro, o driver do MongoDB para Node não implementa a API de *promises* do *JavaScript*. Por este motivo, será necessário instalar um segundo módulo, chamado de *bluebird.js* (<http://bluebirdjs.com/docs/getting-started.html>), que se integrará ao driver e o habilitará para ser manipulado com *promises*.

Por último, um terceiro módulo chamado *mongoose.js* (<http://mongoosejs.com/>) permite desenvolver arquivos de modelo para a aplicação. Diferentemente dos bancos de dados relacionais, a lógica do negócio de uma aplicação baseada em MongoDB deve ser totalmente desenvolvida pelo servidor da aplicação, já que esses tipos de banco de dados não possuem ferramentas como chaves primárias e estrangeiras, ou definição de tipos de dados para criar restrições e validação dos registros.

Para instalar os módulos citados nesta seção, vamos abrir o terminal e digitar o comando `npm install --save mongodb mongoose bluebird`.



```
C:\Users\binho\Desktop\app (app@0.0.0)
λ npm install --save mongodb mongoose bluebird
app@0.0.0 C:\Users\binho\Desktop\app
+-- bluebird@3.4.7
+-- mongodb@2.2.22
|   +-- es6-promise@3.2.1
|   +-- mongodb-core@2.1.7
|   |   +-- require_optional@1.0.0
|   |   |   +-- resolve-from@2.0.0
|   |   |   +-- readable-stream@2.1.5
|   +-- mongoose@4.8.1
|   +-- async@2.1.4
|   +-- bson@1.0.4
|   +-- hooks-fixed@1.2.0
|   +-- kareem@1.2.0
|   +-- mpath@0.2.1
|   +-- mpromise@0.5.5
```

Figura 11.1: Instalação dos módulos de acesso ao MongoDB

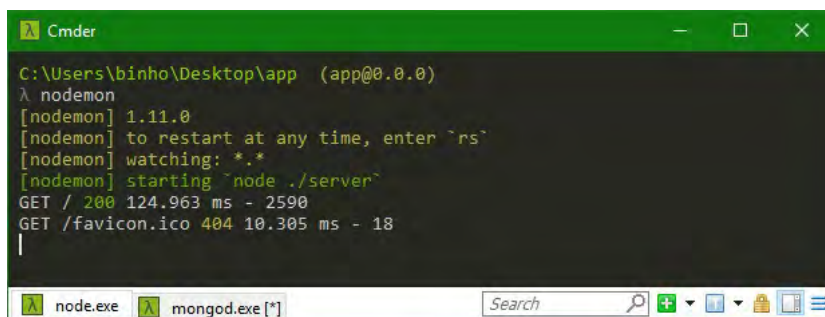
11.3 CONFIGURAÇÃO DOS MÓDULOS

Para configurar os módulos instalados na aplicação, devemos acessar o arquivo `app.js` e fazer a importação. Para isso, logo abaixo da linha de código `var methodOverride = require('method-override')`, precisamos inserir os módulos e estabelecer a conexão com o banco de dados.

```
var mongoose = require('mongoose')
mongoose.Promise = require('bluebird');
mongoose.connect('mongodb://localhost/musics');
```

Esse código importa o módulo `mongoose`, configura o módulo `bluebird` para ser a implementação do sistema de *promises* e define o endereço da conexão com o banco de dados.

Caso o servidor do MongoDB esteja iniciado na porta padrão, a aplicação será executada sem erros. Caso o servidor MongoDB não esteja executando, você precisa abrir um terminal e digitar o comando `mongod`, que iniciará o servidor.



```
C:\Users\binho\Desktop\app (app@0.0.0)
λ nodemon
[nodemon] 1.11.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node ./server`
GET / 200 124.963 ms - 2590
GET /favicon.ico 404 10.305 ms - 18
|
```

Figura 11.2: Aplicação executada após configuração dos módulos de conexão com MongoDB

11.4 CRIAÇÃO DO SCHEMA PARA MUSICAS

Como dito anteriormente, é preciso definir a lógica de negócio diretamente na aplicação. O módulo *mongoose* permite definir um objeto *Schema*, que pode trazer dentro de si as regras de validação e outras funcionalidades (como métodos *helpers*, não cobertos neste livro).

Para validar os dados da aplicação, vamos utilizar apenas as regras de definição de tipo de dados e de obrigatoriedade. Uma lista completa de regras pode ser visualizada em <http://mongoosejs.com/docs/guide.html>.

Para criar o esquema de dados para músicas, criaremos uma nova pasta no diretório raiz da aplicação chamada `models`. Dentro da nova pasta criada, crie um arquivo chamado `musics.js`. Este arquivo representará o modelo de dados das músicas.

O arquivo receberá um novo objeto *Schema* `new mongoose.Schema()`, que conterá as regras dos dados: `nome` e `artista` devem ser do tipo `String` e obrigatórios, e `estrelas` deve ser do tipo *número* e obrigatório. Observe que essas regras são semelhantes àquelas que definimos no capítulo *Aplicação com MySQL*. O trecho de código a seguir apresenta um exemplo de definição destas regras:

```
var mongoose = require('mongoose');

var MusicaSchema = new mongoose.Schema({
  nome: {
    type: String,
    required: true
  },
  artista: {
    type: String,
    required: true
  }
});
```

```

    },
    estrelas: {
      type: Number,
      required: true
    },
  },
});

module.exports = mongoose.model('Musica', MusicaSchema);

```

A última linha do código permite exportar as definições como um módulo do tipo `mongoose.model()`, que recebe como parâmetro o nome de acesso e as definições do esquema. É preciso disponibilizar o módulo de *schema* para a aplicação *Express*.

Para isso, devemos inserir o código `require('./models/musics')` antes da linha `var mongoose = require('mongoose')` no arquivo `app.js`. Dessa forma, o módulo contendo o *schema* para as músicas é carregado pela aplicação e fica disponível para uso a partir do módulo *mongoose*.

11.5 OPERAÇÕES COM DADOS

O arquivo `musicas.njk` da pasta `routes` conterà agora as definições de acesso ao MongoDB, e não mais ao *MySQL*. Portanto, vamos remover todas as operações com registros anteriores do arquivo, deixando apenas a definição das rotas.

Isso permitirá reaproveitar toda a aplicação, trocando apenas o modelo de dados. Para que as operações com o MongoDB funcionem corretamente, precisamos trocar a linha de código `var db = require('../db')` do arquivo pela linha `var mongoose = require('mongoose')`, que fará a importação do *mongoose*.

Consulta de dados

A rota para a página inicial define o retorno de todas as músicas presentes no banco de dados. Redefinindo o código de acesso aos dados como mostrado a seguir, a nossa aplicação passa a utilizar os módulos que acessam o MongoDB para fazer a consulta de dados.

```
router.get('/', (req, res, next) => {
  mongoose.model('Musica').find().then((musicas) => {
    res.render('index', {
      musicas: musicas
    });
  }, next);
});
```

Observe que a função `model()` acessa o *schema* definido anteriormente, que corresponde à tabela no modelo relacional. Em seguida, a função `find()` recupera todos os registros, colocando-os na variável `musicas`. O processo de renderização da página é idêntico ao feito anteriormente.

Inserção de dados

Faremos o mesmo procedimento com a rota de inserção de dados.

```
router.post('/', (req, res, next) => {
  var Musica = mongoose.model('Musica');
  var m = new Musica(req.body);

  m.save().then((result) => {
    res.redirect('/');
  }, next);
});
```

A variável `Musica` recebe uma referência ao *schema* criado. Com ela, é possível criar um novo objeto, que recebe os dados do

formulário enviado. Esse objeto definido pelo *mongoose* acessa um método `save()` que faz a validação dos dados, e caso obtenha sucesso, chama a função `then()`. Caso contrário, a função `next()` trata os erros ocorridos durante a execução.

Edição de dados

Ambas as rotas de acesso do formulário de edição de músicas e alteração de dados precisam sofrer alterações. Na primeira, a função `findOne()` do *mongoose* recebe um parâmetro para realizar a busca (`id`), retornando os dados para a página, como mostra o código a seguir:

```
router.get('/edit/:id', (req, res, next) => {  
  const {id} = req.params;  
  
  mongoose.model('Musica').findOne({_id: id}).then((musica) => {  
    res.render('edit.njk', {musica: musica});  
  }, next);  
});
```

Na segunda, a função `update()` recebe o parâmetro que indica o identificador do objeto a ser alterado e, em seguida, define os parâmetros que serão alterados. A sintaxe desse método é bem próxima à usada pelo MongoDB, tornando o processo bem intuitivo. Você pode utilizar o código a seguir para alterar a rota:

```
router.put('/edit/:id', (req, res, next) => {  
  const {id} = req.params;  
  
  mongoose.model('Musica').update( { _id: id },  
    {  
      $set: {  
        nome: req.body.nome,  
        artista: req.body.artista,  
        estrelas: req.body.estrelas  
      }  
    }  
  );  
});
```

```

    }).then((musica) => {
      res.redirect('/')
    },next);
  });

```

Com essas alterações, a função de edição da página passa a funcionar normalmente com o banco de dados MongoDB.

Exclusão de dados

A última operação a ser resolvida é a exclusão de músicas. A função `remove()` recebe o `id` como parâmetro e apaga o registro do banco de dados:

```

router.delete('/delete/:id', (req,res,next)=>{
  const {id} = req.params;

  mongoose.model('Musica').remove({_id: id}).then((musica) => {
    res.redirect('/')
  },next);
});

```

11.6 CONSIDERAÇÕES

Neste capítulo, foi possível observar como é simples alterar e inserir módulos de acesso a dados em uma aplicação que pré-configurada (rotas, páginas web, express). O módulo *mongoose* não é um acessório obrigatório para realizar as operações com dados envolvendo Node.js e MongoDB. Mas ele traz uma série de recursos que simplificam as operações de manipulação dos registros, além de fornecer uma camada que cuida da lógica de negócios.

A arquitetura do *mongoose* possibilita separar em módulos os *schemas* de definição dos dados, além de possibilitar a criação de

métodos utilitários (*helper methods*), não explorados neste livro.

SESSIONS

Uma das tarefas mais comuns em aplicações web é a possibilidade de manter o usuário conectado em sua conta, enquanto navega entre diversas páginas, sem ter de ficar fazendo um novo *login* a cada mudança. Mas para desenvolver a *feature* (ferramenta) que habilita o servidor a disponibilizar essa funcionalidade, é preciso entender um pouco sobre como o protocolo *HTTP* funciona.

A ideia de permanecer conectado ao servidor enquanto navega é realizada a partir das *sessions*. Uma *session* controla o estado da conexão entre cliente e servidor, mesmo que o protocolo usado para a comunicação seja sem estado, como é o *HTTP*.

12.1 HTTP STATELESS

Dizemos que o protocolo *HTTP* é *stateless*, ou seja, sem estado. Isso significa que uma requisição feita com esse protocolo permite ao cliente (no caso, o navegador) pedir informações ao servidor sem criar uma conexão permanente.

Este comportamento inerente ao *HTTP* garante que não existam transferências desnecessárias de dados entre o cliente e o servidor enquanto uma requisição não é feita pelo cliente,

deixando o servidor livre para atender novas requisições.

Por outro lado, uma série de dificuldades técnicas surgem quando se deseja manter o estado da conexão (como um usuário conectado em sua conta). O servidor precisa, de alguma forma, saber que em um determinado momento vai receber uma série de requisições vindas de um mesmo cliente, que acabou de estabelecer uma **conexão estável** (*login*).

Login com sessões

Um dos recursos utilizados para desenvolver uma **conexão estável** é a criação de sessões. Uma das formas de se desenvolver essa *feature* é utilizando *cookies*.

Um *cookie* é um arquivo de texto que contém um par `id : value` que pode ser armazenado pelo navegador, quando, por exemplo, acessa uma página web. Esse arquivo pode ser transmitido ao servidor quando a requisição *http* é realizada.

Quando o usuário acessa a sua página de *login*, ele recebe um formulário para preencher com seus dados de acesso (nome e senha, por exemplo). Quando o formulário é enviado, o servidor cria uma **sessão**, que também é um par `id : value` responsável por identificar a pessoa que *está se conectando*.

A sessão fica salva no servidor (em memória ou banco de dados), que transmite, junto aos dados solicitados pelo cliente, um arquivo de *cookie*. A cada nova requisição feita pelo cliente, o arquivo de *cookie* é retransmitido, permitindo ao servidor identificar o usuário que *se encontra conectado*.

