

Node.js

Aplicações web real-time com Node.js



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

AGRADECIMENTOS

Primeiramente, quero agradecer a Deus por tudo que fizeste em minha vida! Agradeço também ao meu pai e à minha mãe pelo amor, força, incentivo e por todo apoio desde o meu início de vida. Obrigado por tudo e principalmente por estar ao meu lado em todos os momentos.

Agradeço à sra. Charlotte Bento de Carvalho, pelo apoio e incentivo nos meus estudos desde a escola até a minha formatura na faculdade.

Um agradecimento ao meu primo Cláudio Souza. Foi graças a ele que entrei nesse mundo da tecnologia. Ele foi a primeira pessoa a me apresentar o computador, e me aconselhou anos depois a entrar em uma faculdade de TI.

Um agradecimento ao Bruno Alvares da Costa, Leandro Alvares da Costa e Leonardo Pinto, esses caras me apresentaram um mundo novo da área de desenvolvimento de *software*. Foram eles que me influenciaram a escrever um blog, a palestrar em eventos, a participar de comunidades e fóruns, e principalmente a nunca cair na zona de conforto, a aprender sempre. Foi uma honra trabalhar junto com eles em 2011. E hoje, mesmo muita coisa tendo mudado, ainda tenho a honra de trabalhar com o Leandro em uma nova startup que já está virando uma empresa, a BankFacil.

Obrigado ao pessoal da editora Casa do Código, em especial ao Paulo Silveira e Adriano Almeida. Muito obrigado pelo suporte e pela oportunidade!

Obrigado à galera da comunidade NodeBR. Seus *feedbacks* ajudaram a melhorar este livro e também agradeço a todos os leitores do blog Underground WebDev (<http://udgwebdev.com>) e também um agradecimento especial para a leitora Amanda Pedroso que contribuiu enviando um tutorial sobre como configurar Node.js, MongoDB e Redis na plataforma Windows.

Por último, obrigado a você, prezado leitor, por adquirir este livro. Espero que ele seja uma ótima referência para você.

COMENTÁRIOS

Veja a seguir alguns comentários no blog *Underground WebDev* a respeito do conteúdo que você esta prestes a ler.

"Parabéns pelo Post! Adorei, muito explicativo. A comunidade brasileira agradece." — Rafael Henrique Moreira - virtualjoker@gmail.com - <http://nodebr.com>

"Tive o prazer de trocar experiências e aprender muito com o Caio. Um cara singular à “instância”, do típico nerd que abraça um problema e não desgruda até resolvê-lo. Obrigado pela ajuda durante nosso tempo trabalho e não vou deixar de acompanhar essas aulas. Parabéns!" — Magno Ozzyr - magno_ozzyr@hotmail.com

"Digno de reconhecimento o empenho do Caio no projeto de contribuir com o desenvolvimento e propagação dessa tecnologia. Isso combina com o estilo ambicioso e persistente que sempre demonstrou no processo de formação. Sucesso! Continue compartilhando os frutos do seu trabalho para assim deixar sua marca na história da computação." — Fernando Macedo - fernando@fmacedo.com.br - <http://fmacedo.com.br>

"Ótimo conteúdo, fruto de muito trabalho e dedicação. Conheci o Caio ainda na faculdade, sempre enérgico, às vezes impaciente por causa de sua ânsia pelo novo. Continue assim buscando aprender mais e compartilhando o que você conhece com os outros. Parabéns pelo trabalho!" — Thiago Ferauche - thiago.ferauche@gmail.com

"Wow, muito bacana Caio! Eu mesmo estou ensaiando para

aprender Javascript e cia. Hoje trabalho mais com HTML/CSS, e essa ideia de "para Leigos" me interessa muito! Fico no aguardo dos próximos posts!! =)" — Marcio Toledo - mntoledo@gmail.com - <http://marciotoledo.com>

"Caião, parabéns pela iniciativa, pelo trabalho e pela contribuição para a comunidade. Trabalhamos juntos e sei que você é uma pessoa extremamente dedicada e ansioso por novos conhecimentos. Continue assim e sucesso!" — Leonardo Pinto - leonardo.pinto@gmail.com

"Caio, parabéns pelo curso e pelo conteúdo. É sempre bom contar com material de qualidade produzido no Brasil, pois precisamos difundir o uso de novas tecnologias e encorajar seu uso." — Evaldo Junior - evaldojuniorbento@gmail.com - <http://evaldojunior.com.br>

"Parabéns pela iniciativa! Acredito que no futuro você e outros façam mais cursos do mesmo, sempre buscando compartilhar o conhecimento pra quem quer aprender." — Jadson Lourenço - <http://twitter.com/jadsonlourenco>

SOBRE O AUTOR



Figura 1: Caio Ribeiro Pereira

Um Web Developer com forte experiência no domínio dessa sopa de letrinhas: Node.js, JavaScript, Meteor, Ruby On Rails, Agile, Filosofia Lean, Scrum, XP, Kanban e TDD.

Bacharel em Sistemas de Informação pela Universidade Católica de Santos, blogueiro nos tempos livres, apaixonado por programação, por compartilhar conhecimento, testar novas tecnologias, e assistir filmes e seriados.

Participo das comunidades:

- **NodeBR:** comunidade brasileira de Node.js;
- **MeteorBrasil:** comunidade brasileira de Meteor;
- **DevInSantos:** grupo de desenvolvedores de software em Santos.

Blog: <http://udgwebdev.com>.

PREFÁCIO

As mudanças do mundo web

Tudo na web se trata de consumismo e produção de conteúdo. Ler ou escrever blogs, assistir ou enviar vídeos, ver ou publicar fotos, ouvir músicas, e assim por diante. Isso fazemos naturalmente todos os dias na internet. E cada vez mais aumenta a necessidade dessa interação entre os usuários com os diversos serviços da web.

De fato, o mundo inteiro quer interagir mais e mais na internet, seja por meio de conversas com amigos em chats, jogando games online, atualizando constantemente suas redes sociais, ou participando de sistemas colaborativos. Esses tipos de aplicações requerem um poder de processamento extremamente veloz, para que seja eficaz a interação em tempo real entre cliente e servidor. E mais, isto precisa acontecer em uma escala massiva, suportando de centenas a milhões de usuários.

Então o que nós, desenvolvedores, precisamos fazer? Precisamos criar uma comunicação em tempo real entre cliente e servidor – que seja rápida, atenda muitos usuários ao mesmo tempo e utilize recursos de I/O (dispositivos de entrada ou saída) de forma eficiente. Qualquer pessoa com experiência em desenvolvimento web sabe que o HTTP não foi projetado para suportar estes requisitos. E pior, infelizmente existem sistemas que os adotam de forma inefficiente e incorreta, implementando soluções *workaround* ("gambiarras") que executam constantemente requisições assíncronas no servidor, mais

conhecidas como *long-polling*.

Para sistemas trabalharem em tempo real, servidores precisam enviar e receber dados utilizando comunicação bidirecional, em vez de utilizar intensamente requisição e resposta do modelo HTTP através do *Ajax*. E também temos que manter esse tipo comunicação de forma leve e rápida para continuar escalável, reutilizável e de desenvolvimento fácil de ser mantido a longo prazo.

A quem se destina este livro?

Esse livro é destinado aos desenvolvedores web, que tenham pelo menos conhecimentos básicos de JavaScript e arquitetura cliente-servidor. Ter domínio desses conceitos, mesmo que seja um conhecimento básico deles, será essencial para que a leitura deste livro seja de fácil entendimento.

Como devo estudar?

Ao decorrer da leitura, serão apresentados diversos conceitos e códigos, para que você aprenda na prática toda a parte teórica do livro. A partir do capítulo *Iniciando com o Express* até o capítulo final, vamos desenvolver na prática um projeto web, utilizando os principais frameworks e aplicando as boas práticas de desenvolvimento JavaScript para Node.js.

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>

Sumário

1 Bem-vindo ao mundo Node.js	1
1.1 O problema das arquiteturas bloqueantes	1
1.2 E assim nasceu o Node.js	2
1.3 Single-thread	3
1.4 Event-Loop	4
1.5 Instalação e configuração	5
1.6 Gerenciando módulos com NPM	8
1.7 Entendendo o package.json	10
1.8 Escopos de variáveis globais	12
1.9 CommonJS, como ele funciona?	13
2 Desenvolvendo aplicações web	15
2.1 Criando nossa primeira aplicação web	15
2.2 Como funciona um servidor http?	17
2.3 Trabalhando com diversas rotas	18
2.4 Separando o HTML do JavaScript	21
2.5 Desafio: implementando um roteador de URL	23
3 Por que o assíncrono?	25

Sumário		Casa do Código
3.1 Desenvolvendo de forma assíncrona	25	
3.2 Assincronismo versus sincronismo	29	
3.3 Entendendo o Event-Loop	31	
3.4 Evitando callbacks hell	34	
4 Iniciando com o Express	37	
4.1 Por que utilizá-lo?	37	
4.2 Instalação e configuração	38	
4.3 Criando um projeto de verdade	39	
4.4 Gerando scaffold do projeto	41	
4.5 Organizando os diretórios do projeto	47	
5 Dominando o Express	54	
5.1 Estruturando views	54	
5.2 Controlando as sessões de usuários	55	
5.3 Criando rotas no padrão REST	61	
5.4 Aplicando filtros antes de acessar as rotas	68	
5.5 Indo além: criando páginas de erros amigáveis	71	
6 Programando sistemas real-time	76	
6.1 Como funciona uma conexão bidirecional?	76	
6.2 Conhecendo o framework Socket.IO	77	
6.3 Implementando um chat real-time	78	
6.4 Organizando o carregamento de Sockets	85	
6.5 Socket.IO e Express usando a mesma sessão	86	
6.6 Gerenciando salas do chat	91	
6.7 Notificadores na agenda de contatos	96	
6.8 Principais eventos do Socket.IO	101	

7 Integração com banco de dados	104
7.1 Bancos de dados mais adaptados para Node.js	104
7.2 Instalando o MongoDB	106
7.3 MongoDB no Node.js utilizando Mongoose	108
7.4 Modelando com Mongoose	110
7.5 Implementando um CRUD na agenda de contatos	112
7.6 Persistindo estruturas de dados usando Redis	118
7.7 Mantendo um histórico de conversas do chat	119
7.8 Persistindo lista de usuários online	122
8 Preparando um ambiente de testes	125
8.1 Mocha, o framework de testes para Node.js	125
8.2 Criando um Environment para testes	126
8.3 Instalando e configurando o Mocha	128
8.4 Rodando o Mocha no ambiente de testes	130
8.5 Testando as rotas	131
8.6 Deixando seus testes mais limpos	140
9 Aplicação Node em produção – Parte 1	142
9.1 O que vamos fazer?	142
9.2 Configurando clusters	142
9.3 Redis controlando as sessões da aplicação	146
9.4 Monitorando aplicação através de logs	148
9.5 Otimizações no Express	150
10 Aplicação Node em produção – Parte 2	153
10.1 Mantendo a aplicação protegida	153
10.2 Externalizando variáveis de configurações	158

10.3 Aplicando Singleton nas conexões do Mongoose	161
10.4 Mantendo o sistema no ar com Forever	161
11 Node.js e Nginx	166
11.1 Servindo arquivos estáticos do Node.js usando o Nginx	166
12 Continuando os estudos	171
13 Bibliografia	174

Versão: 20.9.6

CAPÍTULO 1

BEM-VINDO AO MUNDO NODE.JS

1.1 O PROBLEMA DAS ARQUITETURAS BLOQUEANTES

Os sistemas para web desenvolvidos sobre plataforma **.NET**, **Java**, **PHP**, **Ruby** ou **Python** nativamente possuem uma característica em comum: eles paralisam um processamento enquanto utilizam um I/O no servidor. Essa paralisação é conhecida como modelo bloqueante (*Blocking-Thread*).

Em um servidor web, podemos visualizá-lo de forma ampla e funcional. Vamos considerar que cada processo é uma requisição feita pelo usuário. Com o decorrer da aplicação, novos usuários vão acessando-a, gerando uma requisição no servidor. Um sistema bloqueante enfileira as requisições e depois as processa, uma a uma, não permitindo múltiplos processamentos. Enquanto uma requisição é processada, as demais ficam em espera, mantendo por um período de tempo uma fila de requisições ociosas.

Esta é uma arquitetura clássica, existente em diversos sistemas, e que possui um design inefficiente. É gasto grande parte do tempo mantendo uma fila ociosa enquanto é executado um I/O. Tarefas

como enviar e-mail, consultar o banco de dados, e leitura em disco são exemplos de tarefas que gastam uma grande parcela desse tempo, bloqueando o sistema inteiro enquanto não são finalizadas.

Com o aumento de acessos no sistema, a frequência de gargalos será maior, aumentando a necessidade de fazer um *upgrade* nos *hardwares* dos servidores. Mas como *upgrade* das máquinas é algo muito custoso, o ideal seria buscar novas tecnologias que façam bom uso do *hardware* existente, e que utilizem ao máximo o poder do processador atual, não o mantendo ocioso quando realizar tarefas do tipo bloqueante.

1.2 E ASSIM NASCEU O NODE.JS



Figura 1.1: Logotipo do Node.js

Foi baseado neste problema que, no final de 2009, Ryan Dahl, com a ajuda inicial de 14 colaboradores, criou o Node.js. Esta tecnologia possui um modelo inovador: sua arquitetura é totalmente *non-blocking thread* (não bloqueante), se sua aplicação trabalha com processamento de arquivos e ou realiza muito I/O adotar esse tipo de arquitetura vai resultar em uma boa performance com relação ao consumo de memória e usa ao máximo e de forma eficiente o poder de processamento dos servidores, principalmente em sistemas que produzem uma alta

carga de processamento.

Sistemas que utilizam Node.js não sofrem de *dead-locks* no sistema, porque o Node.js trabalha apenas em *single-thread* (única thread por processo), e desenvolver sistemas nesse paradigma é simples e prático.

Esta é uma plataforma altamente escalável e de baixo nível, pois você vai programar diretamente com diversos protocolos de rede e internet, ou utilizar bibliotecas que acessam recursos do sistema operacional, principalmente recursos de sistemas baseado em Unix. O JavaScript é a sua linguagem de programação, e isso foi possível graças à **engine JavaScript V8**, a mesma utilizada no navegador *Google Chrome*.

1.3 SINGLE-THREAD

Suas aplicações serão *single-thread*, ou seja, cada aplicação terá instância de um único processo. Se você está acostumado a trabalhar com programação concorrente em plataforma *multi-thread*, infelizmente não será possível com Node, mas saiba que existem outras maneiras de se criar um sistema concorrente. Por exemplo, podem-se utilizar *clusters* (assunto a ser explicado na seção *Configurando clusters*, no capítulo *Aplicação Node em produção — Parte 1*), que são um módulo nativo do Node.js e super fácil de implementar.