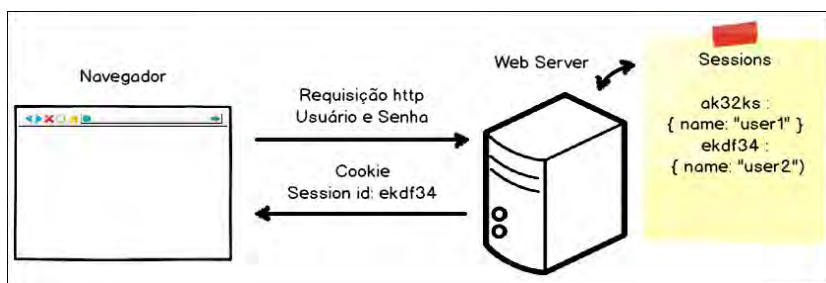


Figura 12.1: Processo de criação de sessão no servidor

Caso o *cookie* seja apagado (tanto pelo cliente como pelo servidor), a conexão é *encerrada*, já que o servidor não possui mais recursos para identificar o cliente solicitante.

Fluxo de navegação

A figura adiante ilustra uma das formas de fluxo da navegação do usuário, após o processo de *login* em um servidor de aplicações *Express*. A cada nova solicitação de página (ou dados), o navegador envia um *cookie* com o *id* do cliente, além dos recursos solicitados (1).

O servidor *Express* atende o pedido, e busca em sua base de sessões ativas o *id* vindo do cliente (2). Caso encontre (3), ele busca na base de dados vinculada à aplicação os dados solicitados (4), renderiza a página web com os dados (5) e então retorna a página solicitada (6).

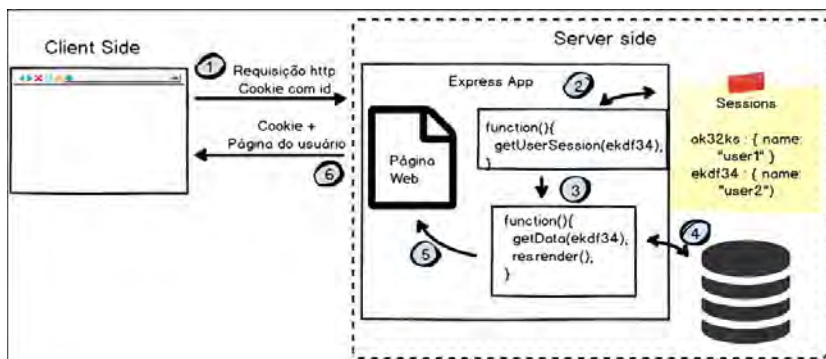


Figura 12.2: Fluxo de navegação em uma sessão iniciada

12.2 APLICAÇÃO COM SESSÕES

Para demonstrar esse processo, vamos implementar o uso de sessões para um usuário que deseja realizar um *login*. Para simplificar, não vamos fazer acesso ao banco de dados, armazenando as sessões em memória. Como usaremos a memória para armazenar as sessões, quando o servidor é reiniciado, todos os clientes conectados são perdidos!

Exemplos do capítulo

Para realizarmos os exemplos deste capítulo, vamos utilizar o projeto que desenvolvemos no capítulo *Nunjucks – template engine*. Você pode baixar o projeto completo no repositório do GitHub do livro (<https://github.com/bmarchete/primeiros-passos-nodejs.git>).

12.3 CONFIGURAÇÃO DO APP.JS

Vamos iniciar instalando um módulo que manipula sessões

junto ao *Express*. Para isso, precisamos abrir o terminal e executar o comando `npm install --save express-session`.

Agora devemos configurar nossa aplicação a partir do arquivo `app.js` para utilizar o módulo de sessões. Para isso, importaremos o módulo com o código `var session = require('express-session')`.

Por fim, precisamos configurar uma chave de segredo para o sistema de criptografia da sessão e outras características do módulo (configurações padrão). Portanto, vamos criar o seguinte código após a configuração dos módulos do *Nunjucks*.

```
app.use(session({
  secret: 'teste sessoes',
  resave: false,
  saveUninitialized: false,
}));
```

As configurações do código permitem que a sessão criada utilize uma frase segredo para gerar os *ids* da sessão (`secret: 'teste sessoes'`), não altere os dados salvos da sessão a cada requisição (`resave: false`) e não force o *Express* salvar uma sessão automática ou não modificada (`saveUninitialized: false`).

12.4 DEFINIÇÃO DO ARQUIVO DE ROTAS

Para a criação das rotas de acesso da aplicação, utilizaremos o arquivo `index.js` (pasta `routes`). A primeira rota deve definir o retorno do formulário de login. Caso a sessão já tenha sido iniciada com uma chave `nome` , a requisição é redirecionada para a rota `/home` .

Vamos inserir o seguinte trecho de código para configurar esta rota:

```
router.get('/', function(req, res, next) {  
  if (req.session.nome) {  
    res.redirect('/home')  
  }else{  
    res.render('login');  
  }  
});
```

A próxima rota define o recebimento dos dados enviados pelo login, e a criação da sessão caso os dados informados sejam corretos. Neste exemplo, o nome do usuário deve ser **admin**, e a senha **1234**. Se a autenticação falhar, a aplicação é redirecionada para o formulário de *login*.

```
router.post('/', (req, res, next)=> {  
  if (req.body.nome == 'admin' && req.body.senha == '1234') {  
    req.session.nome = req.body.nome;  
    res.redirect('/home')  
  }else{  
    res.redirect('/');  
  }  
});
```

A próxima rota que vamos criar define o carregamento da página inicial do usuário. Ela só pode ser carregada caso a sessão já tenha sido iniciada. Caso contrário, a aplicação é redirecionada para a página de *login*.

```
router.get('/home', (req, res, next)=> {  
  if (req.session.nome == 'admin') {  
    res.render('home', {  
      session: req.session,  
      usuario: req.session.nome  
    });  
  }else{  
    res.redirect('/');  
  }  
}
```

```
});
```

A última rota vai definir o encerramento da sessão. Ao acessar o endereço `localhost:3000/logout`, o usuário é desconectado (a sessão é destruída no servidor), redirecionando a aplicação para o formulário de *login*.

```
router.get('/logout', (req, res, next)=> {  
  req.session.destroy();  
  res.redirect('/')  
});
```

12.5 PÁGINAS DA APLICAÇÃO

Para o exemplo apresentado neste capítulo, vamos desenvolver duas páginas bem simples. A primeira (`login.njk`) conterá o formulário de *login* da aplicação. A seguir, é apresentado um exemplo da página.

Você pode observar que a página carrega o *layout* padrão da aplicação. Ela também cria um formulário com dois campos de texto e um botão, para o envio das informações à rota `/` a partir do método `post`, utilizando as formatações padrões do *Bootstrap*.

```
{% extends "layout.njk" %}  
  
{% block pagina %}  
  
<h1 class="page-header">Formulário de Login</h1>  
  
  <form action="/" method="post">  
    <div class="form-group">  
      <label for="nome">Digite o nome</label>  
      <input type="text" name="nome" class="form-control" id="nome" placeholder="Nome">  
    </div>  
  
    <div class="form-group">
```

```

        <label for="senha">Digite a Senha</label>
        <input type="password" name="senha" class="form-control" id:
"senha" placeholder="Senha">
    </div>
    <button type="submit" class="btn btn-primary">Entrar</button>

</form>

{% endblock %}

```

A segunda e última página apenas apresenta uma mensagem de boas-vindas ao usuário conectado. Ela é carregada sempre que o usuário consegue efetuar o login. Quando o usuário não efetua o login, ele é redirecionado para a página com o formulário de login.

```

{% extends "layout.njk" %}

{% block pagina %}

    <h1>0 usuário {{usuario}} está conectado</h1>

{% endblock %}

```

Isso finaliza nossa aplicação. Você pode testar tentando realizar o login a partir da página principal. Observe que, ao realizar o login, surge uma tela com a mensagem *"O usuário x está conectado"*, em que 'x' corresponde ao nome do usuário que se conectou.

12.6 CONSIDERAÇÕES

As sessões não precisam obrigatoriamente tratar de questões de autenticação. De fato, elas são utilizadas em diversas situações, como apresentação personalizada de conteúdo (conforme a navegação do usuário), criação de carrinho de compras etc.

Como visto neste capítulo, diversas configurações precisam ser

realizadas para criar um sistema de autenticação usando sessões. Existem formas mais rápidas e eficientes de se fazer isso!

Como veremos nos próximos capítulos, podemos usar módulos especializados de autenticação, que usam em suas implementações o sistema de sessões.

AUTENTICAÇÃO COM PASSPORT

No capítulo anterior, vimos como utilizar sessões para criar um sistema simples de autenticação. A aplicação criada fazia um controle básico de *login*. Apesar disso, lidar com o controle de autenticação requer muito cuidado, pois envolve diversas práticas de segurança (*hashing* de senhas, por exemplo), ou comportamentos específicos para cada rota acessada pelo usuário autenticado.

Por esses e outros motivos, módulos que auxiliem no controle de um sistema de autenticação, e que forneçam ferramentas simplificadas, são uma excelente opção. Neste capítulo, vamos conhecer o módulo *Passport*, um dos mais famosos para lidar com controle de autenticação.

13.1 MIDDLEWARE DE AUTENTICAÇÃO

O *Passport* é tido como um *middleware*. Um *middleware* é uma ferramenta que pode ser acoplada à aplicação, integrando uma nova funcionalidade a ela. Eles facilitam o acoplamento de ferramentas, e simplificam o processo de desenvolvimento. O *Express* permite que *middlewares* sejam adicionados ao seu

mecanismo de rotas.

Além disso, o *Passport* tem uma diversidade de *strategies* para lidar com as várias formas de se fazer autenticação: *login* tradicional, *login* com *OAuth*, *login* com *APIs* sociais (*Facebook*, *Twitter*), dentre outros.

Para o exemplo apresentado neste capítulo, vamos construir um sistema de *login* tradicional, utilizando *local strategy*. Nesta configuração, o usuário precisa apenas inserir um *username* e *password*. O processo é um pouco mais complexo com outras *strategies*, que envolvem, por exemplo, o uso de tokens para autenticação.

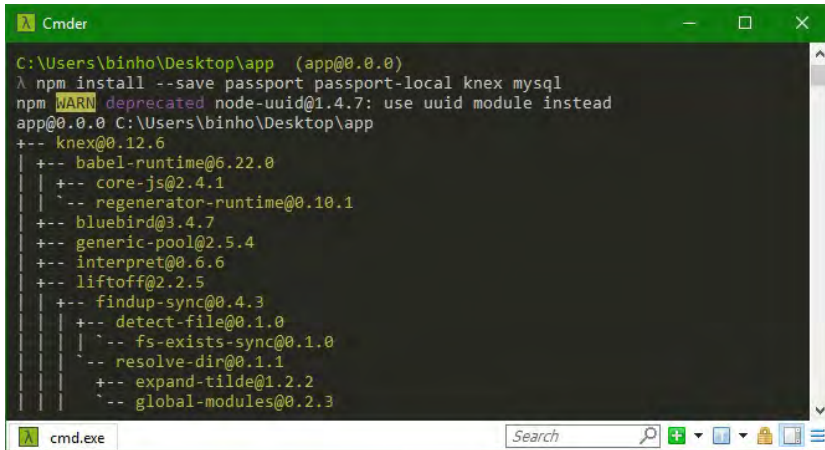
13.2 EXEMPLOS DO CAPÍTULO

Vamos executar uma série de procedimentos para configurar uma aplicação com autenticação utilizando o módulo *passport*. Por isso, precisamos recuperar o projeto desenvolvido no capítulo *Sessions*. Você pode baixar o projeto completo no repositório do GitHub do livro (<https://github.com/bmarchete/primeiros-passos-nodejs.git>).

Instalação dos módulos

Para iniciarmos a configuração do nosso projeto, precisamos instalar os módulos do *passport*. Precisamos também configurar a aplicação para se conectar com o *MySQL*. Para isso, basta abrir um terminal acessando a pasta do projeto, e digitar o comando:

```
npm install --save passport passport-local knex mysql .
```



```
C:\Users\binho\Desktop\app (app@0.0.0)
λ npm install --save passport passport-local knex mysql
npm WARN deprecated node-uuid@1.4.7: use uuid module instead
app@0.0.0 C:\Users\binho\Desktop\app
+-- knex@0.12.6
| +-- babel-runtime@6.22.0
| | +-- core-js@2.4.1
| | -- regenerator-runtime@0.10.1
+-- bluebird@3.4.7
+-- generic-pool@2.5.4
+-- interpret@0.6.6
+-- liftoff@2.2.5
| +-- findup-sync@0.4.3
| | +-- detect-file@0.1.0
| | | -- fs-exists-sync@0.1.0
| | -- resolve-dir@0.1.1
| | +-- expand-tilde@1.2.2
| | -- global-modules@0.2.3
```

Figura 13.1: Instalação dos módulos da aplicação

O módulo `passport-local` permite usar uma estratégia de autenticação bem simples, em que o usuário entra com seu e-mail e senha para acessar a aplicação.

Arquivo `db.js`

Agora devemos fazer as configurações de conexão com o banco de dados. Para este exemplo, você pode criar um banco de dados com o nome `autenticacao`. Feito isso, vamos criar um novo arquivo na pasta padrão, chamado `db.js`. Vamos inserir as configurações de conexão com o banco de dados nele.

O código, como visto no capítulo *Integração com MySQL*, é apresentado a seguir:

```
var knex = require('knex');

var db = knex({
  client: 'mysql',
  connection: {
```

```
    host : '127.0.0.1',  
    user : 'root',  
    database : 'autenticacao'  
  }  
});  
  
module.exports = db;
```

Como visto no capítulo *Integração com MySQL*, podemos usar esse módulo toda vez que precisarmos de uma nova conexão.

Arquivo passport.js

Neste momento, precisamos fazer as configurações do `passport`. Elas servirão de base para que o *Express* consiga fazer a autenticação de novos usuários.

Para isso, devemos criar um novo arquivo na pasta padrão chamado `passport.js`. Neste arquivo, colocaremos as configurações de autenticação da nossa aplicação, fazendo com que o módulo reconheça e saiba lidar com o *login* de um usuário.

Vamos importar os módulos que serão utilizados no arquivo. Isso permitirá a manipulação de cada um deles no contexto do arquivo, como mostra o código a seguir:

```
var db = require('./db');  
var passport = require('passport');  
var LocalStrategy = require('passport-local').Strategy;
```

Observe que o módulo `LocalStrategy` recebe a configuração de estratégia para autenticação. Ele será usado para definir as funções de autenticação do sistema.

O `passport` requer um objeto que contenha as configurações (ou estratégia) de autenticação, pois ele consegue lidar com

diversas delas. Cada estratégia recebe como parâmetro uma série de configurações que vão definir o funcionamento do mecanismo de autenticação.

Para configurar a estratégia de autenticação local, é preciso adicionar ao módulo `passport` o objeto `LocalStrategy`. Ele recebe uma função *callback* que lida com a autenticação.

```
passport.use(new LocalStrategy(username,password,done)=>{  
  
});
```

Observe os parâmetros `username`, `password` e `done`. Quando os dados de nome do usuário e senha forem recebidos, eles serão capturados por esta função, que lidará com o processo de autenticação. O parâmetro `done` representa uma função que tem as seguintes características:

- `done(null, false)` – Indica que as credenciais estão inválidas;
- `done(null, user)` – Repassa os dados do usuário autenticado;
- `done(err)` – Repassa o erro apresentado no processo de autenticação (banco de dados indisponível, por exemplo).

Uma vez recebidos os dados da autenticação, a aplicação precisa acessar o banco de dados e verificar a existência do usuário, bem como se a senha está correta.

```
passport.use(new LocalStrategy((username,password,done)=>{  
  db("users")  
    .where("username", username)  
    .first()
```

```

        .then((user)=>{
            if(!user || user.password !== password){
                return done(null, false);
            }

            done(null, user)
        }, done);
    }));

```

Quando o processo de autenticação estiver finalizado, falta registrar no *cookie* da sessão uma informação que identifique o usuário que está autenticado. Para isso, o `passport` deve ser configurado com outras duas funções.

A função `serializeUser()` é acionada quando o *cookie* da sessão criada é transmitido ao cliente. Ela permite inserir dados da sessão neste *cookie*, e deve ser utilizada para identificar o usuário que está conectado.

Para isso, ela recebe os dados do `user` e a função `done` para repassar os dados para a aplicação. Uma implementação da função se encontra no código a seguir:

```

passport.serializeUser((user, done)=>{
    done(null, user.id);
});

```

Já a função `deserializeUser()` faz o processo inverso. Ela recebe o `id` armazenado no *cookie* vindo do cliente, e resgata no banco de dados os valores do usuário autenticado. Isso garante que a aplicação tenha os dados do usuário autenticado em questão.

```

passport.deserializeUser((id, done)=>{
    db("users")
        .where("id", id)
        .first()
        .then((user)=>{
            done(null, user)
        })
});

```

```
    }, done);  
  })
```

Desta forma, concluímos o módulo de autenticação da aplicação. Basta agora configurar a aplicação para que se conecte com o módulo criado.

13.3 ARQUIVO APP.JS

No arquivo `app.js`, precisamos fazer a importação dos módulos de configuração do `passport`, bem como as configurações de acoplamento ao *Express*. Para realizar este processo, precisamos começar importando os módulos de configuração do `passport`:

```
var passport = require('passport');  
require('./passport');
```

Em seguida, cabe a nós também configurar o `passport` como um *middleware* de autenticação da aplicação. A função `use()` do *Express* permite inserir estes *middlewares* e suas configurações.

Neste exemplo que estamos desenvolvendo, o `passport` vai utilizar o mecanismo de *sessions* do *Express* para controlar a autenticação. Essa integração garante que os *cookies* da sessão funcionem corretamente.

Sendo assim, usamos a função `initialize()` para carregar as configurações do `passport`, e a função `session()` para acoplarmos o `passport` ao mecanismo de sessão. Esse procedimento deve ser realizado depois que a sessão já foi inicializada.

```
app.use(session({  
  secret: 'teste sessoes',
```

```
    resave: false,  
    saveUninitialized: false,  
  }));  
  
app.use(passport.initialize());  
app.use(passport.session());
```

13.4 ARQUIVO DE ROTAS

O arquivo com as rotas necessárias para um processo de autenticação já havia sido desenvolvido por nós no capítulo *Sessions*. Neste momento, precisamos apenas refatorá-lo para incluir o `passport` como mecanismo de autenticação.

A rota principal deve fornecer a página de login ao usuário, caso ele ainda não esteja conectado. Caso contrário, ela deve redirecioná-lo para a sua página inicial (`/home`). Essa verificação é possível graças à função `isAuthenticated()` que o `passport` anexa ao objeto de requisição.

```
router.get('/', (req, res, next)=> {  
  if (req.isAuthenticated()) {  
    res.redirect('/home')  
  }else{  
    res.render('login');  
  }  
});
```

Ao submeter o formulário, a rota principal recebe por `post` os dados do formulário, repassando-os para a função do `passport` responsável pela autenticação. Essa função recebe como parâmetros a estratégia de autenticação e as opções `successRedirect` para o caso de autenticação validada, ou `failureRedirect` para o caso de erro.

```
router.post('/', passport.authenticate('local',{
```

```
    successRedirect: '/home',  
    failureRedirect: '/error',  
  }));
```

A rota `/home` apresenta a página principal do usuário caso este esteja conectado. Novamente precisamos da função `isAuthenticated` para este tipo de validação. A página a ser renderizada recebe os dados do usuário para impressão.

```
router.get('/home', (req, res, next)=> {  
  if (!req.isAuthenticated()) {  
    return res.redirect('/')  
  }  
  
  res.render('home', {  
    session: req.session,  
    usuario: req.user  
  });  
});
```

Por último, a rota `/logout` termina a sessão do *Express* e redireciona o usuário para a página inicial.

```
router.get('/logout', (req, res, next)=> {  
  req.session.destroy();  
  res.redirect('/')  
});
```

A aplicação neste momento está configurada e pronta para ser testada. Sugiro que você acesse o navegador e, com um usuário cadastrado no banco de dados, efetue *login* e *logout*, observando os resultados.

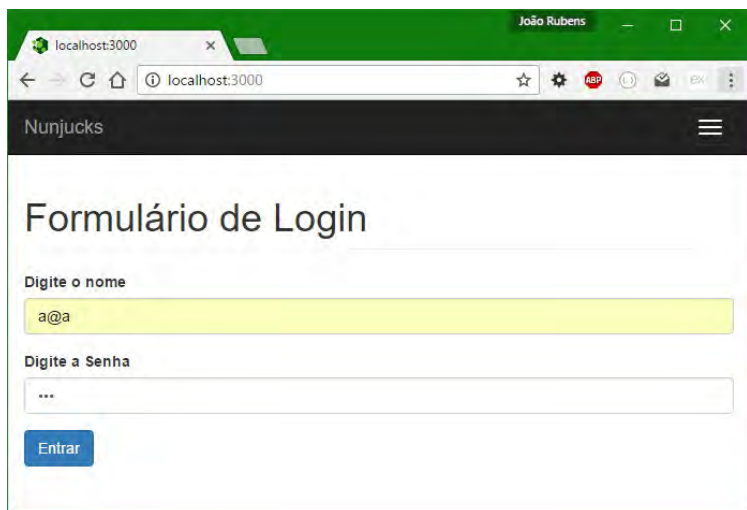


Figura 13.2: Tela de login da aplicação

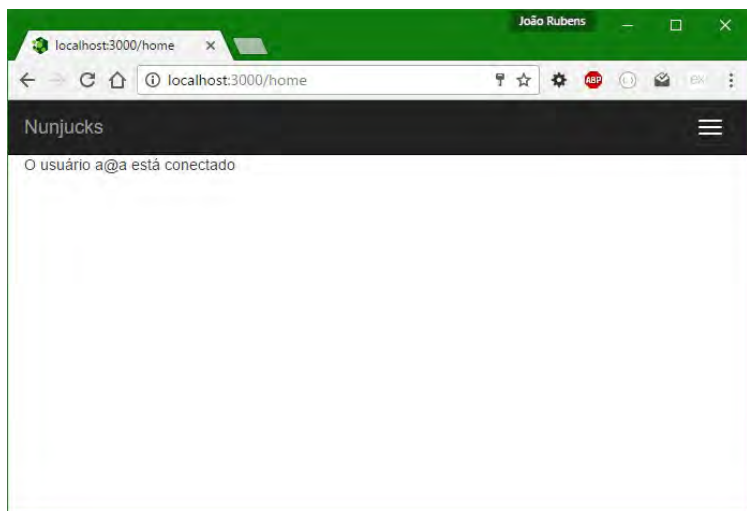


Figura 13.3: Usuário autenticado na aplicação

13.5 CONSIDERAÇÕES

Neste capítulo, vimos como o módulo `passport` lida com as questões de autenticação em uma aplicação *Express*. A partir do mecanismo de *middleware*, o *Express* apresenta uma solução elegante para integrar módulos que vão aperfeiçoando a aplicação.

RESTRIÇÃO DE ACESSO A PÁGINAS

No capítulo anterior, vimos como utilizar módulos (mais especificamente, o *passport*) para manipular a autenticação de usuários em nossa aplicação. Apesar de o *passport* garantir que apenas usuários válidos tenham acesso à aplicação, é importante fazer um filtro de restrição às páginas.

Em uma aplicação real, algumas páginas podem ser acessadas pelo usuário, mesmo se este não estiver autenticado. Já outras devem ser acessadas apenas por usuários autenticados. Neste capítulo, vamos criar um módulo que, com uso de autenticação, permite restringir e liberar acesso a rotas específicas e, conseqüentemente, a páginas.

14.1 EXEMPLOS DO CAPÍTULO

Para realizarmos os exemplos deste capítulo, vamos utilizar o projeto que desenvolvemos no capítulo *Autenticação com Passport*. Você pode baixar o projeto completo no repositório do GitHub do livro (<https://github.com/bmarchete/primeiros-passos-nodejs.git>).

14.2 APLICAÇÃO

Para resolver o problema de acesso restrito às páginas, vamos criar um pequeno *middleware*, que pode ser aplicado às rotas que necessitam de autenticação para serem acessadas. Apesar de muito simples, o *middleware* se integra facilmente ao *Express*, e evita o uso de condicionais dentro das rotas para verificar se um usuário está autenticado ou não.

14.3 MIDDLEWARE PARA RESTRIÇÃO DE ACESSO

Dentro da pasta raiz de nossa aplicação, vamos criar uma pasta chamada `middlewares` para armazenar o módulo de restrição. Em seguida, devemos criar também um arquivo chamado `authorize.js` dentro desta pasta.

O objetivo deste arquivo é expor um módulo que contenha duas funções. Para isso, dentro do arquivo `authorize.js`, vamos criar uma variável chamada `authorize`, para encapsular as funções, além de exportá-la como módulo.

```
var authorize = {  
  
};  
  
module.exports = authorize;
```

A primeira função dentro da variável verifica se já existe um usuário autenticado (sessão ativa). Caso sim, a aplicação segue seu fluxo normal. Caso não, a aplicação redireciona o usuário para uma página em específico, aqui configurada como a página que permite ao usuário realizar o login:

```

var authorize = {
  isAuthenticated : function(req, res, next) {

    if (!req.isAuthenticated()) {
      return res.redirect('/login')
    }

    next();
  },
};

```

Vale observar que este módulo precisa estar acoplado ao fluxo das rotas, porque ele utiliza o objeto `req` para resgatar a função `isAuthenticated()` criada pelo `passport`, que verifica se um usuário já efetuou um login. Faremos isso logo mais neste capítulo.