Outra maneira é usar ao máximo a programação assíncrona. Esse será o assunto mais abordado durante o decorrer deste livro, pelo qual explicarei diversos cenários e exemplos práticos nos quais são executadas, em paralelo, funções em *background*, que

aguardam o seu retorno através de funções de *callback*. E tudo isso é trabalhado forma não-bloqueante.

1.4 EVENT-LOOP

Node.js é orientado a eventos. Ele segue a mesma filosofia de orientação de eventos do JavaScript client-side; a única diferença é que não existem eventos de `click` do mouse, `keyup` do teclado, ou qualquer evento de componentes HTML. Na verdade, trabalhamos com eventos de I/O do servidor, como por exemplo: o evento `connect` de um banco de dados, um `open` de um arquivo, um `data` de um *streaming* de dados e muitos outros.

O *Event-Loop* é o agente responsável por escutar e emitir eventos no sistema. Na prática, ele é um loop infinito que, a cada iteração, verifica em sua fila de eventos se um determinado evento foi emitido. Quando ocorre, é emitido um evento. Ele o executa e envia para fila de executados. Quando um evento está em execução, nós podemos programar qualquer lógica dentro dele. Isso tudo acontece graças ao mecanismo de função *callback* do JavaScript.

O design *event-driven* do Node.js foi inspirado pelos frameworks Event Machine do Ruby (<http://rubyeventmachine.com>) e Twisted do Python (<http://twistedmatrix.com>). Porém, o *Event-loop* do Node é mais performático por que seu mecanismo é nativamente executado de forma não-bloqueante. Isso faz dele um grande diferencial em relação aos seus concorrentes que realizam chamadas bloqueantes para iniciar os seus respectivos *Event-loops*.

1.5 INSTALAÇÃO E CONFIGURAÇÃO

Para configurar o ambiente Node.js, independente de qual sistema operacional você utilizar, as dicas serão as mesmas. É claro que os procedimentos serão diferentes para cada sistema (principalmente para o Windows, mas não será nada grave).

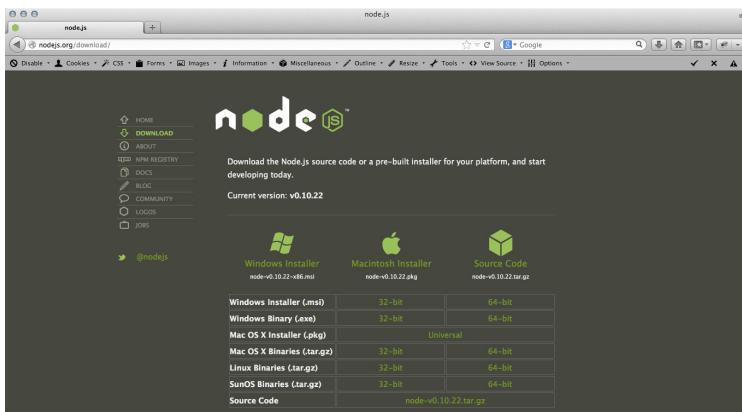


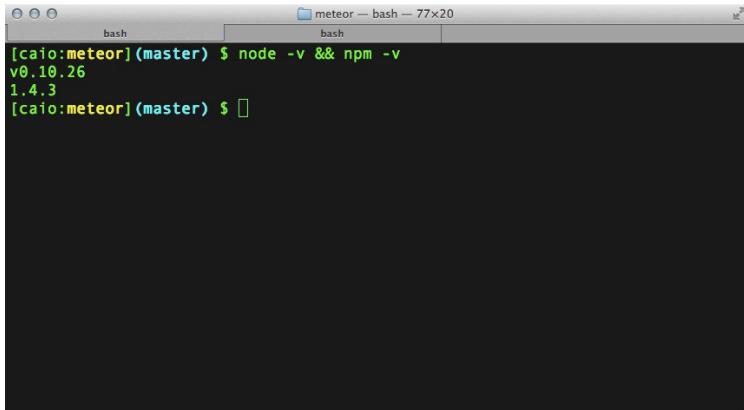
Figura 1.2: Página de download do Node.js

Instalando Node.js

O primeiro passo é acessar o site oficial (<http://nodejs.org>) e clicar em **Download**. Para usuários do *Windows* e *MacOSX*, basta baixar os seus instaladores e executá-los normalmente. Já para quem já usa *Linux* com *Package Manager* instalado, acesse esse <https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager>, que é referente às instruções sobre como instalá-lo em diferentes sistemas.

Instale o Node.js de acordo com seu sistema e, caso não ocorra problemas, basta abrir o seu terminal console ou prompt de

comando, e digitar o comando: `node -v && npm -v` para ver as respectivas versões do Node.js e NPM (*Node Package Manager*) que foram instaladas.



```
[caio:meteor] (master) $ node -v && npm -v
v0.10.26
1.4.3
[caio:meteor] (master) $ 
```

Figura 1.3: Versão do Node.js e NPM utilizada neste livro

A última versão estável usada neste livro é **Node 0.10.26**, junto do **NPM 1.4.3**. Aliás, é altamente recomendável utilizar essa versão ou superior, pois recentemente foi identificada uma vulnerabilidade de ataque *DoS* em versões anteriores.

Veja mais detalhes no blog oficial, em <http://blog.nodejs.org/2013/10/22/cve-2013-4450-http-server-pipeline-flood-dos>.

DICA

Todo o conteúdo deste livro será compatível com versões do Node.js **0.8.0** ou superiores.

Configurando o ambiente de desenvolvimento

Para configurá-lo, basta adicionar uma variável de ambiente **NODE_ENV** no sistema operacional. Em sistemas Linux ou OSX, basta acessar com um editor de texto qualquer e em modo super user (**sudo**) o arquivo **.bash_profile** ou **.bashrc**, e adicionar o seguinte comando: `export NODE_ENV='development'`. No Windows 7, o processo é um pouco diferente.

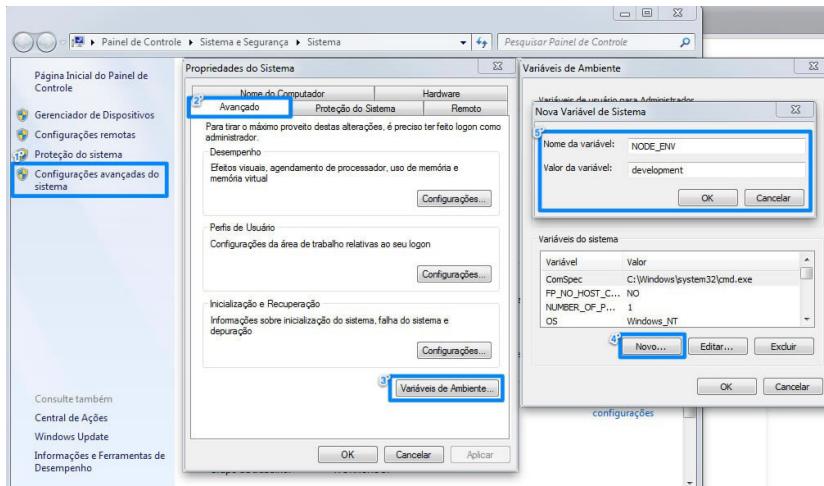


Figura 1.4: Configurando a variável NODE_ENV no Windows 7

Clique com botão direito no ícone **Meu Computador** e selecione a opção **Propriedades**. No lado esquerdo da janela, clique no link **Configurações avançadas do sistema**. Na janela seguinte, acesse a aba **Avançado** e clique no botão **Variáveis de Ambiente....** Agora, no campo **Variáveis do sistema**, clique no botão **Novo....** Em nome da variável, digite **NODE_ENV** e em valor da variável, digite **development**.

Após finalizar essa tarefa, reinicie seu computador para carregar essa variável no sistema operacional.

Rodando o Node

Para testarmos o ambiente, executaremos o nosso primeiro programa *Hello World*. Execute o comando `node` para acessarmos o REPL (*Read-Eval-Print-Loop*), que permite executar código JavaScript diretamente no terminal. Digite `console.log("Hello World");` e tecle `ENTER` para executá-lo na hora.

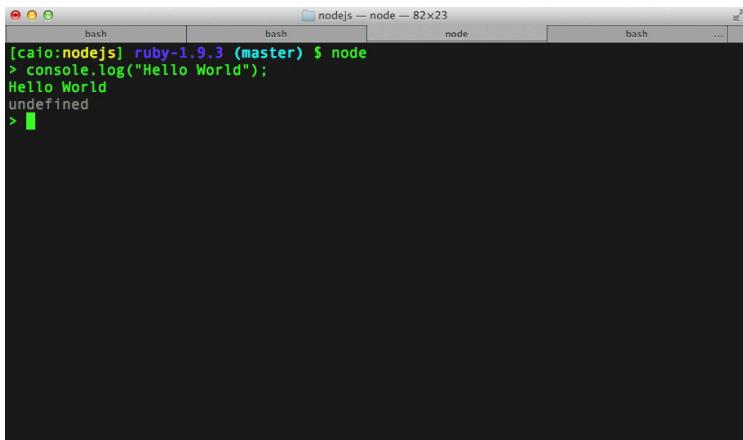
A screenshot of a terminal window titled "nodejs — node — 82x23". The window contains three tabs: "bash", "bash", and "node". The "node" tab is active and shows the following command and output:
[caio:nodejs] ruby-1.9.3 (master) \$ node
> console.log("Hello World");
Hello World
undefined
> █

Figura 1.5: Hello World via REPL do Node.js

1.6 GERENCIANDO MÓDULOS COM NPM

Assim como o *Gems* do *Ruby*, ou o *Maven* do *Java*, o `Node.js` também possui o seu próprio gerenciador de pacotes: ele se chama **NPM** (*Node Package Manager*). Ele se tornou tão popular pela comunidade, que foi a partir da versão **0.6.0** do `Node.js` que ele se

integrado ao instalador, tornando-se o gerenciador *default*. Isso simplificou a vida dos desenvolvedores na época, pois fez com que diversos projetos se convergirem para essa plataforma.

Não listarei todos, mas apenas os comandos principais para que você tenha noções de como gerenciar módulos nele:

- `npm install nome_do_módulo` : instala um módulo no projeto;
- `npm install -g nome_do_módulo` : instala um módulo global;
- `npm install nome_do_módulo --save` : instala o módulo no projeto, atualizando o `package.json` na lista de dependências;
- `npm list` : lista todos os módulos do projeto;
- `npm list -g` : lista todos os módulos globais;
- `npm remove nome_do_módulo` : desinstala um módulo do projeto;
- `npm remove -g nome_do_módulo` : desinstala um módulo global;
- `npm update nome_do_módulo` : atualiza a versão do módulo;
- `npm update -g nome_do_módulo` : atualiza a versão do módulo global;
- `npm -v` : exibe a versão atual do NPM;
- `npm adduser nome_do_usuário` : cria uma conta no NPM, através do site <https://npmjs.org>.
- `npm whoami` : exibe detalhes do seu perfil público NPM (é necessário criar uma conta antes);
- `npm publish` : publica um módulo no site do NPM (é necessário ter uma conta antes).

1.7 ENTENDENDO O PACKAGE.JSON

Todo projeto Node.js é chamado de módulo. Mas, o que é um módulo?

No decorrer da leitura, perceba que falarei muito sobre o termo módulo, biblioteca e framework, e, na prática, eles possuem o mesmo significado. O termo módulo surgiu do conceito de que a arquitetura do Node.js é modular. E todo módulo é acompanhado de um arquivo descritor, conhecido pelo nome de `package.json`.

Este arquivo é essencial para um projeto Node.js. Um `package.json` mal escrito pode causar bugs ou impedir o funcionamento correto do seu módulo, pois ele possui alguns atributos chaves que são compreendidos pelo Node.js e NPM.

No código a seguir, apresentarei um `package.json` que contém os principais atributos para descrever um módulo:

```
{  
  "name": "meu-primeiro-node-app",  
  "description": "Meu primeiro app em Node.js",  
  "author": "Caio R. Pereira <caio@email.com>",  
  "version": "1.2.3",  
  "private": true,  
  "dependencies": {  
    "modulo-1": "1.0.0",  
    "modulo-2": "~1.0.0",  
    "modulo-3": ">=1.0.0"  
  },  
  "devDependencies": {  
    "modulo-4": "*"  
  }  
}
```

Com esses atributos, você já descreve o mínimo possível o que será sua aplicação. O atributo `name` é o principal. Com ele, você

descreve o nome do projeto, nome pelo qual seu módulo será chamado via função `require('meu-primeiro-node-app')`.

Em `description`, descrevemos o que será este módulo. Ele deve ser escrito de forma curta e clara, fornecendo um resumo do módulo. O `author` é um atributo para informar o nome e e-mail do autor. Utilize o formato `Nome <email>` para que sites como <https://npmjs.org> reconheça corretamente esses dados. Outro atributo principal é o `version`, com o qual definimos a versão atual do módulo. É extremamente recomendado que tenha esse atributo, se não será impossível instalar o módulo via comando `npm`. O atributo `private` é um booleano, e determina se o projeto terá código aberto ou privado para download no <https://npmjs.org>.

Os módulos no Node.js trabalham com **3 níveis de versionamento**. Por exemplo, a versão `1.2.3` está dividida nos níveis:

1. *Major*;
2. *Minor*;
3. *Patch*.

Repare que no campo `dependencies` foram incluídos 4 módulos, sendo que cada um utilizou uma forma diferente de definir a versão que será adicionada no projeto. O primeiro, o `modulo-1`, somente será incluído sua versão fixa, a `1.0.0`. Utilize este tipo versão para instalar dependências cuja atualizações possam quebrar o projeto pelo simples fato de que certas funcionalidades foram removidas e ainda as utilizamos na aplicação.

O segundo módulo já possui uma certa flexibilidade de update. Ele utiliza o caractere ~ , que faz atualizações a nível de *patch* (1.0.x) . Geralmente, essas atualizações são seguras, trazendo apenas melhorias ou correções de bugs. O modulo-3 atualiza versões que sejam maior ou igual a 1.0.0 em todos os níveis de versão. Em muitos casos, usar ">=" pode ser perigoso, porque a dependência pode ser atualizada a nível *major* ou *minor*, contendo grandes modificações que podem quebrar um sistema em produção, comprometendo seu funcionamento e exigindo que você atualize todo código até voltar ao normal.

O último, o modulo-4 , utiliza o caractere * ; este sempre pegará a última versão do módulo em qualquer nível. Ele também pode causar problemas nas atualizações e tem o mesmo comportamento do versionamento do modulo-3 . Geralmente, ele é usado em devDependencies , que são dependências focadas para testes automatizados, e as atualizações dos módulos não prejudicam o comportamento do sistema que já está no ar.

1.8 ESCOPOS DE VARIÁVEIS GLOBAIS

Assim como no browser, utilizamos o mesmo JavaScript no Node.js. Ele também usa **escopos locais e globais** de variáveis. A única diferença é na forma como são implementados esses escopos. No client-side, as variáveis globais são criadas da seguinte maneira:

```
window.hoje = new Date();
alert(window.hoje);
```

Em qualquer browser, a palavra-chave window permite criar variáveis globais que são acessadas em qualquer lugar. Já no

Node.js, usamos uma outra *keyword* para aplicar essa mesma técnica:

```
global.hoje = new Date();
console.log(global.hoje);
```

Ao utilizar `global`, mantemos uma variável global acessível em qualquer parte do projeto, sem a necessidade de chamá-la via `require` ou passá-la por parâmetro em uma função.

Esse conceito de variável global é existente na maioria das linguagens de programação, assim como sua utilização; portanto, é recomendado trabalhar com o mínimo possível de variáveis globais para evitar futuros gargalos de memória na aplicação.

1.9 COMMONJS, COMO ELE FUNCIONA?

O Node.js utiliza nativamente o padrão *CommonJS* para organização e carregamento de módulos. Na prática, diversas funções deste padrão serão usadas com frequência em um projeto Node.js. A função `require('nome-do-modulo')` é um exemplo disso, ela carrega um módulo. E para criar um código JavaScript que seja modular e carregável pelo `require`, utilizam-se as variáveis globais: `exports` ou `module.exports`.

A seguir, apresento-lhe dois exemplos de códigos que utilizam esse padrão do *CommonJS*. Primeiro, crie o código `hello.js`:

```
module.exports = function(msg) {
  console.log(msg);
};
```

Depois, crie o código `human.js` com o seguinte código:

```
exports.hello = function(msg) {
```

```
    console.log(msg);
};
```

A diferença entre o `hello.js` e o `human.js` está na maneira como eles serão carregados. Em `hello.js`, carregamos uma única função modular e, em `human.js`, é carregado um objeto com funções modulares. Essa é a grande diferença entre eles. Para entender melhor na prática, crie o código `app.js` para carregar esses módulos:

```
var hello = require('./hello');
var human = require('./human');

hello('Olá pessoal!');
human.hello('Olá galera!');
```

Tenha certeza de que os códigos `hello.js`, `human.js` e `app.js` estejam na mesma pasta e rode no console o comando: `node app.js`.

E então, o que aconteceu? O resultado foi praticamente o mesmo: o `app.js` carregou os módulos `hello.js` e `human.js` via `require()`, em seguida foi executada a função `hello()` que imprimiu a mensagem `Olá pessoal!` e, por último, o objeto `human`, que executou sua função `human.hello('Olá galera!')`.

Percebiam o quanto simples é programar com Node.js! Com base nesses pequenos trechos de código, já foi possível criar um código altamente escalável e modular que utiliza as boas práticas do padrão *CommonJS*.

CAPÍTULO 2

DESENVOLVENDO APLICAÇÕES WEB

2.1 CRIANDO NOSSA PRIMEIRA APLICAÇÃO WEB

Node.js é multiprotocolo, ou seja, com ele será possível trabalhar com os protocolos: **HTTP**, **HTTPS**, **FTP**, **SSH**, **DNS**, **TCP**, **UDP** e **WebSockets**. Também existem outros protocolos, que são disponíveis através de módulos não oficiais criados pela comunidade.

Um dos mais utilizados para desenvolver sistemas web é o protocolo **HTTP**. De fato, é o com a maior quantidade de módulos disponíveis para trabalhar no Node.js.

Na prática, desenvolveremos um sistema web utilizando o módulo nativo HTTP, mostrando suas vantagens e desvantagens. Também apresentarei soluções de módulos estruturados para desenvolver aplicações complexas de forma modular e escalável.

Toda aplicação web necessita de um servidor para disponibilizar todos os seus recursos. Na prática, com o Node.js você desenvolve uma *aplicação middleware*, ou seja, além de programar as funcionalidades da sua aplicação, você também

programa códigos de configuração de infraestrutura da sua aplicação.

Inicialmente, isso parece ser muito trabalhoso, pois o Node.js utiliza o mínimo de configurações para servir uma aplicação; entretanto, esse trabalho permite que você customize ao máximo o seu servidor. Uma vantagem disso é poder configurar em detalhes o sistema, permitindo desenvolver algo performático e controlado pelo programador.

Caso performance não seja prioridade no desenvolvimento do seu sistema, recomendo que use alguns módulos adicionais que já vêm com o mínimo necessário de configurações prontas para você não perder tempo trabalhando com isso.

Alguns módulos conhecidos são:

- **Connect** – <https://github.com/senchalabs/connect>;
- **Express** – <http://expressjs.com>;
- **Geddy** – <http://geddyjs.org>;
- **CompoundJS** – <http://compoundjs.com>;
- **Sails** – <http://balderdashy.github.io/sails>.

Esses módulos já são preparados para lidar com desde uma infraestrutura mínima até uma mais enxuta, permitindo trabalhar com arquiteturas *RESTFul*, padrão MVC (*Model-View-Controller*) e também com conexões *real-time* utilizando *WebSockets*.

Primeiro, usaremos apenas o módulo nativo **HTTP**, pois precisamos entender todo o seu conceito, visto que todos os frameworks citados o utilizam como estrutura inicial em seus projetos. A seguir, mostro a vocês uma clássica aplicação *Hello World*.

Crie o arquivo `hello_server.js` com o seguinte conteúdo:

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

var server = http.createServer(function(request, response){
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("<h1>Hello World!</h1>");
    response.end();
});
server.listen(3000);
```

Esse é um exemplo clássico e simples de um servidor Node.js. Ele está sendo executado na **porta 3000** e, por padrão, responde através da **rota raiz** `/` um resultado em formato `html`, com a mensagem `Hello World!`.

Vá para a linha de comando e rode `node hello_server.js`. Faça o teste acessando, no seu navegador, o endereço <http://localhost:3000>.

2.2 COMO FUNCIONA UM SERVIDOR HTTP?

Um servidor Node.js utiliza o mecanismo *Event loop*, sendo responsável por lidar com a emissão de eventos. Na prática, a função `http.createServer()` é responsável por levantar um servidor, e o seu callback `function(request, response)` apenas é executado quando o servidor recebe uma requisição. Para isso, o *Event loop* verifica constantemente se o servidor foi requisitado e, quando ele recebe uma requisição, ele emite um evento para que seja executado o seu callback.

O Node.js trabalha muito com chamadas assíncronas que respondem através de callbacks do JavaScript. Por exemplo, se quisermos notificar que o servidor está de pé, mudamos a linha

`server.listen` para receber em parâmetro uma função que faz esse aviso:

```
server.listen(3000, function(){
  console.log('Servidor Hello World rodando!');
});
```

O método `listen` também é assíncrono, e você só saberá que o servidor está de pé quando o Node invocar sua função de callback.

Se você ainda está começando com JavaScript, pode estranhar um pouco ficar passando como parâmetro uma `function` por todos os lados, mas isso é algo muito comum no mundo JavaScript. Como sintaxe alternativa, caso o seu código fique muito complicado em encadeamentos de diversos blocos, podemos isolá-lo em funções com nomes mais significativos, por exemplo:

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

var atendeRequisicao = function(request, response) {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("<h1>Hello World!</h1>");
  response.end();
}
var server = http.createServer(atendeRequisicao);

var servidorLigou = function() {
  console.log('Servidor Hello World rodando!');
}
server.listen(3000, servidorLigou);
```

2.3 TRABALHANDO COM DIVERSAS ROTAS

Até agora respondemos apenas o endereço `/`, mas queremos possibilitar que nosso servidor também responda a outros endereços. Utilizando um palavreado comum entre

desenvolvedores rails, queremos adicionar novas **rotas**.

Vamos adicionar duas novas rotas: uma rota `/bemvindo` para página de "Bem-vindo ao Node.js!" e uma rota genérica, que leva para uma página de erro. Faremos isso por meio de um simples encadeamento de condições, em um novo arquivo, o `hello_server3.js`:

```
var http = require('http');
var server = http.createServer(function(request, response){
  response.writeHead(200, {"Content-Type": "text/html"});
  if(request.url == "/"){
    response.write("<h1>Página principal</h1>");
  }else if(request.url == "/bemvindo"){
    response.write("<h1>Bem-vindo :)</h1>");
  }else{
    response.write("<h1>Página não encontrada :(</h1>");
  }
  response.end();
});
server.listen(3000, function(){
  console.log('Servidor rodando!');
});
```

Rode novamente e faça o teste acessando a URL `http://localhost:3000/bemvindo`. Acesse também uma outra, diferente desta. Viu o resultado?

Reparam na complexidade do nosso código: o roteamento foi tratado através dos comandos `if` e `else`, e a leitura de URL é obtida através da função `request.url()`, que retorna uma string sobre o que foi digitado na barra de endereço do browser.

Esses endereços usam padrões para capturar valores na URL. Esses padrões são: *query strings* (`?nome=joao`) e *path* (`/admin`). Em um projeto maior, tratar todas as URLs dessa maneira seria trabalhoso e confuso demais. No Node.js, existe o módulo nativo

chamado `url`, que é responsável por fazer *parser* e formatação de URLs. Acompanhe como capturamos valores de uma query string no exemplo a seguir. Aproveite e crie o novo arquivo `url_server.js`:

```
var http = require('http');
var url = require('url');

var server = http.createServer(function(request, response){
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write("<h1>Dados da query string</h1>");
  var result = url.parse(request.url, true);
  for(var key in result.query){
    response.write("<h2>" + key + " : " + result.query[key] + "</h2>");
  }
  response.end();
});
server.listen(3000, function(){
  console.log('Servidor http.');
});
```

Neste exemplo, a função `url.parse(request.url, true)` fez um parser da URL obtida pela requisição do cliente (`request.url`).

Esse módulo identifica por meio do retorno da função `url.parser()` os seguintes atributos:

- `href` – retorna a URL completa:
`http://user:pass@host.com:8080/p/a/t/h?query=string#hash`
- `protocol` – retorna o protocolo: `http`
- `host` – retorna o domínio com a porta:
`host.com:8080`
- `auth` – retorna dados de autenticação: `user:pass`
- `hostname` – retorna o domínio: `host.com`
- `port` – retorna a porta: `8080`

- pathname – retorna os pathnames da URL:
/p/a/t/h
- search – retorna uma query string: ?
query=string
- path – retorna a concatenação de pathname com
query string: /p/a/t/h?query=string
- query – retorna uma query string em JSON:
{'query': 'string'}
- hash – retorna ancora da URL: #hash

Resumindo, o módulo URL permite organizar todas as URLs da aplicação.

2.4 SEPARANDO O HTML DO JAVASCRIPT

Agora precisamos organizar os códigos HTML, e uma boa prática é separá-los do JavaScript, fazendo com que a aplicação renderize código HTML quando o usuário solicitar uma determinada rota. Para isso, usaremos outro módulo nativo **FS** (*File System*). Ele é responsável por manipular arquivos e diretórios do sistema operacional.

O mais interessante desse módulo é que ele possui diversas funções de manipulação tanto de forma assíncrona como de forma síncrona. Por padrão, as funções nomeadas com o final `Sync()` são para tratamento síncrono. No exemplo a seguir, apresento as duas maneiras de ler um arquivo utilizando *File System*:

```
var fs = require('fs');
fs.readFile('/index.html', function(erro, arquivo){
  if (erro) throw erro;
  console.log(arquivo);
});
```

```
var arquivo = fs.readFileSync('/index.html');
console.log(arquivo);
```

Diversos módulos do Node.js possuem funções com versões assíncronas e síncronas. O `fs.readFile()` faz uma leitura assíncrona do arquivo `index.html`. Depois que o arquivo foi carregado, é invocada uma função callback para fazer os tratamentos finais, seja de erro ou de retorno do arquivo. Já o `fs.readFileSync()` realizou uma leitura síncrona, bloqueando a aplicação até terminar sua leitura e retornar o arquivo.

LIMITAÇÕES DO FILE SYSTEM NOS SISTEMAS OPERACIONAIS

Um detalhe importante sobre o módulo *File System* é que ele não é 100% consistente entre os sistemas operacionais. Algumas funções são específicas para sistemas Linux, OS X e Unix, e outras são apenas para Windows.

Para melhores informações leia sua documentação, em <http://nodejs.org/api/fs.html>.

Voltando ao desenvolvimento da nossa aplicação, utilizaremos a função `fs.readFile()` para renderizar HTML de forma assíncrona. Crie um novo arquivo, chamado `site_pessoal.js`, com o seguinte código:

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function(request, response){
  // A constante __dirname retorna o diretório raiz da
  // aplicação.
  fs.readFile(__dirname + '/index.html', function(error, html){
```

```

        response.writeHeader(200, {'Content-Type': 'text/html'});
        response.write(html);
        response.end();
    });
});
server.listen(3000, function(){
    console.log('Executando Site Pessoal');
});

```

Para que isso funcione, você precisa do arquivo `index.html` dentro do mesmo diretório. Segue um exemplo de `hello` que pode ser usado:

```

<!DOCTYPE html>
<html>
    <head>
        <title>Olá este é o meu site pessoal!</title>
    </head>
    <body>
        <h1>Bem vindo ao meu site pessoal</h1>
    </body>
</html>

```

Rode o `node site_pessoal.js`, e acesse novamente `http://localhost:3000`.

2.5 DESAFIO: IMPLEMENTANDO UM ROTEADOR DE URL

Antes de finalizar esse capítulo, quero propor um desafio. Já que aprendemos a utilizar os **módulos HTTP, URL e FS** (File System), que tal reorganizar a nossa aplicação para renderizar um determinado arquivo HTML baseado no path da URL?

As regras do desafio são:

- Crie 3 arquivos HTML: `artigos.html`,