De modo semelhante, mas oposto, a segunda função evita que usuários que já estejam autenticados acessem determinados recursos. Quando isso acontecer, a função redireciona a aplicação para uma página específica, neste caso, a página da rota inicial:

```

var authorize = {

  //...código da função isAuthenticated...

  isAuthenticated : function(req, res, next) {

    if (req.isAuthenticated()) {
      return res.redirect('/')
    }

    next();
  }
};

```

Apesar de simples, essas duas funções evitam que o desenvolvedor repita o processo de verificação de autenticidade para cada rota restrita na aplicação.

14.4 ARQUIVO DE ROTAS

Para acoplar o *middleware* desenvolvido à aplicação, acessamos o arquivo `index.js` dentro da pasta `routes`, e refatoramos as rotas que precisam de restrição de uso. Mas antes, devemos importar o módulo de autorização. Vamos separar as funções em duas chamadas distintas:

```
var isAuth = require('../middlewares/authorize.js').isAuth;
var isNotAuth = require('../middlewares/authorize.js').isNotAuth;
```

Vamos analisar, caso a caso, cada uma das rotas. A rota principal (`/`) vai permitir ao usuário visualizar o que poderia ser correspondente à sua página inicial. Portanto, devemos acoplar a função `isAuth` a ela, já que esta rota só pode ser acessada por usuários autenticados:

```
router.get('/', isAuth, (req, res, next)=> {
  res.render('home', {
    session: req.session,
    usuario: req.user
  });
});
```

A rota `/login` apresenta o formulário de autenticação de usuários. Quando um usuário está conectado, ele não deve ter acesso a esta página:

```
router.get('/login', isNotAuth, (req, res, next)=> {
  res.render('login')
});
```

As rotas de autenticação de usuários (`/` , método `post`) e de *logout* (`/logout`) podem permanecer inalteradas, como

apresentadas no capítulo *Autenticação com Passport*.

Neste momento, você pode testar as rotas e verificar que, quando não conectado, consegue apenas acesso à página de *login*. Quando conectado, apenas acesso à página de painel!

14.5 CONSIDERAÇÕES

O intuito deste capítulo foi apresentar a simplicidade com que é possível definir níveis de acesso à rotas no *Express*. Você pode criar, em um *middleware*, regras que distinguem, por exemplo, administradores de usuários comuns.

Vale destacar mais uma vez o papel importante do módulo `passport` com relação ao processo de autenticação de usuários, pois utilizamos esse recurso durante todo o processo descrito neste capítulo.

APLICAÇÃO FINAL

Neste capítulo, vamos criar uma aplicação completa, do começo ao fim, explorando todos os recursos vistos no livro. Assim, você pode rever os conceitos aqui aprendidos, além de sanar algumas dúvidas que podem ter surgido durante a leitura dos capítulos anteriores.

Como vamos explorar conceitos já apresentados, os detalhes de implementação (o que cada linha de código faz) do projeto proposto não serão rigorosamente pontuados. A cada passo da criação da aplicação apresentada neste capítulo, sugiro que você visite as seções anteriores do livro para lembrar dos detalhes da implementação.

Antes de começar, certifique-se de que você tenha todas as ferramentas exploradas anteriormente instaladas e funcionando corretamente. Espero que você esteja ansioso para criar uma aplicação completa!

15.1 APLICAÇÃO - CONTROLE DE TAREFAS

Vamos desenvolver uma aplicação que permita ao usuário controlar uma lista de tarefas. Desta forma, usuários autenticados poderão adicionar novas tarefas, que conterão um título e uma

descrição, além de alterar e remover tarefas existentes. Usuários não autenticados terão permissão apenas para visualizar as tarefas criadas.

A aplicação deverá conter um sistema de *login*. Este sistema abrange um formulário para *login*, um formulário para registro (caso o usuário não esteja registrado) e um link para *logout*.

Neste exemplo, será adotado o MySQL como SGBD, mas você pode optar por outros (como o MongoDB). O *id* das tarefas será gerado automaticamente pelo MySQL.

15.2 PÁGINAS WEB

A aplicação conterá 5 páginas:

- **Inicial** - Apresentará a lista de tarefas cadastradas (em forma de cartões), além dos *links* para adição, alteração e exclusão de tarefas (a exclusão será realizada diretamente na página inicial);
- **Adição** - Conterá um formulário para preencher os dados de uma nova tarefa;
- **Alteração** - Conterá um formulário para realizar a alteração dos dados de uma tarefa;
- **Login** - Conterá um formulário para que um usuário registrado possa se conectar;
- **Registro** - Conterá um formulário que permite a criação de novos usuários.

A seguir, você pode visualizar o resultado final de cada página.

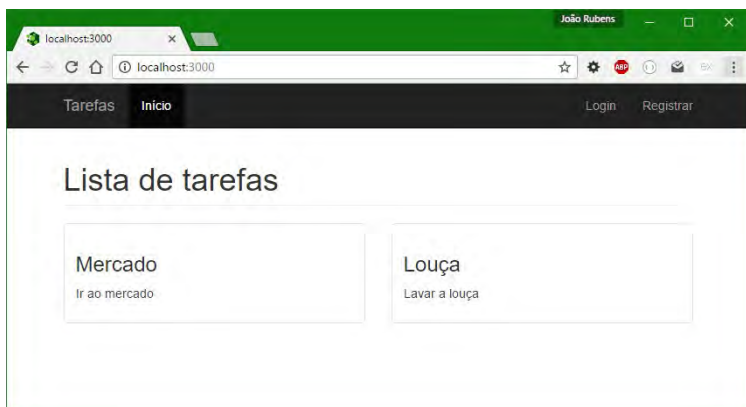


Figura 15.1: Página inicial (usuário não conectado)

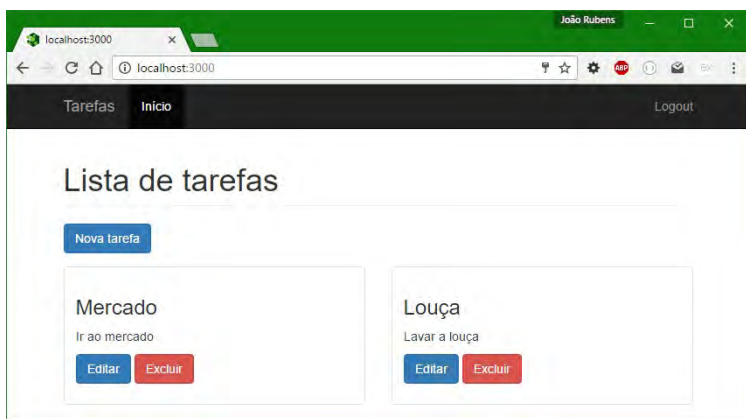


Figura 15.2: Página inicial (usuário conectado)

The screenshot shows a web browser window with the address bar displaying 'localhost:3000/login'. The browser's title bar includes the name 'João Rubens'. The page has a dark green header with a navigation menu containing 'Tarefas' and 'Início', and links for 'Login' and 'Registrar'. The main content area is titled 'Formulário de Login' and contains two input fields: 'Nome' and 'Senha'. Below the 'Senha' field is a blue button labeled 'Entrar'.

Figura 15.3: Página de login de usuários

The screenshot shows a web browser window with the address bar displaying 'localhost:3000/register'. The browser's title bar includes the name 'João Rubens'. The page has a dark green header with a navigation menu containing 'Tarefas' and 'Início', and links for 'Login' and 'Registrar'. The main content area is titled 'Novo usuário' and contains two input fields: 'Nome de acesso' and 'Senha'. Below the 'Senha' field are two buttons: a blue 'Registrar' button and a white 'Voltar' button.

Figura 15.4: Página de registro de usuários

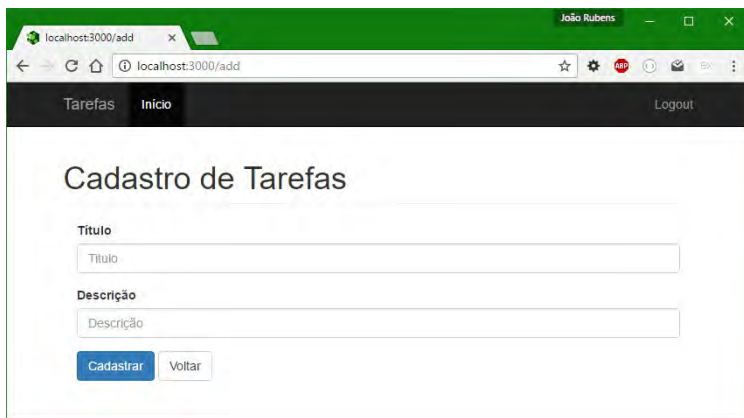


Figura 15.5: Página de adição de novas tarefas

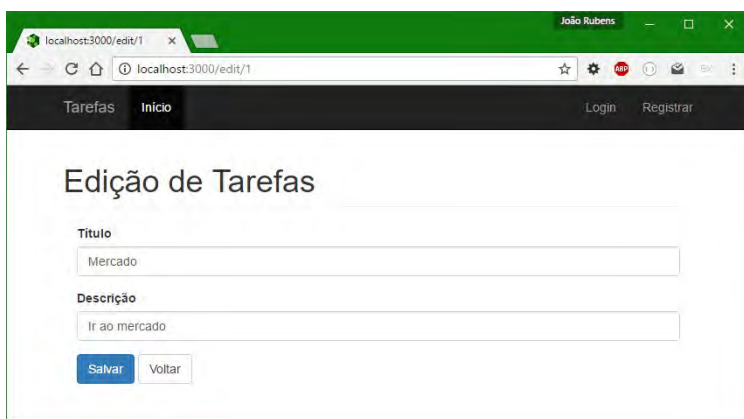


Figura 15.6: Página de edição de tarefas

15.3 CONFIGURAÇÃO DO EXPRESS

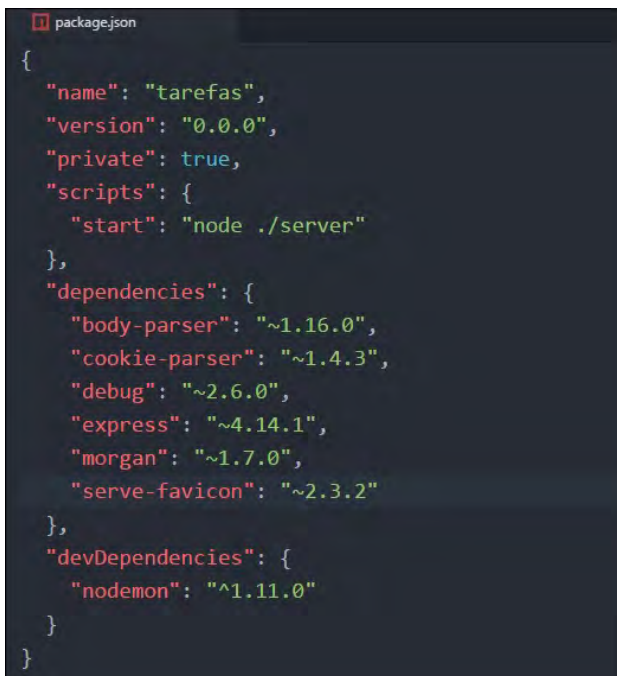
Inicialmente, precisaremos criar uma nova aplicação *Express*. Criaremos uma aplicação dentro da pasta *tarefas*, acessando o prompt de comando. Vamos inserir o comando `express tarefas`.

Ao finalizar a instalação, uma nova pasta chamada `tarefas` é criada. Dentro dela, vamos mover o arquivo criado na pasta `bin` para a pasta raiz do projeto, renomeando-o para `server.js`. Este é o padrão que temos adotado durante todo o livro. Não esqueça de excluir a pasta `bin` !

Por conta desta alteração, dentro do arquivo `server.js`, precisamos apontar que o arquivo `app.js` agora se encontra no mesmo diretório. Para isso, alteramos a chamada do módulo `app` para `var app = require('./app')`.

Para iniciarmos nosso servidor e podermos monitorá-lo de acordo com cada alteração, vamos instalar como dependência de desenvolvedor no nosso projeto o módulo `nodemon`, a partir do comando `npm install --save-dev nodemon`.