- ```
 contato.html e erro.html ;
```
- Coloque qualquer conteúdo para cada página HTML;
  - Ao digitar no browser o path, /artigos deve renderizar artigos.html ;
  - A regra anterior também se aplica para o arquivo contato.html ;
  - Ao digitar qualquer path diferente de /artigos e /contato , deve renderizar erro.html ;
  - A leitura dos arquivos HTML deve ser assíncrona;
  - A rota principal " / " deve renderizar artigos.html .

Algumas dicas importantes:

1. Utilize o retorno da função `url.parse()` para capturar o *pathname* digitado e renderizar o HTML correspondente. Se o pathname estiver vazio, significa que deve renderizar a página de artigos, e se estiver com um valor diferente do nome dos arquivos HTML, renderize a página de erros.
2. Você também pode inserir conteúdo HTML na função `response.end(html)` , economizando linha de código ao não usar a função `response.write(html)` .
3. Utilize a função `fs.existsSync(html)` para verificar se existe o HTML com o mesmo nome do pathname digitado.

O resultado desse desafio se encontra na página GitHub deste livro, em <https://github.com/caio-ribeiro-pereira/livro-nodejs/tree/master/desafio-1>.

# POR QUE O ASSÍNCRONO?

## 3.1 DESENVOLVENDO DE FORMA ASSÍNCRONA

É importante focar no uso das chamadas assíncronas quando trabalhamos com Node.js, assim como entender quando elas são invocadas.

O código a seguir exemplifica as diferenças entre uma função síncrona e assíncrona em relação à linha do tempo na qual elas são executadas. Basicamente, criaremos um loop de 5 iterações, sendo que a cada iteração será criado um arquivo texto com o mesmo conteúdo `Hello Node.js!` .

Primeiro, começaremos com o código síncrono. Crie o arquivo `text_sync.js` com o código a seguir:

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

for(var i = 1; i <= 5; i++) {
 var file = "sync-txt" + i + ".txt";
 var out = fs.writeFileSync(file, "Hello Node.js!");
 console.log(out);
}
```

Agora vamos criar o arquivo `text_async.js` , com seu respectivo código, diferente apenas na forma de chamar a função

`writeFileSync` , que será a versão assíncrona `writeFile` , recebendo uma função como argumento:

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

for(var i = 1; i <= 5; i++) {
 var file = "async-txt" + i + ".txt";
 fs.writeFile(file, "Hello Node.js!", function(err, out) {
 console.log(out);
 });
}
```

Vamos rodar? Execute os comandos `node text_sync` e, depois, `node text_async` . Se forem gerados 10 arquivos no mesmo diretório do código-fonte, então deu tudo certo. Mas a execução de ambos foi tão rápida que não foi possível ver as diferenças entre o `text_async` e o `text_sync` . Para entender melhor as diferenças, veja as *timelines* que foram geradas. O `text_sync` , por ser um código síncrono, invocou chamadas de I/O bloqueantes, gerando o seguinte gráfico:

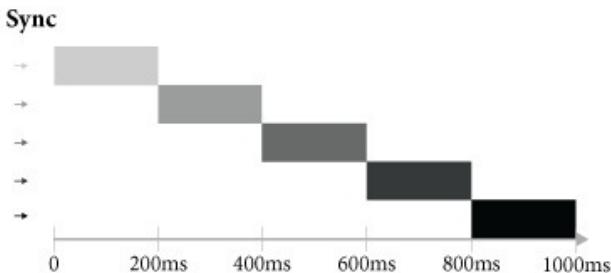


Figura 3.1: Timeline síncrona bloqueante

Repare no tempo de execução. O `text_sync` demorou 1.000 milissegundos, isto é, 200 milissegundos para cada arquivo criado.

Já em `text_async` , foram criados os arquivos de forma totalmente assíncrona, ou seja, as chamadas de I/O eram não

bloqueantes, sendo executadas totalmente em paralelo:

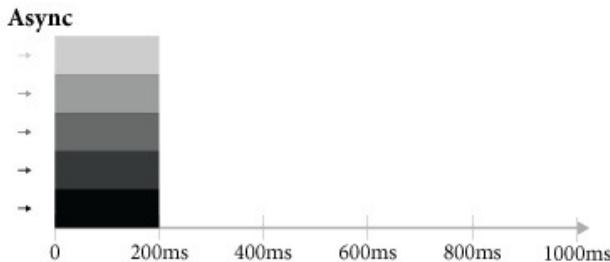


Figura 3.2: Timeline assíncrona não bloqueante

Isto fez com que o tempo de execução levasse 200 milissegundos, afinal, foi invocada 5 vezes, em paralelo, a função `fs.writeFileSync()`, maximizando processamento e minimizando o tempo de execução.

## THREADS VERSUS ASSÍNCRONISMOS

Por mais que as funções assíncronas possam executar em paralelo várias tarefas, elas jamais serão consideradas uma Thread (por exemplo, Threads do Java). A diferença é que Threads são manipuláveis pelo desenvolvedor, ou seja, você pode pausar a execução de uma Thread, ou fazê-la esperar o término de uma outra. Chamadas assíncronas apenas invocam suas funções em uma ordem de que você não tem controle, e você só sabe quando uma chamada terminou quando seu callback é executado.

Pode parecer vantajoso ter o controle sobre as Threads a favor de um sistema que executa tarefas em paralelo, mas pouco domínio sobre eles pode transformar seu sistema em um caos de travamentos *dead-locks*, afinal, Threads são executadas de forma bloqueante. Este é o grande diferencial das chamadas assíncronas; elas executam em paralelo suas funções sem travar processamento das outras e, principalmente, sem bloquear o sistema principal.

É fundamental que o seu código Node.js invoque o mínimo possível de funções bloqueantes. Toda função síncrona impedirá, naquele instante, que o Node.js continue executando os demais códigos até que aquela função seja finalizada. Por exemplo, se essa função fizer um *I/O* em disco, ele vai bloquear o sistema inteiro, deixando o processador ocioso enquanto ele usa outros recursos de hardware, como por exemplo, leitura em disco, utilização da rede

etc.

Sempre que puder, use funções assíncronas para aproveitar essa característica principal do Node.js.

Talvez você ainda não esteja convencido. A próxima seção vai lhe mostrar como e quando utilizar bibliotecas assíncronas não bloqueantes, tudo isso por meio de teste prático.

## 3.2 ASSINCRONISMO VERSUS SINCRONISMO

Para exemplificar melhor, os códigos adiante representam um benchmark comparando o tempo de bloqueio de execução **assíncrona versus síncrona**. Para isso, crie 3 arquivos: `processamento.js`, `leitura_async.js` e `leitura_sync.js`.

Criaremos isoladamente o código `leitura_async.js` que faz leitura assíncrona:

```
var fs = require('fs');
var leituraAsync = function(arquivo){
 console.log("Fazendo leitura assíncrona");
 var inicio = new Date().getTime();
 fs.readFile(arquivo);
 var fim = new Date().getTime();
 console.log("Bloqueio assíncrono: "+(fim - inicio)+ "ms");
};
module.exports = leituraAsync;
```

Em seguida, criaremos o código `leitura_sync.js`, que faz leitura síncrona:

```
var fs = require('fs');
var leituraSync = function(arquivo){
 console.log("Fazendo leitura síncrona");
 var inicio = new Date().getTime();
 fs.readFileSync(arquivo);
```

```
var fim = new Date().getTime();
console.log("Bloqueio síncrono: "+(fim - inicio)+ "ms");
};

module.exports = leituraSync;
```

Para finalizar, carregamos os dois tipos de leituras dentro do código `processamento.js`:

```
var http = require('http');
var fs = require('fs');
var leituraAsync = require('./leitura_async');
var leituraSync = require('./leitura_sync');
var arquivo = "./node.exe";
var stream = fs.createWriteStream(arquivo);
var download = "http://nodejs.org/dist/latest/node.exe";
http.get(download, function(res) {
 console.log("Fazendo download do Node.js");
 res.on('data', function(data){
 stream.write(data);
 });
 res.on('end', function(){
 stream.end();
 console.log("Download finalizado!");
 leituraAsync(arquivo);
 leituraSync(arquivo);
 });
});
```

Rode o comando `node processamento.js` para executar o benchmark. E agora, ficou clara a diferença entre o modelo bloqueante e o não bloqueante?

Parece que o método `readFile` executou muito rápido, mas não quer dizer que o arquivo foi lido. Ele recebe um último parâmetro, que é um callback indicando quando o arquivo foi lido, que não passamos na invocação que fizemos.

Ao usar o `fs.readFileSync()`, bastaria fazer `var conteudo = fs.readFileSync()`. Mas qual é o problema dessa abordagem?  
**Ela segura todo o mecanismo do Node.JS!**

Basicamente, esse código fez o download de um arquivo grande (código-fonte do Node.js) e, quando terminou, realizou um benchmark comparando o **tempo de bloqueio** entre as funções de leitura síncrona (`fs.readFileSync()`) e assíncrona (`fs.readFile()`) do Node.js.

A seguir, apresento o resultado do benchmark realizado em minha máquina:

- Modelo: MacBook Air 2011
- Processador: Core i5
- Memória: 4GB RAM
- Disco: 128GB SSD

Veja a pequena (porém significante) diferença de tempo entre as duas funções de leitura:

```
Fazendo download do Node.js
Download finalizado!
Fazendo leitura assíncrona
Bloqueio assíncrono: 0ms
Fazendo leitura síncrona
Bloqueio síncrono: 10ms
```

Figura 3.3: Benchmark de leitura Async vs. Sync

Se esse teste foi com um arquivo de mais ou menos 50 MB, imagine-o em larga escala, lendo múltiplos arquivos de 1 GB ao mesmo tempo, ou realizando múltiplos uploads em seu servidor. Esse é um dos pontos fortes do Node.js!

### 3.3 ENTENDENDO O EVENT-LOOP

Realmente trabalhar de forma assíncrona tem ótimos

benefícios em relação ao processamento I/O. Isso acontece devido ao fato de que uma chamada de I/O é considerada uma tarefa muito custosa para um computador realizar. Tão custosa que chega a ser perceptível para um usuário, por exemplo, quando ele tenta abrir um arquivo de 1 GB, e o sistema operacional trava alguns segundos para abri-lo.

Vendo o contexto de um servidor, por mais potente que seja seu *hardware*, eles terão os mesmos bloqueios perceptíveis pelo usuário; a diferença é que um servidor estará lidando com milhares usuários requisitando I/O, com a grande probabilidade de ser ao mesmo tempo.

É por isso que o Node.js trabalha com assincronismo. Ele permite que você desenvolva um sistema totalmente orientado a eventos, tudo isso graças ao *Event-loop*. Ele é um mecanismo interno, dependente das bibliotecas da linguagem C: libev (<http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod>) e libeio (<http://software.schmorp.de/pkg/libeio.html>), responsáveis por prover o assíncrono I/O no Node.js.

A figura a seguir apresenta como funciona o Event-Loop:



Figura 3.4: Event-Loop do Node.js

Basicamente, ele é um loop infinito, que em cada iteração verifica se existem novos eventos em sua **fila de eventos**. Tais eventos somente aparecem nessa fila quando são emitidos durante suas interações na aplicação.

O `EventEmitter` é o módulo responsável por emitir eventos, e a maioria das bibliotecas do Node.js herdam desse módulo suas funcionalidades de eventos. Quando um determinado código emite um evento, ele é enviado para a **fila de eventos** para que o *Event-loop* execute-o e, em seguida, retorne seu *callback*. Tal callback pode ser executado por meio de uma função de escuta, semanticamente conhecida pelo nome: `on()`.

Programar orientado a eventos vai manter sua aplicação mais robusta e estruturada para lidar com eventos que são executados de forma assíncrona não bloqueantes.

Para conhecer mais sobre as funcionalidades do EventEmitter , acesse sua documentação em <http://nodejs.org/api/events.html>.

## 3.4 EVITANDO CALLBACKS HELL

De fato, vimos o quanto é vantajoso e performático trabalhar de forma assíncrona. Porém, em certos momentos, inevitavelmente implementaremos diversas funções assíncronas, que serão encadeadas uma na outra através das suas funções callback. No código a seguir, apresentarei um exemplo desse caso.

Crie um arquivo chamado `callback_hell.js` , implemente e execute o código:

```
var fs = require('fs');
fs.readdir(__dirname, function(error, contents) {
 if (error) { throw error; }
 contents.forEach(function(content) {
 var path = './' + content;
 fs.stat(path, function(error, stat) {
 if (error) { throw error; }
 if (stat.isFile()) {
 console.log('%s %d bytes', content, stat.size);
 }
 });
 });
});
```

Reparem na quantidade de callbacks encadeados que existem em nosso código. Detalhe: ele apenas faz uma simples leitura dos arquivos de seu diretório, e imprime na tela seu nome e tamanho em bytes. Uma pequena tarefa como essa deveria ter menos encadeamentos, concorda?

Agora, imagine como seria a organização disso para realizar

tarefas mais complexas? Praticamente o seu código seria um caos e totalmente difícil de fazer manutenções.

Por ser assíncrono, você perde o controle do que está executando em troca de ganhos com performance. Porém, um detalhe importante sobre assincronismo é que, na maioria dos casos, os callbacks bem elaborados possuem como parâmetro uma variável de erro. Verifique nas documentações sobre sua existência e sempre faça o tratamento deles na execução do seu callback: `if (erro) { throw erro; }`. Isso vai impedir a continuação da execução aleatória quando for identificado um erro.

Uma boa prática de código JavaScript é criar funções que expressem seu objetivo e de forma isolada, salvando em variável e passando-as como callback. Em vez de criar funções anônimas, por exemplo, crie um arquivo chamado `callback_heaven.js` com o código a seguir:

```
var fs = require('fs');
var lerDiretorio = function() {
 fs.readdir(__dirname, function(erro, diretorio) {
 if (erro) return erro;
 diretorio.forEach(function(arquivo) {
 ler(arquivo);
 });
 });
 var ler = function(arquivo) {
 var path = './' + arquivo;
 fs.stat(path, function(erro, stat) {
 if (erro) return erro;
 if (stat.isFile()) {
 console.log('%s %d bytes', arquivo, stat.size);
 }
 });
 };
 lerDiretorio();
}
```

Veja o quanto melhorou a legibilidade do seu código. Dessa forma, deixamos mais semântico e legível o nome das funções, e diminuímos o número de encadeamentos das funções de callback. A boa prática é ter o bom senso de manter no máximo até dois encadeamentos de callbacks. Ao passar disso, significa que está na hora de criar uma função externa para ser passada como parâmetro nos callbacks, em vez de continuar criando um *callback hell* em seu código.

## CAPÍTULO 4

# INICIANDO COM O EXPRESS

## 4.1 POR QUE UTILIZÁ-LO?

Programar utilizando apenas a API (*Application Programming Interface*) HTTP nativa é muito trabalhoso! Conforme surgem necessidades de implementar novas funcionalidades, códigos gigantescos seriam acrescentados, aumentando a complexidade do projeto e dificultando futuras manutenções.

Foi a partir desse problema que surgiu um framework muito popular, que se chama Express. Ele é um módulo para desenvolvimento de aplicações web de grande escala. Sua filosofia de trabalho foi inspirada pelo framework Sinatra da linguagem Ruby. O site oficial do projeto é <http://expressjs.com>.



Figura 4.1: Framework Express

Ele possui as seguintes características:

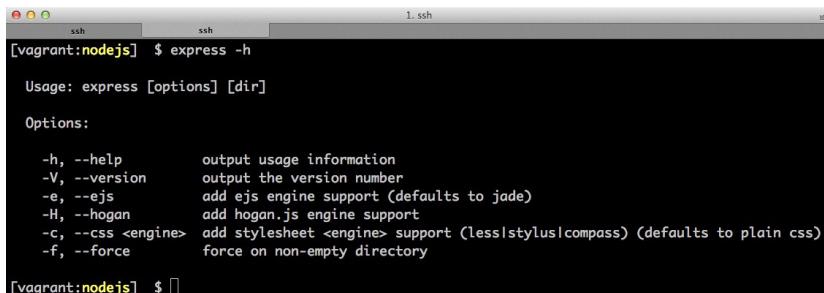
- MVR (*Model-View-Routes*);
- MVC (*Model-View-Controller*);
- Roteamento de URLs via callbacks;
- Middleware;
- Interface *RESTful*;
- Suporte a *File Uploads*;
- Configuração baseado em variáveis de ambiente;
- Suporte a *helpers* dinâmicos;
- Integração com *Template Engines*;
- Integração com SQL e NoSQL.

## 4.2 INSTALAÇÃO E CONFIGURAÇÃO

Sua instalação é muito simples e há algumas opções de configurações para começar um projeto. Para aproveitar todos os seus recursos, recomendo que também instale seu gerador inicial de aplicação, que se chama `express-generator` :

```
npm install -g express-generator
```

Feito isso, será necessário fechar e abrir seu terminal para habilitar o comando `express`, que é um CLI (*Command Line Interface*) do framework. Esse comando permite gerar um projeto inicial com algumas configurações básicas para definir, como a escolha de um *template engine* (por padrão, ele inclui o framework Jade, mas é possível escolher o EJS ou Hogan) e um CSS engine (por padrão, ele utiliza CSS puro, mas é possível escolher o LESS, Stylus ou Compass). Para visualizar todas as opções, execute o comando `express -h`.



```
[vagrant:nodejs] $ express -h
Usage: express [options] [dir]

Options:

-h, --help output usage information
-V, --version output the version number
-e, --ejs add ejs engine support (defaults to jade)
-H, --hogan add hogan.js engine support
-c, --css <engine> add stylesheet <engine> support (less|stylus|compass) (defaults to plain css)
-f, --force force on non-empty directory

[vagrant:nodejs] $
```

Figura 4.2: Express em modo CLI

## 4.3 CRIANDO UM PROJETO DE VERDADE

Vamos criar uma aplicação de verdade com Express? Dessa vez, criaremos um projeto que será trabalhado durante os demais capítulos do livro. Vamos criar uma agenda de contatos em que seus contatos serão integrados em um web chat funcionando em real-time.

Os requisitos do projeto são:

- O usuário deve criar, editar ou excluir um contato;
- O usuário deve se logar informando seu nome e e-mail;
- O usuário deve conectar ou desconectar no chat;
- O usuário deve enviar e receber mensagens no chat somente entre os contatos online;

O nome do projeto será Ntalk (*Node talk*) e usaremos as seguintes tecnologias:

- **Node.js:** back-end do projeto;
- **MongoDB:** banco de dados NoSQL orientado a documentos;

- **MongooseJS**: ODM (*Object Data Mapper*) MongoDB para Node.js;
- **Redis**: banco de dados NoSQL para estruturas de chave-valor;
- **Express**: framework para aplicações web;
- **Socket.IO**: módulo para comunicação real-time;
- **Node Redis**: cliente Redis para Node.js;
- **EJS**: *template engine* para implementação de HTML dinâmico;
- **Mocha**: framework para testes automatizados;
- **SuperTest**: módulo para emular requisições que será utilizado no teste de integração;
- **Nginx**: servidor web de alta performance para arquivos estáticos.

Exploraremos essas tecnologias no decorrer desses capítulos, então muita calma e boa leitura!

Caso você esteja com pressa de ver esse projeto rodando, você pode cloná-lo através do meu repositório público, em <https://github.com/caio-ribeiro-pereira/livro-nodejs>.

Para instalá-lo em sua máquina, faça os comandos a seguir:

```
git clone git@github.com:caio-ribeiro-pereira/livro-nodejs.git
cd livro-nodejs/projeto/ntalk
npm install
npm start
```

Depois, accesse no seu navegador favorito o endereço:  
<http://localhost:3000>

Agora se você quer aprender passo a passo a desenvolver esse projeto, continue lendo este livro, seguindo todas as dicas que irei

passar.

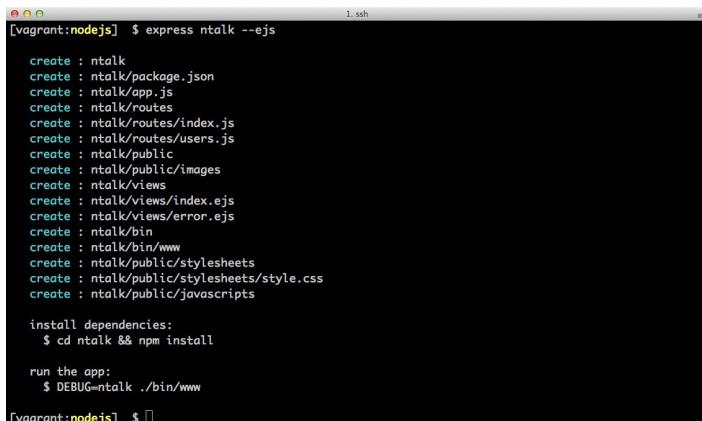
## 4.4 GERANDO SCAFFOLD DO PROJETO

Criaremos o diretório da aplicação já com alguns recursos do Express que é gerado a partir de seu CLI. Para começar, execute os seguintes comandos:

```
express ntalk --ejs
cd ntalk
npm install
```

Parabéns! Você acabou de criar o projeto `ntalk`.

Ao acessar o diretório do projeto, veja como foi gerado o seu *scaffold*:



```
[vagrant:nodejs] $ express ntalk --ejs
create : ntalk
create : ntalk/package.json
create : ntalk/app.js
create : ntalk/routes
create : ntalk/routes/index.js
create : ntalk/routes/users.js
create : ntalk/public
create : ntalk/public/images
create : ntalk/views
create : ntalk/views/index.ejs
create : ntalk/views/error.ejs
create : ntalk/bin
create : ntalk/bin/www
create : ntalk/public/stylsheets
create : ntalk/public/stylsheets/style.css
create : ntalk/public/javascripts

install dependencies:
$ cd ntalk && npm install

run the app:
$ DEBUG=ntalk ./bin/www
[vagrant:nodejs] $ []
```

Figura 4.3: Estrutura do Express

- `package.json` : contém as principais informações sobre a aplicação, como nome, autor, versão, colaboradores, URL, dependências e muito mais;

- `public` : pasta pública que armazena conteúdo estático, por exemplo, imagens, CSS, JavaScript etc.;
- `app.js` : arquivo que inicializa o servidor do projeto, através do comando `node app.js` ;
- `routes` : diretório que mantém todas as rotas da aplicação;
- `views` : diretório que contém todas as *views* que são renderizadas pelas rotas;
- `bin` : diretório com um arquivo que permite iniciar a aplicação via linha de comando.

Ao rodarmos o comando `npm install` , por padrão ele instalou as dependências existentes no `package.json` . Neste caso, ele vai instalar o Express, o EJS (*Embedded Javascript*) e alguns middlewares do Express: `debug` , `body-parser` , `static-favicon` , `morgan` e o `cookie-parser` .

Agora faremos algumas alterações nos códigos gerados pelo comando `express` . O primeiro passo será criar uma **descrição sobre o projeto**, definir `false` no atributo `private` e, por último, remover o atributo `scripts` (afinal, esse atributo será melhor explicado em um capítulo futuro). Isso tudo será modificado no arquivo `package.json` . Veja como deve ficar:

```
{
 "name": "ntalk",
 "description": "Node talk - Agenda de contatos",
 "version": "0.0.1",
 "private": false,
 "dependencies": {
 "express": "~4.2.0",
 "static-favicon": "~1.0.0",
 "morgan": "~1.0.0",
 "cookie-parser": "~1.0.1",
 "body-parser": "~1.0.0",
```

```
 "debug": "~0.7.4",
 "ejs": "~0.8.5"
 }
}
```

Para simplificar nosso trabalho, vamos remover algumas coisas que foram geradas pelo comando `express`. Primeiro, exclua o diretório `bin`, pois não vamos utilizá-lo; vamos aplicar uma técnica mais simples de iniciar um servidor que será otimizada no decorrer do livro. Em seguida, vamos remover os seguintes módulos: `morgan`, `debug` e `static-favicon`, que também não serão usados.

```
npm remove debug static-favicon morgan --save
```

Vamos atualizar os demais módulos do `package.json`, afinal, estamos começando do zero o projeto e não há nada melhor do que começar utilizando as últimas versões dos frameworks em nossa aplicação.

Para usar as últimas versões do Express, EJS e demais middlewares, faremos o seguinte truque: modifique no `package.json` as versões de cada módulo, substituindo-os o caractere `~` pelo caractere `>`, deixando-os da seguinte maneira:

```
"dependencies": {
 "express": "> 4.2.0",
 "cookie-parser": "> 1.0.1",
 "body-parser": "> 1.0.0",
 "ejs": "> 0.8.5"
}
```

Dessa forma, vamos garantir uma atualização desses módulos para suas últimas versões. Agora, para atualizá-los, basta executar o comando:

```
npm update --latest --save
```

Até o momento em que este livro esta sendo escrito, as dependências do package.json estão usando as seguintes versões:

```
"dependencies": {
 "express": "~4.6.1",
 "cookie-parser": "~1.3.2",
 "body-parser": "~1.4.3",
 "ejs": "~1.0.0"
}
```

Pronto! Agora que temos as últimas versões dos principais módulos, vamos modificar o app.js , deixando-o com o mínimo de código possível para explicarmos em *baby-steps* o que realmente vamos usar no desenvolvimento deste projeto. Recomendo que **apague todo código gerado**, e coloque o seguinte código:

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

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

app.use('/', routes);
app.use('/usuarios', users);

app.listen(3000, function(){
 console.log("Ntalk no ar.");
});
```

## Como se inicia um servidor Express?

Essa versão inicialmente atende os requisitos mínimos de uma aplicação Express. A brincadeira começa quando executamos a função express() , pois o seu retorno habilita todas as suas

funcionalidades de seu framework, que armazenamos na variável `app`.

Com `app.listen()`, fazemos algo parecido com o `http.listen()`, ou seja, ele é um *alias* responsável por colocar a aplicação no ar através de uma porta da rede.

## Como se configura os middlewares?

O `app.set(chave, valor)` é uma estrutura de chave e valor mantida dentro da variável `app`. Seria o mesmo que criar no JavaScript o código `app["chave"] = "valor";`. Um exemplo prático são as configurações de `views` que foram definidas no código anterior: `(app.set('views', __dirname + '/views'))` e `(app.set('view engine', 'ejs'))`.

## DETALHES SOBRE O EXPRESS 4

Um detalhe importante a explicar é que, em versões antigas do Express (por exemplo, o Express 3.x), a maioria das funções chamadas diretamente pela variável `express` eram herdadas de seus submódulos `Connect` e `HTTP`.

Hoje estamos usando uma nova versão do Express, que é a versão 4.x. Uma das grandes mudanças nessa versão é que não existe mais esse tipo de herança, sendo necessário que você instale por conta própria tais submódulos e, principalmente, os middlewares necessários para trabalhar em sua aplicação.

Um bom exemplo disso são os módulos: `body-parser`, `cookie-parser` e `morgan`. Eles eram middlewares internos do Express 3.x, que eram chamados através da variável `express`.

O único middleware ainda existente no Express é o middleware responsável por servir arquivos estáticos: `express.static`.

Caso queira conhecer mais a fundo as grandes mudanças entre as versões Express 3.x para Express 4.x, recomendo que leia atentamente este link:  
<https://github.com/visionmedia/express/wiki/Migrating-from-3.x-to-4.x>.

Por enquanto, a nossa aplicação possui apenas dois

middlewares configurados: um é *template engine* EJS e o outro é o servidor de arquivos estáticos. No decorrer deste livro, serão incluídos e explicados novos middlewares.

## Como se cria rotas no Express?

De início, existem apenas duas rotas em nossa aplicação: / e /usuarios .

Reparam como são executados seus respectivos callbacks; eles vieram das variáveis var routes = require('./routes/index') e var users = require('./routes/users') . Ou seja, apenas carregamos seus respectivos módulos do diretório routes , cada um contendo suas regras de negócio.

## 4.5 ORGANIZANDO OS DIRETÓRIOS DO PROJETO

Quando o assunto é organização de códigos, o Express se comporta de forma bem flexível e liberal. Temos a total liberdade de modificar sua estrutura de diretórios e arquivos. Tudo vai depender da complexidade do projeto e, principalmente, conhecimento sobre boas práticas de organização e convenções.

Por exemplo, se o projeto for um sistema *single-page*, você pode desenvolver todo back-end dentro código app.js , ou se o projeto possuir diversas rotas, *views*, *models* e *controllers*. O ideal é que seja montado uma estrutura modularizável permitindo a utilização do *pattern* que melhor se encaixa em seu projeto.

Em nosso projeto, vamos o implementar padrão MVC. Para

isso, crie os seguintes diretórios `models` e `controllers`, deixando sua estrutura dessa forma:

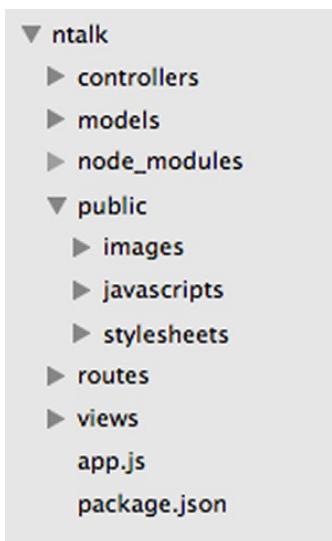


Figura 4.4: Estrutura de diretórios do ntalk

Cada *model* que for usado em um determinado *controller* realizará uma chamada a função `require('/models/nome-do-model');`. Em *controllers*, ou qualquer outro código, diversas chamadas à função `require` serão realizadas, e isso pode gerar uma poluição carregamento de módulos em um código. Com base nesse problema, surgiu um *plugin* que visa minimizar essas terríveis chamadas, o `express-load`, sendo responsável por mapear diretórios para carregar e injetar módulos dentro de uma variável que definirmos na função `load('módulos').into(app)`.