Dentro do arquivo `package.json`, você já pode configurar o *script* de execução, apontando para o arquivo `server`. Você pode remover também a referência ao módulo `jade`, que não será usado neste projeto.



```

package.json
{
  "name": "tarefas",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./server"
  },
  "dependencies": {
    "body-parser": "~1.16.0",
    "cookie-parser": "~1.4.3",
    "debug": "~2.6.0",
    "express": "~4.14.1",
    "morgan": "~1.7.0",
    "serve-favicon": "~2.3.2"
  },
  "devDependencies": {
    "nodemon": "^1.11.0"
  }
}

```

Figura 15.7: Estado atual do arquivo package.json

Com estas alterações realizadas, precisamos instalar os módulos listados até aqui nas dependências do projeto. Fazemos isso executando o comando `npm install`.

Já é possível ver se a aplicação está rodando. Inicie o servidor com o comando `nodemon`, e navegue até o endereço `localhost:3000`. Como removemos as dependências do `jade` de nossa aplicação, você provavelmente vai visualizar uma página com erro, conforme a figura a seguir. Isso não é um problema na verdade, pois indica que o servidor foi iniciado com sucesso!

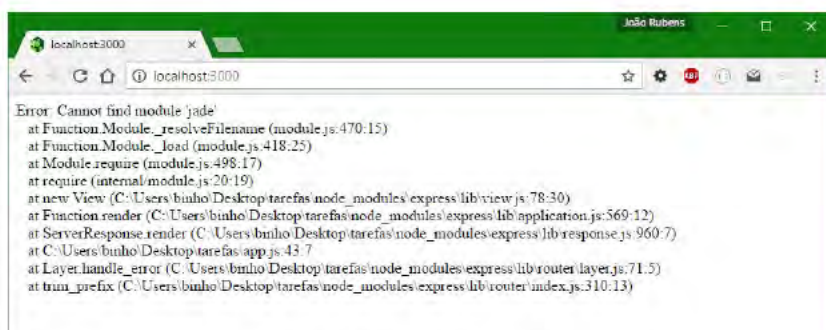


Figura 15.8: Servidor Express iniciado

15.4 MÓDULOS ADICIONAIS

Instalaremos agora os módulos adicionais que usaremos durante o projeto:

- `nunjucks` – Bibliotecas do Nunjucks *template engine*;
- `express-nunjucks` – Conexão com Nunjucks *template engine*;
- `express-session` – Manipulação de sessões;
- `mysql` – Acesso ao SGBD MySQL;
- `knex` – Manipulação de dados oriundos do MySQL;
- `method-override` – Manipulação de verbos *http* não tratados por formulários *HTML*;
- `passport`, `passport-local` – Manipulação de autenticação com usuário e senha;

Todos estes módulos podem ser instalados com apenas uma linha de comando:

```
npm i --save nunjucks express-nunjucks express-session mysql knex  
method-override passport passport-local
```

Nunjucks

Como visto no capítulo *Nunjucks template engine*, devemos configurar o arquivo `app.js` para trabalhar com o *Nunjucks*. Para isso, na sessão de importação de pacotes, inserimos o código `var expressNunjucks = require('express-nunjucks')`.

Em seguida, logo após a inicialização da função `var app = express()`, trocamos as linhas de `view engine setup` para:

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'njk');
var njk = expressNunjucks(app, {watch: true, noCache: true});
```

Desta forma, podemos criar novos arquivos `.njk` dentro da pasta `views`.

Módulo de autenticação

Faremos agora a importação dos módulos que cuidarão do nosso sistema de autenticação, o `passport` e o `express-session`, na sessão de importações:

```
var express-session = require('express-session');
var passport = require('passport');
```

Logo após esta sessão, vamos importar o arquivo que conterá todas as configurações do mecanismo de autenticação. Para isso, inserimos o código `require('./passport')`. Vamos aproveitar e criar um arquivo na raiz do projeto chamado `passport.js`. Faremos suas configurações mais tarde.

Agora nos falta inserir as configurações para atrelar o `passport` ao sistema de sessões. Faremos isso logo após as linhas de configuração do `nunjucks`:

```
app.use(session({
  secret: 'teste sessoes',
  resave: false,
  saveUninitialized: false,
}));

app.use(passport.initialize());
app.use(passport.session());
```

Method override

Neste momento, vamos configurar a aplicação para interpretar formulários que enviarão pedidos de alteração e exclusão de dados, mapeando esses pedidos para os verbos *http* `put` e `delete`.

Para isso, na sessão de importações, inserimos o módulo `method-override` a partir do código `var methodOverride = require('method-override')`. Já o código de configuração do módulo deve ser inserido após as configurações do módulo `passport`:

```
app.use(methodOverride(function (req, res) {
  if (req.body && typeof req.body === 'object' && '_method' in req.body) {
    var method = req.body._method
    delete req.body._method
    return method
  }
}));
```

Arquivos de rotas

Somente nos resta configurar os arquivos de rota da aplicação. Usaremos dois: o primeiro para configurar as rotas do sistema de autenticação, e o segundo para configurar as rotas de manipulação dos dados.

Para tanto, vamos renomear os arquivos da pasta `routes` para `auth.js` e `tarefas.js`. O código que já se encontra dentro deles pode ser mantido para reaproveitarmos mais adiante.

Vamos ajustar a chamada para estes arquivos, mudando a linha `var index = require('./routes/index')` para `var auth = require('./routes/auth')`. Também mudaremos `var users = require('./routes/users')` para `var tarefas = require('./routes/tarefas')`.

Por fim, trocamos a inclusão destes arquivos como *middlewares*, de `app.use('/', index)` para `app.use('/', auth)`, e de `app.use('/users', users)` para `app.use('/', tarefas);`.

15.5 CONFIGURAÇÃO DE ACESSO AO BANCO DE DADOS

Como visto no capítulo *Integração com MySQL*, podemos criar um arquivo que apresente as configurações de acesso ao banco de dados. Repetiremos o processo criando um arquivo `db.js` na raiz do projeto, e inserindo os dados para acessar um banco de dados chamado `projeto-final`:

```
var knex = require('knex');

var db = knex({
  client: 'mysql',
  connection: {
    host : '127.0.0.1',
    user : 'root',
    database : 'projeto-final'
  }
});
```

```
module.exports = db;
```

Lembre-se de criar o banco de dados `projeto-final` em seu gerenciador de MySQL!

Tabelas para os dados da aplicação

Devemos também criar as tabelas para armazenar os dados da aplicação. Elas serão duas:

- `users` – Guarda os dados dos usuários que serão registrados e autenticados na aplicação;
- `tarefas` – Guarda os dados das tarefas criadas pelos usuários da aplicação.

O código SQL para cada uma delas será:

```
-- Estrutura da tabela `tarefas`
CREATE TABLE `tarefas` (
  `id` int(11) NOT NULL,
  `titulo` varchar(10) NOT NULL,
  `descricao` varchar(30) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- Estrutura da tabela `users`
CREATE TABLE `users` (
  `id` int(11) NOT NULL,
  `username` varchar(20) NOT NULL,
  `password` varchar(20) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

15.6 CONFIGURAÇÕES DO MÓDULO DE AUTENTICAÇÃO

Como apresentado no capítulo *Autenticação com Passport*, precisamos configurar o módulo `passport`, de modo que sejam definidas a estratégia de autenticação, a função

`serializeUser()` e `deserializeUser`. Neste livro, estamos utilizando configurações bem simples, que nos permitem verificar se o usuário que está tentando se autenticar é válido, repassar o `id` do usuário autenticado para o *cookie* da sessão e resgatar os dados do usuário autenticado a partir do *cookie* da sessão.

Dentro do arquivo `passport.js` já criado, vamos definir a estratégia de autenticação, que acessará o banco de dados para verificar as credenciais do usuário:

```
var db = require('./db');
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;

passport.use(new LocalStrategy((username, password, done)=>{

  db("users")
  .where("username", username)
  .first()
  .then((user)=>{
    if(!user || user.password !== password){
      console.log(user);
      return done(null, false);
    }

    done(null, user)

  }, done);
}));
```

Em seguida, vamos inserir as configurações repassadas para o *cookie* da sessão:

```
passport.serializeUser((user, done)=>{
  done(null, user.id);
});

passport.deserializeUser((id, done)=>{
  db("users")
  .where("id", id)
```

```
.first()  
.then((user)=>{  
  done(null, user)  
}, done);  
})
```

Vale uma pausa para lembrar de que a função `done` carrega consigo o resultado da operação executada, podendo ser repassada como argumento `catch` da *promise*, gerada pela consulta ao banco de dados, como feito no exemplo.

15.7 CONFIGURAÇÕES DAS ROTAS

Vamos começar configurando as rotas de autenticação da nossa aplicação. Para isso, devemos editar o arquivo `auth.js` que se encontra dentro da pasta `routes`.

Mas antes disso, como visto no capítulo *Restrição de acesso a páginas*, vamos criar um pequeno *middleware* que verificará se o usuário se encontra ou não autenticado.

Authorize.js

Devemos criar uma nova pasta na raiz do nosso projeto, chamada *middlewares* e, em seguida, um arquivo chamado `authorize.js`. Nele, vamos definir um novo módulo que apresenta duas funções.

A primeira (`isAuth`) redirecionará a aplicação para a página inicial caso o usuário não esteja autenticado, evitando assim que ele acesse páginas restritas. A segunda (`isNotAuth`) faz o oposto, redirecionando o usuário à página inicial se tentar acessar algum recurso que, quando autenticado, não deveria acessar.

Dentro do arquivo `authorize.js`, o módulo deve estar como apresentado a seguir:

```
var authorize = {
  isAuthenticated : function(req, res, next) {

    if (!req.isAuthenticated()) {
      return res.redirect('/')
    }

    next();
  },

  isAuthenticated : function(req, res, next) {

    if (req.isAuthenticated()) {
      return res.redirect('/')
    }

    next();
  }
};

module.exports = authorize;
```

Auth.js

Agora sim, dentro do arquivo de rotas `auth.js`, já podemos inserir as rotas de controle da autenticação.

Vamos começar definindo os módulos a serem importados. Além do padrão para definir as rotas do *Express*, agora importamos também as configurações para acesso ao banco de dados e o módulo *middleware* criado anteriormente.

```
var express = require('express');
var router = express.Router();
var passport = require('passport');
var isAuthenticated = require('../middlewares/authorize').isAuthenticated;
var db = require('../db');
```

Como neste arquivo de rotas não precisaremos usar a função `isAuth` , importamos apenas a função `isNotAuth` . Você pode remover todo o código já presente nos arquivos de rota já trazidos pela instalação inicial do *Express*, ou alterá-lo conforme apresentado a seguir.

A primeira rota vai definir o acesso à página de login, que pode ser feito apenas se o usuário não estiver autenticado:

```
router.get('/login', isNotAuth, (req ,res ,next)=>{  
  res.render('login');  
});
```

A próxima rota chama o mecanismo de autenticação do `passport` que redirecionará o usuário para a página inicial, caso suas credenciais estejam corretas. Caso contrário, a aplicação apresentará uma página com uma mensagem de erro:

```
router.post('/login', passport.authenticate('local',{  
  successRedirect: '/',  
  failureRedirect: '/error',  
}));
```

A rota subsequente encerra a sessão de um usuário autenticado:

```
router.get('/logout', (req, res, next)=> {  
  req.session.destroy();  
  res.redirect('/')  
});
```

A próxima rota retorna a página com o formulário de registro de novos usuários. Ela também é restrita para usuários não autenticados:

```
router.get('/register', isNotAuth, (req ,res ,next)=>{  
  res.render('register');
```

```
});
```

A última rota acessada apenas por usuários ainda não autenticados tenta fazer o registro de um novo usuário no banco de dados. Caso tudo ocorra bem, o usuário é automaticamente autenticado e redirecionado para a página inicial:

```
router.post('/register', isAuthenticated, (req,res,next)=>{
  db("users").insert(req.body).then((ids) => {
    passport.authenticate('local')(req, res, function () {
      res.redirect('/');
    });
  });

},next)
});
```

Não podemos nos esquecer de, no fim do arquivo, exportar o módulo criado!

```
module.exports = router;
```

Tarefas.js

Nos resta configurar as rotas que farão a criação, consulta, alteração e exclusão de novas tarefas. Como queremos apresentar uma opção de logout que aparecerá apenas para usuários conectados, vamos repassar como parâmetro para as páginas apresentadas neste módulo uma variável que indicará se o usuário encontra-se ou não autenticado.