Para entender melhor o que será feito, primeiro instale-o no projeto:

```
npm install express-load --save
```

Em seguida, vamos utilizar o `express-load` aplicando alguns *refactorings*. Primeiro, vamos remover as rotas existentes no `app.js`, deixando-o desse jeito:

```
var express = require('express')
, app = express()
;

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

app.listen(3000, function(){
 console.log("Ntalk no ar.");
});
```

Para finalizar, implementaremos o `express-load` no `app.js`, já configurando-o para carregar qualquer módulo dos diretórios: *models*, *controllers* e *routes*.

```
var express = require('express')
, load = require('express-load')
, app = express()

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.static(__dirname + '/public'));

load('models')
.then('controllers')
.then('routes')
.into(app);

app.listen(3000, function(){
 console.log("Ntalk no ar.");
});
```

## ATENÇÃO

É importante colocar em ordem os recursos a serem carregados pela função `load()`. Neste caso, os *models* são carregados primeiro, para que os *controllers* possam usá-las e, por último, os *routes* usarem toda lógica de seus *controllers*.

Continuando o *refactoring*, exclua os arquivos `routes/user.js` e `routes/index.js` que foram gerado pelo Express. Não os usaremos mais; vamos criar do zero nossas próprias rotas!

Para criar nossa primeira rota, crie o seguinte arquivo `routes/home.js` com a lógica:

```
module.exports = function(app) {
 var home = app.controllers.home;
 app.get('/', home.index);
};
```

Repare que por causa do `express-load`, a variável `app` já possui um subobjeto chamado `controllers`. Neste caso, o `app.controllers.home` está se referenciando ao arquivo `controllers/home.js`, que vamos criar agora. De início, criaremos com apenas uma única função (neste caso, as funções de *controllers* são conhecidas pelo nome *action*). Veja:

```
module.exports = function(app) {
 var HomeController = {
 index: function(req, res) {
 res.render('home/index');
 }
};
```

```
 return HomeController;
};
```

Para terminar o fluxo entre *controllers* e *routes*, temos que mostrar uma *view* que basicamente é uma página HTML mostrando algum resultado para o usuário. Exclua o arquivo *views/index.ejs* que foi criado pelo *express-generator* , e crie o arquivo *views/home/index.ejs* que será nossa homepage com tela de login para acessar o sistema.

Para fins ilustrativos, a lógica desse aplicativo será de implementar um login que autocadastra um novo usuário, quando for informado um login novo no sistema. Em *views/home/index.ejs* , vamos criar um simples formulário que conterá os campos *nome* e *email* . Veja como será essa *view* com base na implementação a seguir:

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>Ntalk - Agenda de contatos</title>
 </head>
 <body>
 <header>
 <h1>Ntalk</h1>
 <h4>Bem-vindo!</h4>
 </header>
 <section>
 <form action="/entrar" method="post">
 <input type="text" name="usuario[nome]"
 placeholder="Nome">

 <input type="text" name="usuario[email]"
 placeholder="E-mail">

 <button type="submit">Entrar</button>
 </form>
 </section>
```

```
<footer>
 <small>Ntalk - Agenda de contatos</small>
</footer>
</body>
</html>
```

Vamos rodar o projeto? Execute no terminal o comando `node app.js` e, em seguida, acesse em seu *browser* o endereço: <http://localhost:3000>.

Se tudo deu certo, você verá uma tela com formulário semelhante à figura:

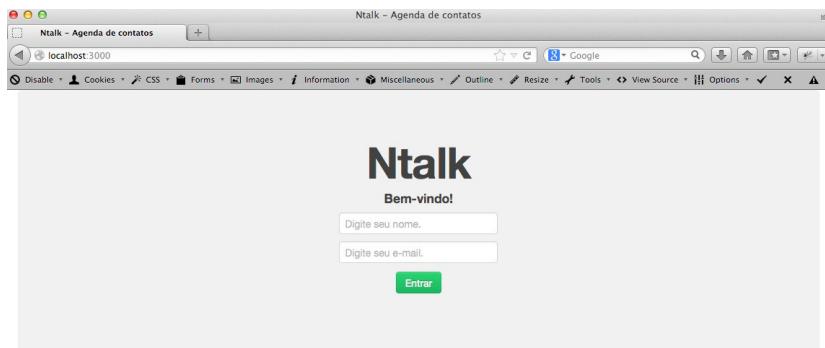


Figura 4.5: Tela de login do Ntalk

## UM DETALHE A INFORMAR

Nos exemplos deste livro, não será implementado código CSS ou boas práticas de HTML, pois vamos focar apenas em código JavaScript. Algumas imagens do projeto serão apresentados com um layout customizado. A versão completa desse projeto que inclui o código CSS e HTML de acordo com os *screenshots* pode ser acessada por meio do meu GitHub, em <http://github.com/caio-ribeiro-pereira/livro-nodejs/tree/master/projeto/ntalk>.

Parabéns! Acabamos de implementar o fluxo da tela inicial do projeto, continue a leitura que tem muito mais pela frente!

## CAPÍTULO 5

# DOMINANDO O EXPRESS

## 5.1 ESTRUTURANDO VIEWS

O módulo EJS possui diversas funcionalidades que permitem programar conteúdo dinâmico em cima de código HTML. Não entraremos a fundo neste framework, apenas usaremos seus principais recursos para renderizar conteúdo dinâmico e minimizar repetições de código.

Com isso, isolaremos em outras *views*, conhecidas como *partials*, possíveis códigos que serão reutilizados com maior frequência em outras *views*. Dentro do diretório `views` , vamos criar dois arquivos que serão reaproveitados em todas as páginas. O primeiro será o cabeçalho com o nome `header.ejs` :

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>Ntalk - Agenda de contatos</title>
 </head>
 <body>
```

E o segundo será o rodapé `footer.ejs` :

```
<footer>
 <small>Ntalk - Agenda de contatos</small>
</footer>
```

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

Agora, modificaremos a nossa homepage, a `views/home/index.ejs`, para que chame esses *partials* através da função `include`:

```
<% include ../header %>
<header>
 <h1>Ntalk</h1>
 <h4>Bem-vindo!</h4>
</header>
<section>
 <form action="/entrar" method="post">
 <input type="text" name="usuario[nome]"
 placeholder="Nome">

 <input type="text" name="usuario[email]"
 placeholder="E-mail">

 <button type="submit">Entrar</button>
 </form>
</section>
<% include ../footer %>
```

Sua homepage ficou mais enxuta, fácil de ler e estruturada para reutilização de *partials*.

## 5.2 CONTROLANDO AS SESSÕES DE USUÁRIOS

Para o sistema fazer *login* e *logout*, é necessário ter um controle de sessão. Esse controle permitirá que o usuário mantenha seus principais dados de acesso em memória no servidor, pois esses dados serão usados com maior frequência por grande parte do sistema enquanto o usuário estiver usando a aplicação.

Trabalhar com sessão é muito simples, e os dados são

manipulados por meio de objeto JSON dentro da estrutura `req.session`.

Para aplicar sessão, será necessário instalar um novo módulo, o `express-session`. Para isso, execute:

```
npm install express-session --save
```

Antes de implementarmos a sessão, vamos criar duas novas rotas dentro de `routes/home.js`. Uma rota será para *login* via função: `app.post('/entrar', home.login)`, e a outra será uma rota para *logout* via função: `app.get('/sair', home.logout)`.

```
module.exports = function(app) {
 var home = app.controllers.home;
 app.get('/', home.index);
 app.post('/entrar', home.login);
 app.get('/sair', home.logout);
};
```

Depois, implementaremos suas respectivas *actions* no `controller/home.js`, seguindo a convenção de nomes `login` e `logout`, que foram utilizados no `routes/home.js`.

Agora temos que codificar suas respectivas *actions* no `controllers/home.js`. Na *action login*, será implementada uma simples regra para validar se existe valores nos campos `nome` e `email`, que serão submetidos pelo futuro formulário de login da aplicação. Se os campos passarem na validação, esses dados serão armazenados na sessão através da estrutura `req.session.usuario`, e também criaremos um *array* vazio (`usuario['contatos'] = [];`) que mantém uma lista simples de contatos para o usuário logado. Se for um usuário válido, ele será redirecionado para rota `/contatos`, **que vamos criar no futuro.**

Já a *action logout*, será chamada apenas para executar a função `req.session.destroy()` que limpará os dados da sessão e, por fim, redirecionará o usuário para homepage. Veja como ficarão essas lógicas no arquivo `controllers/home.js`:

```
module.exports = function(app) {
 var HomeController = {
 index: function(req, res) {
 res.render('home/index');
 },
 login: function(req, res) {
 var email = req.body.usuario.email
 , nome = req.body.usuario.nome
 ;
 if(email && nome) {
 var usuario = req.body.usuario;
 usuario['contatos'] = [];
 req.session.usuario = usuario;
 res.redirect('/contatos');
 } else {
 res.redirect('/');
 }
 },
 logout: function(req, res) {
 req.session.destroy();
 res.redirect('/');
 }
 };
 return HomeController;
};
```

Após criarmos as regras de *login/logout*, vamos criar a tela inicial de contatos para testarmos o redirecionamento de sucesso quando um usuário entrar na aplicação. De início, este *controller* vai renderizar o nome do usuário logado, ou seja, o nome do usuário que existir na sessão da aplicação. Para isso, enviaremos a variável `req.session.usuario` na renderização de sua *view* através da função `res.render()`.

Veja como fazer isso, criando o arquivo controllers/contatos.js :

```
module.exports = function(app) {
 var ContatosController = {
 index: function(req, res) {
 var usuario = req.session.usuario
 , params = {usuario: usuario};
 res.render('contatos/index', params);
 },
 };
 return ContatosController;
};
```

Agora, crie sua respectiva rota em routes/contatos.js :

```
module.exports = function(app) {
 var contatos = app.controllers.contatos;
 app.get('/contatos', contatos.index);
};
```

Para finalizar, vamos criar uma simples *view* para tela de contatos, a qual vamos melhorá-la no decorrer deste livro. Então, crie o arquivo views/contatos/index.ejs com o seguinte HTML:

```
<% include ../header %>
<header>
 <h1>Ntalk</h1>
 <h4>Lista de contatos</h4>
</header>
<section>
 <p>Bem-vindo <%- usuario.nome %></p>
</section>
<% include ../exit %>
<% include ../footer %>
```

Repare que foi adicionado a tag `<% include ../exit %>`. Este é mais um *partial*, em que vamos reaproveitar o link de logout nas *views* da aplicação, e seu código será criado em

views/exit.ejs :

```
<section>
 Sair
</section>
```

Pronto! Agora, que tal testar essas implementações? Dê um restart no servidor teclando no terminal **CTRL+C** (no Windows ou Linux), ou **Command+C** (no MacOSX). Em seguida, execute `node app.js` e, depois, em seu browser, tente fazer um login no sistema.

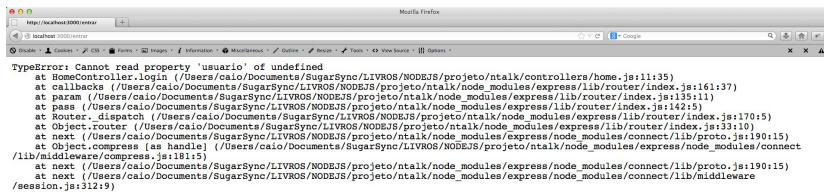


Figura 5.1: Infelizmente, deu mensagem de erro

"O que houve? Eu implementei tudo certo!"" Então, meu amigo, esse erro aconteceu porque faltou usar um middleware que já tínhamos instalado-o desde a criação do projeto, porém esquecemos de habilitá-lo. Seu nome é `body-parser`.

Esse middleware é responsável por receber os dados de um formulário e fazer um *parser* para objeto JSON, afinal, esse erro ocorreu porque não foi reconhecido o objeto `req.body.usuario`. Já adiantando, vamos habilitar o controle de sessão através do módulo `express-session` e também gerenciador de *cookies* do módulo `cookie-parser`, para que na próxima vez tudo funcione corretamente.

Para isso, edite o arquivo app.js incluindo esses middlewares seguindo a ordem do código a seguir, para que esses middlewares funcionem corretamente:

```
var express = require('express')
, load = require('express-load')
, bodyParser = require('body-parser')
, cookieParser = require('cookie-parser')
, expressSession = require('express-session')
, app = express()
;
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(cookieParser('ntalk'));
app.use(expressSession());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
app.use(express.static(__dirname + '/public'));

load('models')
.then('controllers')
.then('routes')
.into(app)
;
app.listen(3000, function(){
 console.log("Ntalk no ar.");
});
```

**Atenção!** É necessário incluir o cookieParser('ntalk') primeiro, para que o expressSession() utilize o mesmo SessionID que será mantido no cookie. Outras configurações habilitadas foram o bodyParser.json() e bodyParser.urlencoded({extended: true}) , que são responsáveis por criar objetos JSON vindos de um formulário HTML.

Esse middleware basicamente cria um objeto através dos atributos name e value , existentes nas tags <input> , <select> e <textarea> . Ao submeter um formulário com a tag

<input name="usuario[idade]" value="18"> , será criado um objeto dentro de req.body , neste caso, será criado o objeto req.body.usuario.idade com o valor 18.

#### ALGUNS CUIDADOS AO TRABALHAR COM SESSIONS

Qualquer nome informado dentro de req.session será armazenado como um subobjeto, por exemplo, req.session.mensagem = "Olá" . Cuidado para não sobreescriver suas funções nativas, como req.session.destroy ou req.session.regenerate . Ao fazer isso, você desabilita suas funcionalidades, fazendo com que, no decorrer de sua aplicação, inesperados bugs possam acontecer no sistema.

Para entender melhor as funções da session , veja em detalhes sua documentação em <https://github.com/expressjs/session>.

Agora sim, a nossa aplicação esta pronta para testes! Reinicie o servidor teclando no terminal CTRL+C (no Windows ou Linux) ou Command+C (no MacOSX), execute node app.js , e por último acesse em seu browser <http://localhost:3000>. Faça novamente um login no sistema. Dessa vez temos uma nova tela, que é a agenda de contatos.

## 5.3 CRIANDO ROTAS NO PADRÃO REST

A agenda de contatos precisa ter como requisito mínimo, um

meio de permitir o usuário criar, listar, atualizar e excluir seus contatos. Esse é o conjunto clássico de funcionalidades, mais conhecido como CRUD (*Create, Read, Update and Delete*).

As rotas que usaremos para implementar o CRUD da agenda de contatos seguirão o padrão de rotas REST. Esse padrão consiste em criar rotas utilizando os principais métodos do HTTP ( `GET` , `POST` , `PUT` e `DELETE` ). Para isso, vamos instalar um novo middleware, conhecido pelo nome de `method-override` :

```
npm install method-override --save
```

Após sua instalação, habilite este middleware editando o `app.js` :

```
var express = require('express')
, load = require('express-load')
, bodyParser = require('body-parser')
, cookieParser = require('cookie-parser')
, expressSession = require('express-session')
, methodOverride = require('method-override')
, app = express()
;
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(cookieParser('ntalk'));
app.use(expressSession());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
app.use(methodOverride('_method'));
app.use(express.static(__dirname + '/public'));

// ...continuação do app.js...
```

O middleware `methodOverride('_method')` permite que uma mesma rota seja reaproveitada entre métodos distintos do HTTP. Essa técnica é conhecida pelo nome de sobrescrita de métodos do HTTP.

Abra o arquivo `routes/contatos.js`. Nele, vamos implementar as futuras rotas do CRUD da agenda de contatos:

```
module.exports = function(app) {
 var contatos = app.controllers.contatos;
 app.get('/contatos', contatos.index);
 app.get('/contato/:id', contatos.show);
 app.post('/contato', contatos.create);
 app.get('/contato/:id/editar', contatos.edit);
 app.put('/contato/:id', contatos.update);
 app.delete('/contato/:id', contatos.destroy);
};
```

Com essas rotas implementadas, vamos agora codificar suas respectivas regras de negócio em `controllers/contatos.js`. Como ainda não estamos usando banco de dados, todos os dados serão persistidos na própria sessão do usuário, ou seja, todos os contatos serão gravados em memória, e ao sair da aplicação os contatos serão apagados.

Mas fique tranquilo! Nos próximos capítulos usaremos um banco de dados dedicado! Por enquanto, só vamos criar os fluxos essenciais da nossa aplicação.

Voltando ao nosso *controller* da agenda de contatos (`controllers/contatos.js`), implemente a seguinte lógica na *action index*:

```
module.exports = function(app) {
 var ContatoController = {
 index: function(req, res) {
 var usuario = req.session.usuario
 , contatos = usuario.contatos
 , params = {usuario: usuario
 , contatos: contatos};
 res.render('contatos/index', params);
 },
 // continuação do controller...
 }
};
```

Já na `action create`, utilizaremos um simples `array` para persistir os contatos do usuário – `usuario.contatos.push(contato)`; – para, em seguida, redirecionar o usuário para rota `/contatos`:

```
create: function(req, res) {
 var contato = req.body.contato
 , usuario = req.session.usuario;
 usuario.contatos.push(contato);
 res.redirect('/contatos');
},
// continuação do controller...
```

Em `show` e `edit`, enviamos via parâmetro no path, o ID do usuário. Neste caso, passaremos apenas o índice do contato referente à sua posição no array e, em seguida, enviamos o contato para a renderização de sua respectiva `view`:

```
show: function(req, res) {
 var id = req.params.id
 , contato = req.session.usuario.contatos[id]
 , params = {contato: contato, id: id};
 res.render('contatos/show', params);
},
edit: function(req, res) {
 var id = req.params.id
 , usuario = req.session.usuario
 , contato = usuario.contatos[id]
 , params = {usuario: usuario
 , contato: contato
 , id: id};
 res.render('contatos/edit', params);
},
// continuação do controller...
```

Agora temos a `action update`. Ela recebe os dados de um contato atualizado que são submetidos pelo formulário da view `contatos/edit`. Também usamos seu índice via `req.params.id` para atualizar o contato dentro do array.

```

update: function(req, res) {
 var contato = req.body.contato
 , usuario = req.session.usuario;
 usuario.contatos[req.params.id] = contato;
 res.redirect('/contatos');
},
// continuação do controller...

```

Por último, temos a *action* `destroy`, que basicamente recebe o índice por meio da variável `req.params.id`, e exclui o contato do array via função `usuario.contatos.splice(id, 1)`.

```

destroy: function(req, res) {
 var usuario = req.session.usuario
 , id = req.params.id;
 usuario.contatos.splice(id, 1);
 res.redirect('/contatos');
}
// fim do controller...
return ContatoController;
};

```

Esse foi um esboço muito básico da agenda de contatos. Porém, com ele foi possível explorar algumas características do Express para a implementação de um CRUD.

Para finalizar, vamos criar as views para o usuário interagir no sistema. Dentro do diretório `views/contatos`, vamos modificar a view `index.ejs` para renderizar uma lista de contatos, e um formulário para cadastrar novos contatos:

```

<% include ../header %>
<header>
 <h2>Ntalk - Agenda de contatos</h2>
</header>
<section>
 <form action="/contato" method="post">
 <input type="text" name="contato[nome]"
 placeholder="Nome">
 <input type="text" name="contato[email]"
 placeholder="E-mail">

```

```

 <button type="submit">Cadastrar</button>
 </form>
 <table>
 <thead>
 <tr>
 <th>Nome</th>
 <th>E-mail</th>
 <th>Ação</th>
 </tr>
 </thead>
 <tbody>
 <% contatos.forEach(function(contato, index) { %>
 <tr>
 <td><%= contato.nome %></td>
 <td><%= contato.email %></td>
 <td>
 <a href="/contato/<%= index %>">Detalhes
 </td>
 </tr>
 <% }) %>
 </tbody>
 </table>
</section>
<% include ../exit %>
<% include ../footer %>
```

Agora, no mesmo diretório, vamos criar o `edit.ejs` e implementar um formulário para o usuário atualizar os dados de um contato:

```

<% include ../header %>
<header>
 <h2>Ntalk - Editar contato</h2>
</header>
<section>
 <form action="/contato/<%= id %>?_method=put" method="post">
 <label>Nome:</label>
 <input type="text" name="contato[nome]"
 value="<%- contato.nome %>">
 <label>E-mail:</label>
 <input type="text" name="contato[email]"
 value="<%- contato.email %>">
 <button type="submit">Atualizar</button>
```

```
</form>
</section>
<% include ../exit %>
<% include ../footer %>
```

Por último e mais fácil de todos, crie a view `show.ejs`. Nela, vamos renderizar os dados de um contato que for selecionado. Incluiremos dois botões: **Editar** (para acessar a tela de edição do contato) e **Excluir** (botão de exclusão do contato atual).

```
<% include ../header %>
<header>
 <h2>Ntalk - Dados do contato</h2>
</header>
<section>
 <form action="/contato/<%- id %>?_method=delete"
 method="post">
 <p><label>Nome:</label><%- contato.nome %></p>
 <p><label>E-mail:</label><%- contato.email %></p>
 <p>
 <button type="submit">Excluir</button>
 <a href="/contato/<%- id %>/editar">Editar
 </p>
 </form>
</section>
<% include ../exit %>
<% include ../footer %>
```

## UTILIZANDO PUT E DELETE DO HTTP

Infelizmente, as especificações atuais do HTTP não dão suporte para usar os verbos `PUT` e `DELETE` de forma semântica em um código HTML, como por exemplo:

```
<form action="/editar" method="put">
<form action="/excluir" method="delete">
```

Há várias técnicas para utilizar esses verbos do HTTP, porém todas são soluções paliativa. Uma delas, pela qual aplicamos em nossa aplicação, é a inclusão de uma query string com nome `_method=put`, ou `_method=delete`, para que seja reconhecido pelo middleware `methodOverride('_method')` e faça uma sobrescrita dessas rotas para o método `PUT` ou `DELETE` da tag `<form>`. Veja um exemplo a seguir:

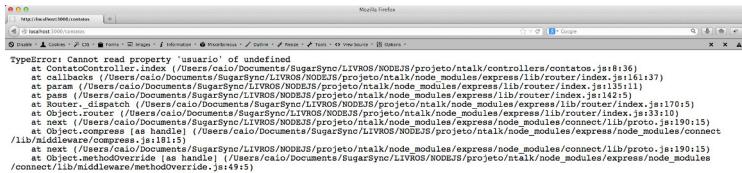
```
<form action="/contatos?_method=put" method="post">
<form action="/contatos?_method=delete" method="post">
```

## 5.4 APLICANDO FILTROS ANTES DE ACESSAR AS ROTAS

Já percebeu que quando acessamos a rota `/contatos` sem logar no sistema acontece um bug? Não? Tente agora mesmo! Mas o que realmente provocou este erro?

Bem, quando fazemos um login com sucesso, armazenamos os principais dados do usuário na sessão para utilizá-los no decorrer da aplicação. Porém, quando acessamos essa mesma rota sem antes

ter feito um login no sistema, o *controller* tenta acessar a variável `req.session.usuario` que não existe ainda. Isso causa o seguinte bug na aplicação:



```
MacBook-Pro:~/Documents/SugarSync/LIVROS/NODEJS/projeto/talk/controllers caio$ node index.js
TypeError: Cannot read property 'usuario' of undefined
 at ContatoController.index (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/talk/controllers/contato.js:8:36)
 at callbacks (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/talk/node_modules/express/lib/router/index.js:161:37)
 at param (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/talk/node_modules/express/lib/router/index.js:149:11)
 at next (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/talk/node_modules/express/lib/router/index.js:142:5)
 at Router._dispatch (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/talk/node_modules/express/lib/router/index.js:170:5)
 at Router.dispatch (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/talk/node_modules/express/lib/router/index.js:170:5)
 at next (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/talk/node_modules/express/lib/router/index.js:170:5)
 at Object.compress [as handle] (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/talk/node_modules/express/node_modules/connect/lib/proto.js:19:15)
 at Object.methodOverride [as handle] (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/talk/node_modules/express/node_modules/connect/lib/middleware/methodOverride.js:49:15)
```

Figura 5.2: Bug — Cannot read property 'usuario' of undefined

Diferente do Rails, Sinatra ou Django, o Express não possui filtros de forma explícita e em código legível. Não existe uma função de *hooks*, geralmente conhecidas pelo nome `before` ou `after` pela qual se possa processar algo antes ou depois de entrar em uma rota. Mas calma! Nem tudo está perdido!

Graças ao JavaScript, temos as funções de callback, e o próprio mecanismo de roteamento do Express utiliza muito bem esse recurso, permitindo a criação de callbacks encadeados em uma rota. Resumindo, quando criamos uma rota, após informar no primeiro parâmetro o seu path, no segundo parâmetro em diante é possível incluir callbacks que são executados de forma ordenada, por exemplo `app.get('/', callback1, callback2, callback3)`.

Para implementarmos isso, primeiro vamos criar um filtro de autenticação. Ele será a criação do nosso primeiro *middleware* e será usado na maioria das rotas. Na raiz do projeto, crie a pasta

*middlewares* incluindo nele o arquivo `autenticador.js`. Nele, implemente a lógica a seguir:

```
module.exports = function(req, res, next) {
 if(!req.session.usuario) {
 return res.redirect('/');
 }
 return next();
};
```

Esse filtro faz uma simples verificação da existência de um usuário dentro da sessão. Se o usuário estiver autenticado, ou seja, estiver na `session`, será executado o callback `return next()` responsável por pular este filtro e indo para função ao lado. Caso a autenticação não aconteça, executamos um simples `return res.redirect('/')`, que faz o usuário voltar para página inicial e impeça que ocorra o bug anterior.

Com esse filtro implementado, agora temos que injetá-lo nos callbacks das rotas que precisam desse tratamento. Faremos essas alterações no `routes/contatos.js`, de acordo com o seguinte código:

```
module.exports = function(app) {
 var autenticar = require('../middlewares/autenticador')
 , contatos = app.controllers.contatos;

 app.get('/contatos', autenticar, contatos.index);
 app.get('/contato/:id', autenticar, contatos.show);
 app.post('/contato', autenticar, contatos.create);
 app.get('/contato/:id/editar', autenticar, contatos.edit);
 app.put('/contato/:id', autenticar, contatos.update);
 app.del('/contato/:id', autenticar, contatos.destroy);
};
```

Basicamente, inserimos o callback `autenticar` antes da função principal da rota. Isso nos permitiu emular a execução de um filtro `before`. Caso queira criar um filtro `after`, não há

segredos: apenas coloque o callback do filtro por último. O importante é colocar as funções na ordem lógica de suas execuções.

## 5.5 INDO ALÉM: CRIANDO PÁGINAS DE ERROS AMIGÁVEIS

O Express oferece suporte para roteamento e renderização de erros do protocolo HTTP. Ele possui apenas duas funções: uma específica para tratamento do famoso erro **404** (página não encontrada), e uma função genérica que recebe por parâmetro uma variável contendo detalhes sobre o status e mensagem do erro HTTP.

### SOBRE O CÓDIGO DE ERROS DO HTTP

O protocolo HTTP tem diversos tipos de erros. O órgão W3C possui uma documentação explicando em detalhes o comportamento e código de cada erro. Para ficar por dentro desse assunto, veja em <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html> a especificação dos status de erro gerados por esse protocolo.

Que tal implementarmos um middleware de controle de erros para renderizar aos usuários páginas customizadas e mais amigáveis?

Primeiro, crie duas novas views, uma para apresentar a tela do erro 404, conhecida na web como **Página não encontrada**. Seu

nome será `views/not-found.ejs` .

```
<% include header %>
<header>
 <h1>Ntalk</h1>
 <h4>Infelizmente essa página não existe :(</h4>
</header>
<hr>
<p>Vamos voltar home page? :)</p>
<% include footer %>
```

A outra será focada em mostrar erros gerados pelo protocolo HTTP, com o nome de `views/server-error.ejs` :

```
<% include header %>
<header>
 <h1>Ntalk</h1>
 <h4>Aconteceu algo terrível! :(</h4>
</header>
<p>
 Veja os detalhes do erro:

 <%- error.message %>
</p>
<hr>
<p>Que tal voltar home page? :)</p>
<% include footer %>
```

Agora, vamos criar as funções de erros para um novo arquivo chamado `middlewares/error.js` , deixando-o da seguinte forma:

```
exports.notFound = function(req, res, next) {
 res.status(404);
 res.render('not-found');
};

exports.serverError = function(error, req, res, next) {
 res.status(500);
 res.render('server-error', {error: error});
};
```

Em seguida, modifique o *stack* de middlewares editando o

`app.js`. Repare que esse redirecionamento para página de erro deve ser adicionado por último, depois do carregamento das demais rotas e middlewares. Esta é uma regra Express 4, que exige que as rotas sejam carregadas primeiro e, por último, as rotas de erro:

```
var express = require('express')
, load = require('express-load')
, bodyParser = require('body-parser')
, cookieParser = require('cookie-parser')
, expressSession = require('express-session')
, methodOverride = require('method-override')
, error = require('./middlewares/error')
, app = express()
;
// ...carregamento dos middlewares...
load('models')
.then('controllers')
.then('routes')
.into(app)
;
// middleware de tratamento erros
app.use(error.notFound);
app.use(error.serverError);

app.listen(3000, function(){
 console.log("Ntalk no ar.");
});
```

Vamos testar o nosso código? Reinicie o servidor e, no browser, digite propositalmente uma nome de uma rota que não exista na aplicação, por exemplo, <http://localhost:3000/url-errada>. Esta foi a renderização da tela de erro para página não encontrada (erro 404).