Usaremos a função `isAuthenticated()` anexada ao objeto `req` pelo `passport`. Registraremos essas rotas no arquivo `tarefas.js`. Precisamos, assim como feito no arquivo `auth.js`, importar as dependências usadas neste módulo:

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

```
var passport = require('passport');
var isAuthenticated = require('../middlewares/authorize').isAuth;
var db = require('../db');
```

Em seguida, definimos a primeira rota, que apresentará a página inicial com os dados de todas as tarefas criadas. Observe que esta rota não tem restrição nenhuma de acesso:

```
router.get('/', (req, res, next) => {
  db("tarefas").then((tarefas)=>{
    res.render('index',{
      tarefas: tarefas,
      isAuthenticated: req.isAuthenticated()
    });
  },next);
});
```

A próxima rota vai definir a página a ser exibida para que o usuário consiga cadastrar novas tarefas. Deve haver um usuário autenticado para que essa rota seja acessada:

```
router.get('/add', isAuthenticated, (req,res,next)=>{
  res.render('add',{
    isAuthenticated: req.isAuthenticated()
  });
});
```

A rota que vem na sequência recebe os dados para inserção de uma nova tarefa no banco de dados, redirecionando a aplicação para a página inicial. Mais uma vez, deve haver um usuário autenticado para que essa rota seja acessada:

```
router.post('/add', isAuthenticated, (req,res,next)=>{
  db("tarefas").insert(req.body).then((ids) => {
    res.redirect('/');
  },next)
});
```

As próximas rotas definirão a página apresentada ao usuário autenticado, para que possa alterar os valores de uma determinada tarefa, e a rota que receberá os dados, alterando-os no banco de dados:

```
router.get('/edit/:id', isAuth, (req,res,next)=>{
  const {id} = req.params;

  db("tarefas")
    .where("id", id)
    .first()
    .then((tarefa) => {
      if (!tarefa) {
        return res.send(400);
      }

      res.render("edit",{
        tarefa: tarefa,
        isAuth: req.isAuthenticated()
      });
    },next);
});

router.put('/edit/:id', isAuth, (req,res,next)=>{
  const {id} = req.params;

  db("tarefas")
    .where('id', id)
    .update(req.body)
    .then((result) => {
      if (result === 0) {
        return res.send(400);
      }
      res.redirect('/');
    },next)
});
```

A última rota define as operações para que um usuário autenticado possa excluir uma tarefa:

```
router.delete('/delete/:id', isAuth, (req,res,next)=>{
```

```

    const {id} = req.params;
    console.log("deletando" + id);
    db("tarefas")
      .where('id', id)
      .delete()
      .then((result) => {
        if (result === 0) {
          return res.send(400);
        }
        res.redirect('/');
      }, next)
  });
});

```

Para concluir o módulo, inserimos o código de exportação:

```

module.exports = router;

```

15.8 CONFIGURAÇÕES DAS PÁGINAS WEB

Para encerrarmos o desenvolvimento da aplicação, vamos criar e configurar as páginas da aplicação. Vamos começar definindo um layout padrão, que conterá acesso ao *Bootstrap framework* e, logo em seguida, inserir o código de cada página. Teremos uma página personalizada que apresentará os erros da aplicação.

Você pode excluir os arquivos de extensão `jade` da pasta `views`, caso não o tenha feito ainda.

Layout.njk

Vamos iniciar criando o arquivo de *layout*. Chamaremos estes de `layout.js`. O arquivo começará definindo o *template* padrão *html*, além de fazer referência às bibliotecas do *bootstrap* por um *link* remoto.

```

<!DOCTYPE html>
<html lang="en">

```



```

<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scal
e=1">
  <title></title>

  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bo
otstrap/3.3.5/css/bootstrap.min.css">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/fo
nt-awesome/4.4.0/css/font-awesome.min.css">
</head>
<body>

  <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.
3/jquery.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js
/bootstrap.min.js"></script>
</body>
</html>

```

Em seguida, dentro de `<body>` , vamos criar uma barra de navegação, que conterà os itens de menu da aplicação. Observe que as diretivas do `nunjucks` permitem imprimir corretamente as opções de menu para quando o usuário se encontra conectado ou não, acessando a variável repassada pelas rotas definidas anteriormente:

```

<nav class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" dat
a-toggle="collapse" data-target="#navbar" aria-expanded="false" a
ria-controls="navbar">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#">Tarefas</a>
    </div>
    <div id="navbar" class="collapse navbar-collapse">

```

```

<ul class="nav navbar-nav">
  <li class="active"><a href="/">Início</a></li>
</ul>

<ul class="nav navbar-nav navbar-right">

  {% if isAuth %}

    <li><a href="/logout">Logout</a></li>

  {% else %}

    <li><a href="/login">Login</a></li>
    <li><a href="/register">Registrar</a></li>

  {% endif %}
</ul>

</div>
</div>
</nav>

```

Após a barra de navegação, falta definir o corpo padrão para cada página. Vamos usar o sistema de grids do bootstrap , e definir o local que serão renderizadas todas as páginas pelo nunjucks :

```

<div class="container">
  <div class="row" style="margin-top: 50px;">
    <div class="col-md-12">
      {% block pagina %} {% endblock %}
    </div>
  </div>
</div>

```

Error.jk

Vamos criar uma página para mostrar os erros apresentados

pela aplicação. Não faremos nenhum tratamento específico, porém, daremos uma aparência de página padrão para esse recurso:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scal
e=1">
  <title></title>

  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bo
otstrap/3.3.5/css/bootstrap.min.css">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/fo
nt-awesome/4.4.0/css/font-awesome.min.css">
</head>
<body>

  <h1 class="page-header">Esse recurso retornou o seguinte erro</
h1>

  <p class="alert-danger">{{error}}</p>

</body>
</html>
```

Login.njk

Agora criaremos uma nova página, chamada de `login.njk`, que apresentará um formulário para que o usuário registrado possa se autenticar:

```
{% extends "layout.njk" %}

{% block pagina %}

<h1 class="page-header">Formulário de Login</h1>

  <form action="/login" method="post">
```

```

<div class="form-group">
  <label for="nome">Digite o nome</label>
  <input type="text" name="username" class="form-control" id=
"nome" placeholder="Nome">
</div>

<div class="form-group">
  <label for="senha">Digite a Senha</label>
  <input type="password" name="password" class="form-control"
id="senha" placeholder="Senha">
</div>
<button type="submit" class="btn btn-primary">Entrar</button>

</form>

{% endblock %}

```

Register.njk

Semelhante à página de *login*, precisamos de uma página chamada `register.njk`, para que novos usuários possam se registrar em nossa aplicação:

```

{% extends "layout.njk" %}

{% block pagina %}

<h1 class="page-header">Novo usuário</h1>
<div class="col-md-6">
  <form action="/register" method="post">
    <div class="form-group">
      <label for="username">Nome de acesso</label>
      <input type="text" class="form-control" name="username" id=
"username" placeholder="Nome">
    </div>
    <div class="form-group">
      <label for="senha">Senha</label>
      <input type="text" class="form-control" name="password" id=
"senha" placeholder="Senha">
    </div>

    <button type="submit" class="btn btn-primary">Registrar</butt

```

```

on>
    <a href="/" class="btn btn-default">Voltar</a>
  </form>
</div>
{% endblock %}

```

Index.njk

É na página inicial que vamos apresentar as tarefas cadastradas. Vamos utilizar a variável `isAuth` para apresentar as opções de cadastro, alteração e exclusão de dados (já que apenas usuários autenticados podem realizar estas operações).

Neste exemplo, um componente de *thumbnail* do *bootstrap* será usado em vez de uma tabela, para dar uma aparência mais agradável à página. Além disso, um pequeno *script* será desenvolvido para que o usuário possa receber uma mensagem de confirmação, caso queira excluir uma tarefa.

A função `excluir(id)` cria dinamicamente um novo formulário e insere nele os campos necessários para que, ao ser submetido, encontre a rota que lida com o processo de exclusão de dados.

```

{% extends "layout.njk" %}

{% block pagina %}

<h1 class="page-header">Lista de tarefas</h1>

{% if isAuth %}
<a href="/add" class="btn btn-primary" style="margin-bottom: 15px;">Nova tarefa</a>
{% endif %}

<div class="row">

```

```

{% for tarefa in tarefas %}

<div class="col-sm-6 col-md-4">
  <div class="thumbnail">
    <div class="caption">
      <h3>{{tarefa.titulo}}</h3>
      <p>{{tarefa.descricao}}</p>
      {% if isAuth %}
        <p>
          <a href="/edit/{{tarefa.id}}" class="btn btn-primary":
Editor</a>
          <button class="btn btn-danger" onClick="excluir('{{tar
efa.id}}')">Excluir</button>
        </p>
      {% endif %}
    </div>
  </div>
</div>

{% endfor %}

</div>

<script>
function excluir(id) {
  const r = confirm("Tem certeza que deseja excluir este registro
?");

  if (r) {

    const form = document.createElement("form");
    form.setAttribute("method", "post");
    form.setAttribute("action", "/delete/" + id);

    const deleteInput = document.createElement("input");
    deleteInput.setAttribute("type", "hidden");
    deleteInput.setAttribute("name", "_method");
    deleteInput.setAttribute("value", "delete");

    form.appendChild(deleteInput);

    document.body.appendChild(form);
    form.submit();
  }
}

```

```

    }
}
</script>

{% endblock %}

```

Add.njk

Vamos criar uma página (add.njk) para que o usuário acesse um formulário e, a partir dele, insira novas tarefas:

```

{% extends "layout.njk" %}

{% block pagina %}

<h1 class="page-header">Cadastro de Tarefas</h1>
<div class="col-md-6">
  <form action="/add" method="post">
    <div class="form-group">
      <label for="titulo">Título</label>
      <input type="text" class="form-control" name="titulo" id="t
itulo" placeholder="Título">
    </div>
    <div class="form-group">
      <label for="descricao">Descrição</label>
      <input type="text" class="form-control" name="descricao" id=
"descricao" placeholder="Descrição">
    </div>

    <button type="submit" class="btn btn-primary">Cadastrar</butt
on>
    <a href="/" class="btn btn-default">Voltar</a>
  </form>
</div>
{% endblock %}

```

Edit.njk

Semelhante à página de cadastro de tarefas, precisamos de uma página com um formulário para que o usuário altere os dados de uma tarefa qualquer. Para isso, criaremos uma página chamada

edit.njk . Observe que é necessário inserir um campo oculto no formulário que defina o método `put` a ser recebido pelo *Express*.

```
{% extends "layout.njk" %}

{% block pagina %}

<h1 class="page-header">Edição de Tarefas</h1>
<div class="col-md-6">
  <form action="/edit/{{tarefa.id}}" method="post">
    <input type="hidden" name="_method" value="put">
    <div class="form-group">
      <label for="titulo">Título</label>
      <input type="text" class="form-control" name="titulo" id="t
itulo" value="{{tarefa.titulo}}">
    </div>
    <div class="form-group">
      <label for="descricao">Descrição</label>
      <input type="text" class="form-control" name="descricao" id=
"descricao" value="{{tarefa.descricao}}">
    </div>

    <button type="submit" class="btn btn-primary">Salvar</button>
    <a href="/" class="btn btn-default">Voltar</a>
  </form>
</div>
{% endblock %}
```

15.9 TESTE DA APLICAÇÃO

Ao fim de todas estas configurações, temos nossa aplicação finalizada! Para testar a aplicação criada, basta iniciar o servidor com o comando `nodemon` , e acessar o endereço `localhost:3000` .

Para que as configurações do *Bootstrap* estejam disponíveis, é necessária uma conexão com a internet ativa. Você pode se registrar como usuário, efetuar *login*, cadastrar, alterar, excluir e consultar tarefas na aplicação!

15.10 CONSIDERAÇÕES

Após conhecer diversas ferramentas de desenvolvimento de uma aplicação em Node.js, este capítulo teve por objetivo sintetizar os conhecimentos adquiridos em uma aplicação real. Diversas ferramentas podem ser acrescentadas à aplicação criada, além das já apresentadas.

Como o livro tem por objetivo apresentar os primeiros passos para um desenvolvedor na plataforma Node.js, acredito que você já tenha condições de começar seus projetos, e ir se aprofundando conforme a necessidade.

DEPLOY

Neste capítulo, vamos fazer o *deploy* da aplicação criada no capítulo *Aplicação final*. A partir desse processo, você vai entender melhor como hospedar sua aplicação!

O termo *deploy* se refere à ação de *instalar* a aplicação desenvolvida em um serviço que pode ser acessado diretamente pela internet. Para fazermos o *deploy* da nossa aplicação, precisamos inicialmente escolher um dos serviços disponíveis.

Um serviço de hospedagem bem popular é o *heroku* (<https://www.heroku.com/>). O *heroku*, assim como a *AWS*, nos permite escolher uma configuração inicial de um servidor web para hospedarmos nossa aplicação.

Uma das vantagens é que o *heroku* possui uma conta gratuita e um serviço de *deploy* que não nos obriga fazer configurações complicadas de servidor. Esse serviço disponibiliza uma opção de *deploy* a partir de comandos *git*. Vamos utilizar o *heroku* para fazer nosso *deploy*!

Apesar de você não encontrar nada relacionado ao *git* neste livro, os comandos que vamos utilizar são bem simples, e quase autoexplicativos.

16.1 CONTA NO HEROKU

Vamos começar acessando o site do heroku e criando uma conta:

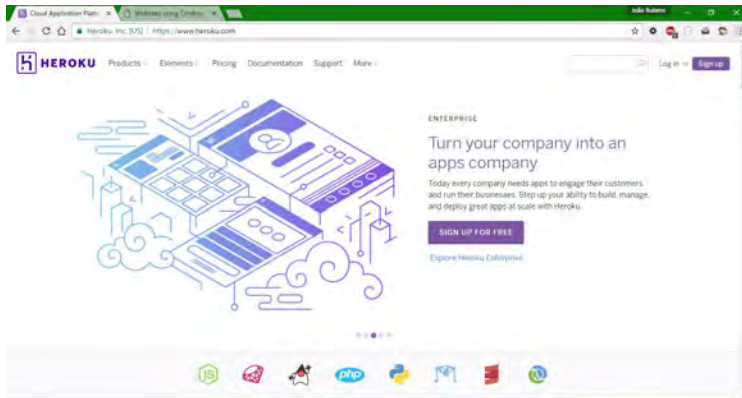


Figura 16.1: Acessando o heroku

Em seguida, clicamos em Sign up for free, e preenchemos o formulário de cadastro. Já aproveitamos para escolher o *Node.js* como linguagem de desenvolvimento primária.

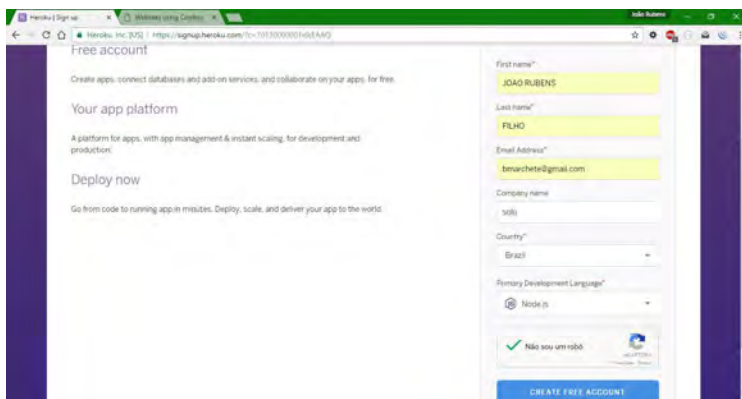


Figura 16.2: Cadastro como novo usuário no heroku

Na tela que segue, inserimos uma senha para completarmos o cadastro. Um e-mail de confirmação será enviado.

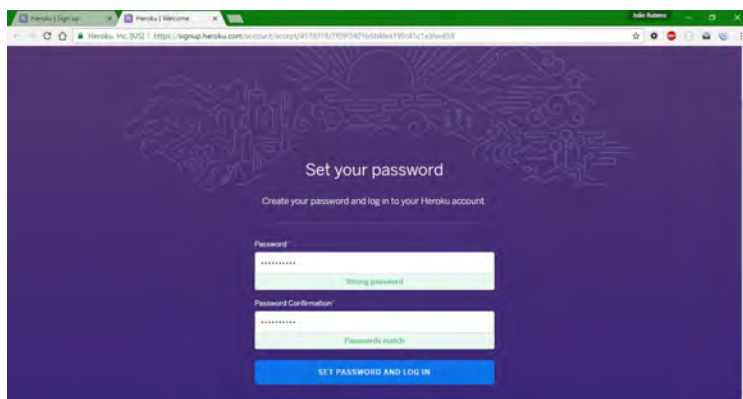


Figura 16.3: Inserção de senha na conta do heroku

Ao confirmar a conta criada, nos deparamos com uma tela que nos permitirá escolher a configuração padrão do servidor web que estamos criando. Você deve selecionar a opção `Node.js - Get Started`.

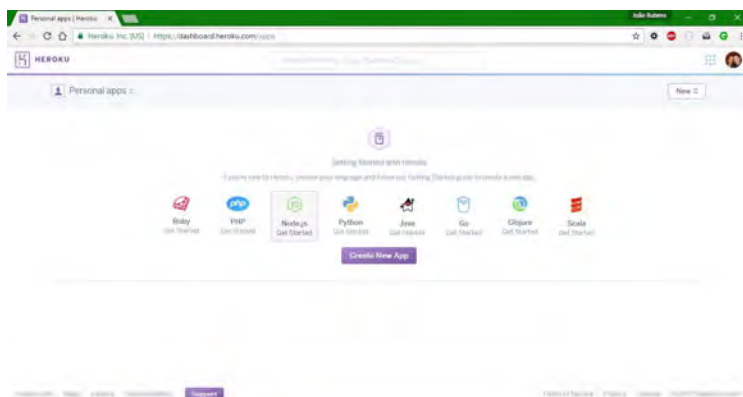


Figura 16.4: Escolha do ambiente de desenvolvimento

16.2 SETUP DO AMBIENTE DE DEPLOY

Após o processo inicial de cadastro, podemos acompanhar um pequeno tutorial que o heroku sugere para fazermos o deploy da aplicação. Vamos começar instalando um aplicativo disponibilizado pelo heroku, para que possamos acessar o serviço de deploy via terminal. Em `setup`, baixe o `Heroku CLI` para sua versão de sistema operacional.

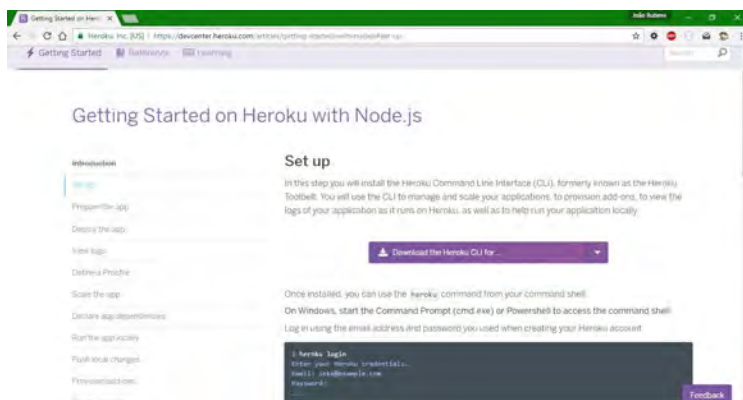


Figura 16.5: Download do CLI para acessar o heroku

Após o download, execute a aplicação baixada. Você pode manter a instalação padrão. Ao fim da instalação, já é possível acessar os serviços do heroku!

Configuração do MySQL

Por padrão, o heroku não vem com um servidor de MySQL. Mas nós podemos adicionar esse recurso a partir do painel do heroku. Esse tipo de serviço exige que você cadastre um cartão de crédito. Apesar disso, não será cobrado nenhum valor, pois

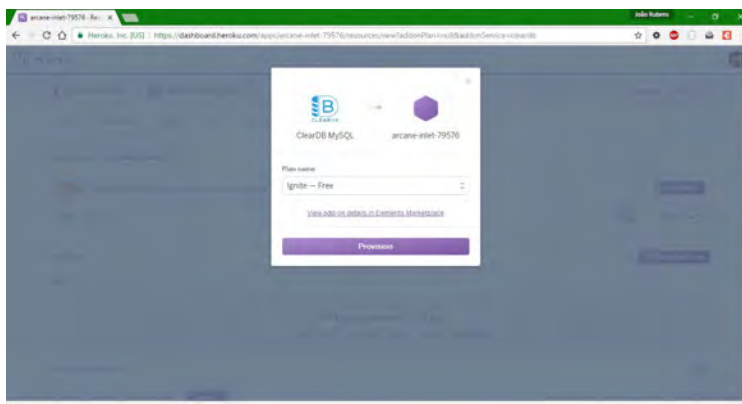


Figura 16.7: Escolha de um serviço de MySQL gratuito

Uma mensagem de erro pode aparecer neste momento, pois é preciso inserir um cartão de crédito válido em sua conta. Isso não lhe cobrará taxa nenhuma, pois a aplicação que vamos testar atende o requisito de gratuidade.

Uma vez que você tenha instalado o *add-on*, você pode acessar a aba `Settings` para verificar as configurações de acesso ao MySQL. Para isso, é preciso clicar na opção `Show Config Vars`.

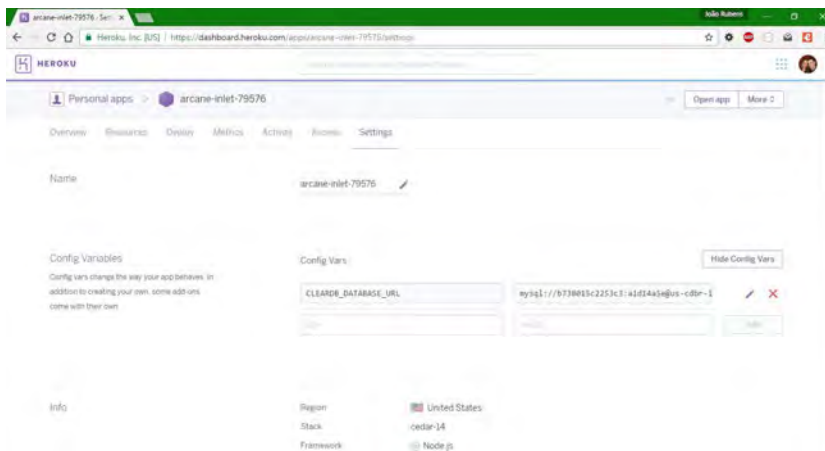


Figura 16.8: Configurações de acesso ao MySQL

No meu caso, as minhas configurações são:

```
mysql://b738015c2253c3:a1d14a5e@us-cdbr-iron-east-03.cleardb.net/heroku_b33068952d57b15?reconnect=true
```

Com essas configurações, você pode acessar um serviço de acesso ao MySQL remoto de sua escolha (Heidi, Workbench ou outro editor qualquer), e criar as tabelas da nossa aplicação. A string apresentada nesta tela representa os dados da conexão da seguinte forma:

Nome do usuário: b738015c2253c3
Senha: a1d14a5e
Host: us-cdbr-iron-east-03.cleardb.net
Banco de dados: heroku_b33068952d57b15

Depois de criadas as tabelas, é preciso alterar as configurações do arquivo `db.js` de nossa aplicação, com as informações do banco de dados criados no heroku. Para isso, vamos abrir o arquivo e inserir as novas configurações.

```
var knex = require('knex');
```



```
var db = knex({
  client: 'mysql',
  connection: {
    host : 'us-cdbr-iron-east-03.cleardb.net',
    user : 'b738015c2253c3',
    database : 'heroku_b33068952d57b15',
    password: 'a1d14a5e'
  }
});

module.exports = db;
```

16.3 INSTALANDO A APLICAÇÃO

A partir do terminal, vamos acessar o diretório da aplicação. Em seguida, precisamos efetuar *login* com a ferramenta do heroku instalada. Para isso, digite o comando `heroku login` e insira suas credenciais.

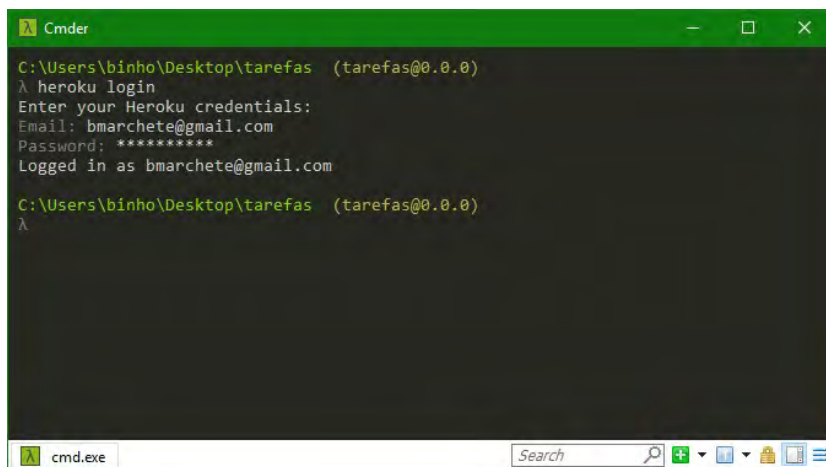
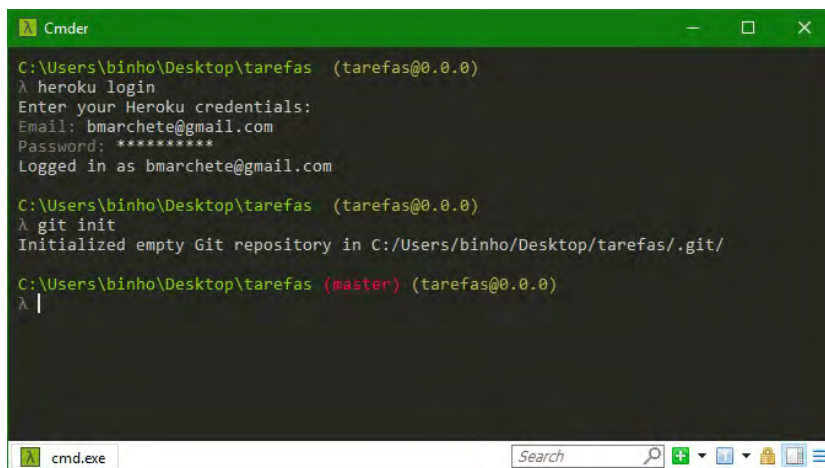


Figura 16.9: Acessando o serviço heroku pelo terminal

Feito isso, precisamos iniciar um novo repositório *git*. A partir

do terminal, fazemos isso com o comando `git init`. O heroku utiliza as configurações do *git* para receber os arquivos de nossa aplicação, a partir de um repositório remoto no *GitHub*.



```
C:\Users\binho\Desktop\tarefas (tarefas@0.0.0)
λ heroku login
Enter your Heroku credentials:
Email: bmarchete@gmail.com
Password: *****
Logged in as bmarchete@gmail.com

C:\Users\binho\Desktop\tarefas (tarefas@0.0.0)
λ git init
Initialized empty Git repository in C:/Users/binho/Desktop/tarefas/.git/

C:\Users\binho\Desktop\tarefas (master) (tarefas@0.0.0)
λ |
```

Figura 16.10: Acessando o serviço heroku pelo terminal

Quando fizermos o *upload* da nossa aplicação, queremos que o heroku faça a instalação automaticamente de todos os pacotes que listamos em nosso `packages.json`. Por isso, neste momento, precisamos excluir a pasta `node_modules`. Isso garante que não faremos o *upload* desnecessário desta pasta, já que o heroku faz a instalação destes módulos no momento de iniciar a aplicação.

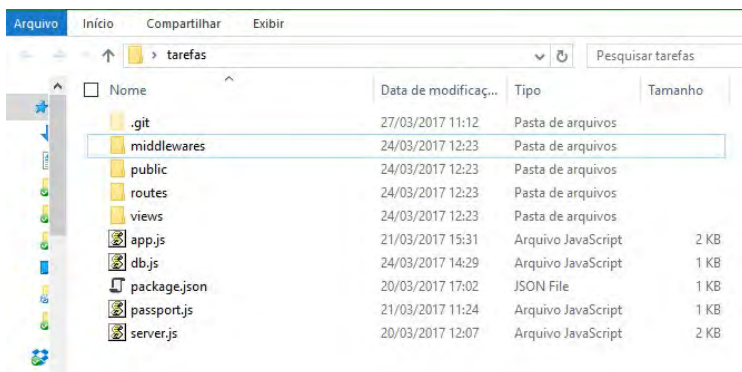
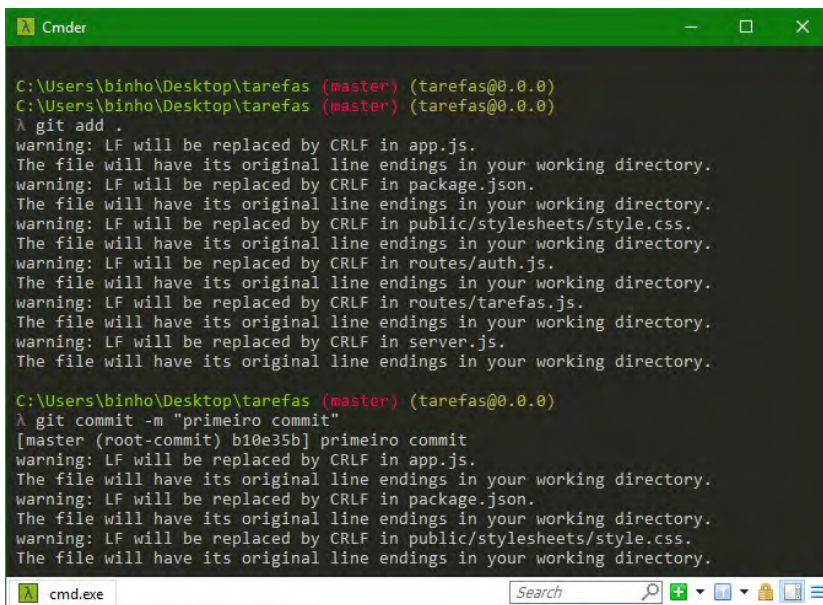


Figura 16.11: Exclusão da pasta node_modules

Depois de realizada a exclusão, vamos usar um comando do *git* que nos permite indicar que todos os arquivos do projeto serão carregados no nosso serviço de hospedagem. Executamos o comando `git add .`

Depois, realizamos um *commit* que cria um ponto de versionamento do nosso repositório. Esse comando garante que todos os arquivos versionados serão carregados no serviço de hospedagem. Assim, digitamos o comando `git commit -m "primeiro commit"`.

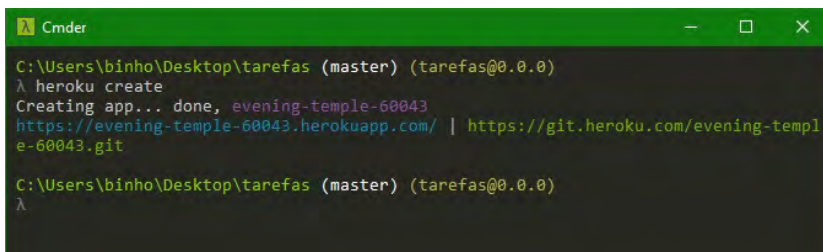


```
C:\Users\binho\Desktop\tarefas (master) (tarefas@0.0.0)
C:\Users\binho\Desktop\tarefas (master) (tarefas@0.0.0)
λ git add .
warning: LF will be replaced by CRLF in app.js.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in package.json.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in public/stylesheets/style.css.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in routes/auth.js.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in routes/tarefas.js.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in server.js.
The file will have its original line endings in your working directory.

C:\Users\binho\Desktop\tarefas (master) (tarefas@0.0.0)
λ git commit -m "primeiro commit"
[master (root-commit) b10e35b] primeiro commit
warning: LF will be replaced by CRLF in app.js.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in package.json.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in public/stylesheets/style.css.
The file will have its original line endings in your working directory.
```

Figura 16.12: Comandos git add e git commit

Agora nos resta levar os arquivos para o serviço do heroku. Como já realizamos o login, vamos criar um novo repositório remoto no heroku, que será o local acessado pelo heroku para instalar nossa aplicação. Para isso, inserimos o comando `heroku create`.

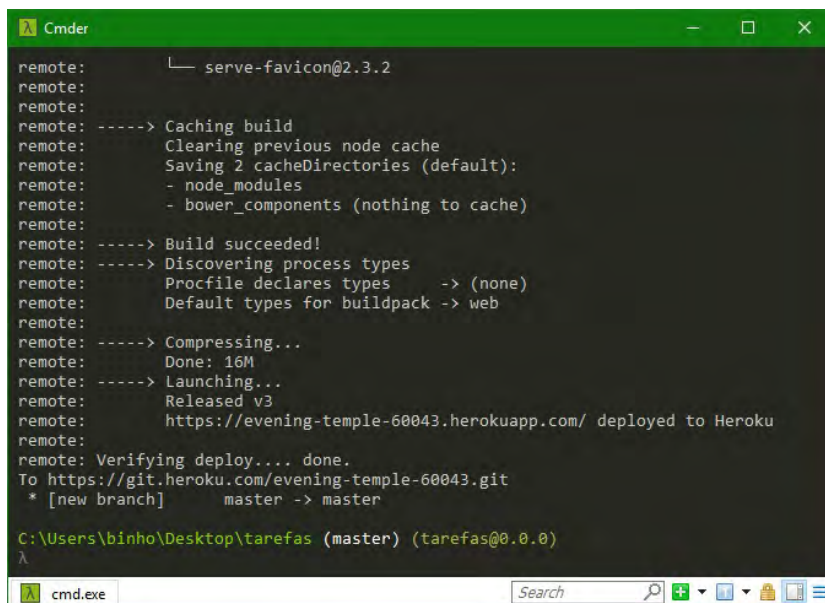


```
C:\Users\binho\Desktop\tarefas (master) (tarefas@0.0.0)
λ heroku create
Creating app... done, evening-temple-60043
https://evening-temple-60043.herokuapp.com/ | https://git.heroku.com/evening-templ
e-60043.git

C:\Users\binho\Desktop\tarefas (master) (tarefas@0.0.0)
λ
```

Figura 16.13: Criação do repositório remoto do heroku

Para concluirmos o *deploy*, basta inserirmos o comando `git push heroku master`, que os arquivos serão carregados no servidor. Automaticamente, o heroku executa um *script* de configuração, que instalará todos os módulos da aplicação e gerará uma URL válida para acessarmos. No meu caso, a URL gerada é a seguinte: `https://evening-temple-60043.herokuapp.com`.



```
remote:      └─ serve-favicon@2.3.2
remote:
remote:
remote: -----> Caching build
remote:      Clearing previous node cache
remote:      Saving 2 cacheDirectories (default):
remote:      - node_modules
remote:      - bower_components (nothing to cache)
remote:
remote: -----> Build succeeded!
remote: -----> Discovering process types
remote:      Procfile declares types     -> (none)
remote:      Default types for buildpack -> web
remote:
remote: -----> Compressing...
remote:      Done: 16M
remote: -----> Launching...
remote:      Released v3
remote:      https://evening-temple-60043.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/evening-temple-60043.git
 * [new branch]      master -> master

C:\Users\binho\Desktop\tarefas (master) (tarefas@0.0.0)
λ
```

Figura 16.14: Deploy da aplicação no heroku

Para testar se o serviço funciona corretamente, você pode acessar a sua URL e utilizar a aplicação normalmente.

16.4 CONSIDERAÇÕES

A escolha de um serviço de hospedagem depende muito das necessidades do desenvolvedor e do usuário da aplicação. Neste

capítulo, vimos de uma maneira simplificada como utilizar um serviço de hospedagem gratuito para aplicações Node.js.

É importante conhecer melhor todos os recursos oferecidos pelo serviço de hospedagem escolhida, pois essa escolha vai influenciar diretamente na performance da aplicação, já que as configurações realizadas no servidor web estão relacionadas diretamente com a aplicação hospedada.

DAQUI PARA A FRENTE...

Tenho convicção de que, após passar por todos os capítulos deste livro, você já possa se considerar um desenvolvedor Node.js, mesmo que seja em nível iniciante. Parabéns!

No decorrer dos capítulos, vimos como a plataforma progrediu nos últimos anos. Vimos também que é possível utilizar o Node.js para outras tarefas que não aplicações web, como a criação de sistemas de linha de comando.

Daqui para a frente, você tem um mundo de opções para estudar e conhecer ainda mais o Node.js. Vou deixar aqui algumas sugestões interessantes:

- **Busque novos módulos para criação de aplicações web além do Express.**

O Express não é o único do seu tipo. Existem diversos outros no mercado! Para começar, indico o Koa (<http://koajs.com/>) e o Hapi (<https://hapijs.com/>).

- **Entenda o padrão REST.**

Muitos desenvolvedores acabam conhecendo o Node.js após pesquisarem sobre REST APIs. O Node possui diversos frameworks que facilitam a criação de

aplicações REST de uma maneira muito rápida e eficiente. Esse é um tópico muito interessante para você se aprofundar, pois a criação de APIs é o assunto da vez, e o Node.js é escolha certa para essa tecnologia!

- **Conheça WebSockets.**

Com o Node.js, você manipula módulos desenvolvidos especificamente para lidar com WebSockets. Já pensou em fazer um chat web em que a sua página aguarda receber os dados de um servidor, sem ter de ficar fazendo requisições *ajax*?

- **Conheça IOTs.**

Você já deve ter ouvido falar em Internet das Coisas, ou IOTs! Com a plataforma Node, você pode desenvolver seus próprios protótipos! Visite o site <http://node-ardx.org/>, e conheça mais sobre a integração Node.js e Arduino.

- **Estude JavaScript.**

Para entender melhor como desenvolver aplicações com Node.js, é preciso conhecer a fundo JavaScript. Essa linguagem evolui constantemente, e ficar antenado na sua evolução garantirá sucesso a você durante sua jornada como desenvolvedor Node.js.

Obrigado por acompanhar o livro. Espero que você possa ter tirado proveito das lições aqui apresentadas, assim como eu aprendi muito durante a escrita dele. Nos encontramos no fórum

da editora Casa do Código, em
<http://forum.casadocodigo.com.br/>.