Figura 5.3: Tela de erro 404

E para testar a página de erro interno do servidor? Esta tela somente será exibida em casos de erros graves no sistema. Para forçar um simples bug no sistema, remova um filtro da rota `/contatos`, forçando o bug que ocorria na seção anterior que falávamos sobre a implementação de filtros.



Figura 5.4: Tela de erro 500

Parabéns! Agora temos uma aplicação que cadastra, edita, exclui e lista contatos. Utilizamos o padrão de rotas REST, criamos um simples controle de login que mantém o usuário na sessão,

implementamos filtros para barrar acesso de usuários não autenticados e, para finalizar, implementamos a renderização de páginas de erros amigáveis – afinal, erros inesperados podem ocorrer em nossa aplicação, e seria desagradável o usuário ver informações complexas na tela.

## CAPÍTULO 6

# PROGRAMANDO SISTEMAS REAL-TIME

## 6.1 COMO FUNCIONA UMA CONEXÃO BIDIRECIONAL?

Este capítulo será muito interessante, pois falaremos sobre um assunto emergente nos sistemas atuais que está sendo largamente utilizado no Node.js. Estou falando sobre desenvolvimento de aplicações real-time.

Tecnicamente, estamos falando de uma conexão bidirecional, que, na prática, é uma conexão que se mantém aberta (*connection keep-alive*) para clientes e servidores interagirem em uma única conexão. A vantagem disso fica para os usuários, pois a interação no sistema será em tempo real, trazendo uma experiência de usuário muito melhor.

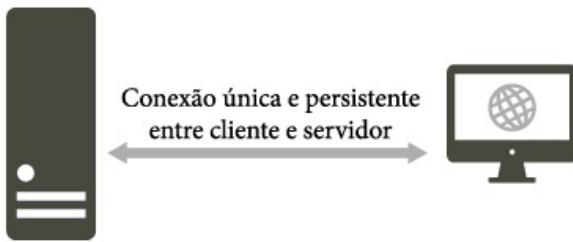


Figura 6.1: Imagem explicando sobre conexão bidirecional

O Node.js tornou-se popular por oferecer bibliotecas de baixo nível que suportam diversos protocolos (HTTP, HTTPS, FTP, DNS, TCP, UDP e outros). O recente protocolo *WebSockets* também é compatível com Node.js, e ele permite desenvolver sistemas de conexão persistente utilizando JavaScript tanto no cliente quanto no servidor.

O único problema em utilizar este protocolo é que nem todos os browsers suportam esse recurso, tornando inviável desenvolver uma aplicação real-time *cross-browser* apenas com *WebSockets*.

## 6.2 CONHECENDO O FRAMEWORK SOCKET.IO

Diante desse problema, nasceu o Socket.IO. Ele resolve a incompatibilidade entre o *WebSockets* com os navegadores antigos, emulando, por exemplo, *Ajax long-polling* ou outros *transports* de comunicação em browsers que não possuem *WebSockets*, de forma totalmente abstraída para o desenvolvedor. Seu site oficial é <http://socket.io>.



Figura 6.2: Framework Socket.IO

O Socket.IO funciona da seguinte maneira: é incluído no cliente um script que detecta informações sobre o browser do

usuário para definir qual será a melhor comunicação com o servidor. Os *transports* de comunicação que ele executa são:

1. *WebSocket*;
2. *Adobe Flash Socket*;
3. *AJAX long polling*;
4. *AJAX multipart streaming*;
5. *Forever iframe*;
6. *JSONP Polling*.

Se o navegador do usuário possuir compatibilidade com *WebSockets* ou *FlashSockets* (utilizando *Adobe Flash Player* do navegador), será realizada uma comunicação bidirecional. Caso contrário, será emulada uma comunicação unidirecional, que em curtos intervalos de tempo faz requisições AJAX no servidor. É claro que o desempenho é inferior, porém garante compatibilidade com browsers antigos e mantém o mínimo de experiência real-time para o usuário.

O mais interessante de tudo isso é que programar utilizando o *Socket.IO* é muito simples, e toda decisão complexa é ele que faz, simplificando a vida do desenvolvedor.

## 6.3 IMPLEMENTANDO UM CHAT REAL-TIME

Vamos ver como funciona na prática? Criaremos o web chat no Ntalk, com o qual o usuário enviará mensagens para os usuários online da agenda de contatos. Depois, integraremos o frameworks *Socket.IO* no Express.

Primeiro, vamos instalar esse módulo:

```
npm install socket.io --save
```

Agora temos de adaptar o `app.js` com esse novo módulo. A função `listen` do servidor web será realizada via módulo nativo do `http` através da função `server.listen(3000)`, para que o Socket.IO utilize o mesmo listener para criar seu ponto de comunicação através do protocolo HTTP. Afinal, o WebSockets é um protocolo que funciona em cima do HTTP ou HTTPS. Veja a seguir como que ficará essas modificações:

```
var express = require('express')
, load = require('express-load')
, bodyParser = require('body-parser')
, cookieParser = require('cookie-parser')
, expressSession = require('express-session')
, methodOverride = require('method-override')
, error = require('./middlewares/error')
, app = express()
, server = require('http').Server(app)
, io = require('socket.io')(server)
;
// carregamento dos middlewares...
// execução da função load()...

app.use(error.notFound);
app.use(error.serverError);

server.listen(3000, function(){
 console.log("Ntalk no ar.");
});
```

Dessa forma, habilitamos o Socket.IO em nossa aplicação, porém até agora nenhuma funcionalidade dele foi implementada. Para isso, vamos criar nosso primeiro evento de mensageria real-time no servidor através do callback da função: `io.sockets.on('connection')`. Ainda no `app.js`, inclua essa função depois da função `load()` com o seguinte código:

```
// carregamento dos módulos..
```

```
// carregamento dos middlewares...
// execução da função load()...

io.sockets.on('connection', function (client) {
 client.on('send-server', function (data) {
 var msg = "" + data.nome + " " + data.msg + "
";
 client.emit('send-client', msg);
 client.broadcast.emit('send-client', msg);
 });
});

// execução do server.listen() ...
```

Toda a brincadeira começa a partir do evento `io.sockets.on('connection')`. Essa função fica aguardando que um cliente envie alguma mensagem para o servidor por meio de algum evento. Qualquer nome de evento pode ser criado, exceto alguns nomes de eventos chaves que serão apresentados no final deste capítulo. Por enquanto, o único evento que foi criado é o `client.on('send-server')`, cuja sua execução ocorrerá quando um cliente enviar uma mensagem para o servidor.

Perceba que pouco código foi incluído e o servidor responde seus clientes através das funções `client.emit('send-client')` e `client.broadcast.emit('send-client')`. Resumindo, o fluxo básico é o cliente enviar uma mensagem para o servidor, e o servidor responde o próprio cliente (via `client.emit()`) e seus demais clientes conectados (via `client.broadcast.emit()`). Para compreender melhor, veja as figuras a seguir:

```
socket.emit("mensagem", "Olá!");
```

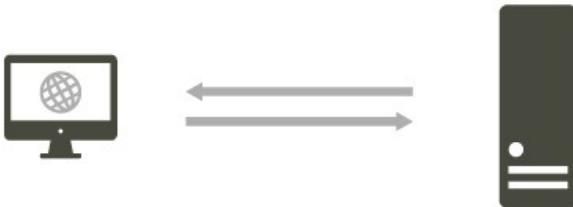


Figura 6.3: Envia mensagens para o cliente ou servidor

```
socket.broadcast.emit("mensagem", "Olá!");
```

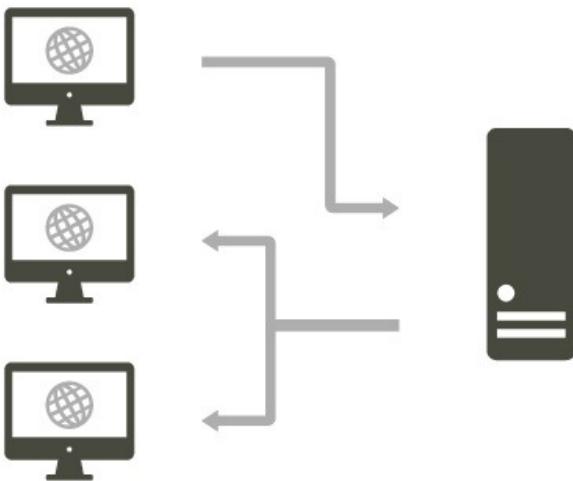


Figura 6.4: Envia mensagens para todos os clientes, exceto o próprio emissor

Com o back-end do Socket.IO implementado, vamos modificar o front-end para interagir neste meio de comunicação. Dentro de `views/contatos/index.ejs`, modificaremos a listagem dos contatos incluindo um link **Conversar** para acessar uma página de chat que vamos criar em seguida.

```
<% contatos.forEach(function(contato, index) { %>
```

```
<tr>
 <td><% contato.nome %></td>
 <td><% contato.email %></td>
 <td>
 <a href="/contato/<% index %>">Detalhes
 Conversar
 </td>
</tr>
<% }) %>
```

Agora, vamos implementar o layout do nosso chat e, principalmente, os eventos de comunicação do cliente para interagir com o servidor. Mas antes temos que criar seu controller em controllers/chat.js :

```
module.exports = function(app) {
 var ChatController = {
 index: function(req, res){
 var params = {usuario: req.session.usuario};
 res.render('chat/index', params);
 }
 };
 return ChatController;
};
```

Sua respectiva rota criando o arquivo routes/chat.js :

```
module.exports = function(app) {
 var autenticar = require('../middlewares/autenticador')
 , chat = app.controllers.chat;
 app.get('/chat', autenticar, chat.index);
};
```

Por último, crie sua view em views/chat/index.ejs . Usaremos JavaScript puro para manipulação de elementos do HTML, mas sinta-se à vontade para utiliar qualquer framework do gênero, como jQuery (<http://jquery.com>) ou ZeptoJS (<http://zeptojs.com>).

O importante é carregar o script /socket.io/socket.io.js

para que a brincadeira comece. Aliás, não se preocupe de onde vêm esse script, pois ele é distribuído automaticamente pelo framework `socket.io-client`. Este já vem como dependência do Socket.IO quando ele é instalado, e é responsável pela comunicação do cliente com o servidor.

É pela função `io('dominio-do-servidor')` que o cliente se conecta com o servidor e começa a trocar mensagens com ele. Essa interação ocorre através do tráfego de objetos JSON. Vamos incluir na view `views/chat/index.ejs` as funções de enviar e receber mensagens:

```
<% include ../header %>
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io('http://localhost:3000');
 socket.on('send-client', function (msg) {
 document.getElementById('chat').innerHTML += msg;
 });
 var enviar = function() {
 var nome = document.getElementById('nome').value;
 var msg = document.getElementById('msg').value;
 socket.emit('send-server', {nome: nome, msg: msg});
 };
</script>
<header>
 <h2>Ntalk - Chat</h2>
</header>
<section>
 <pre id="chat"></pre>
 <input type="hidden" id="nome" value="<%- usuario.nome %>">
 <input type="text" id="msg" placeholder="Mensagem">
 <button onclick="enviar();">Enviar</button>
</section>
<% include ../exit %>
<% include ../footer %>
```

Em seguida, vamos testar nossa implementação. Para isso, reinicie a aplicação e, depois, acesse novamente no endereço

`http://localhost:3000` pelo browser.

Para entender melhor como tudo funciona, acesse o mesmo endereço em um outro browser para ter duas janelas da mesma aplicação. **Atenção: abrir duas abas de um mesmo browser não funcionará por causa do cookie!**

Depois, cadastre dois usuários, cada um em sua respectiva janela, e adicione em contatos o nome e e-mail do usuário da outra janela (por exemplo, na janela do **Usuário A**, adicione os dados do **Usuário B** e vice-versa). Em seguida, escolha uma das janelas, clique no link **Conversar** e envie uma mensagem para o outro usuário.

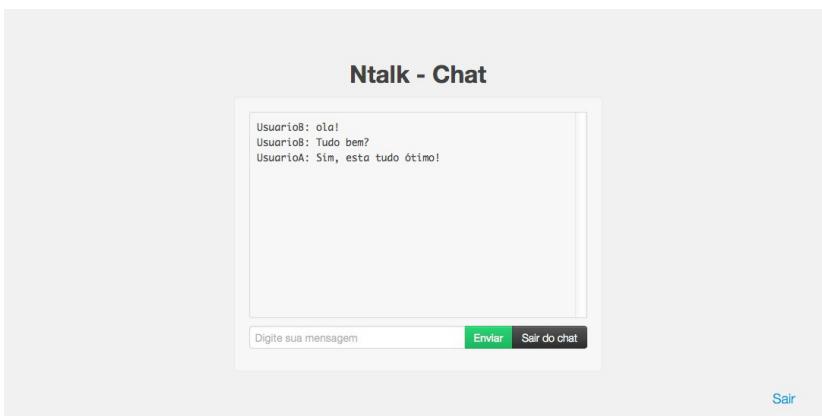


Figura 6.5: Usuário A conversando com Usuário B

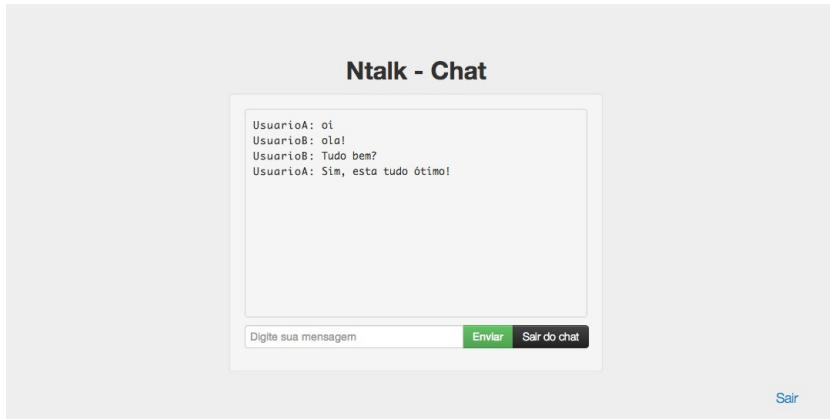


Figura 6.6: Usuário B respondendo mensagens do Usuário A

Se tudo deu certo, parabéns! O web chat com o mínimo de recursos funcionais e em tempo real está no ar. Mesmo que você use um navegador antigo, como o Internet Explorer 6, tudo funcionará bem! Isso porque o Socket.IO fará o trabalho de emular a comunicação entre ambos, mesmo sem a presença do protocolo WebSockets.

## 6.4 ORGANIZANDO O CARREGAMENTO DE SOCKETS

A nossa aplicação terá duas funcionalidades com resposta em tempo real: o chat e o notificador de mensagens. Em sistemas maiores, podem surgir diversas funcionalidades nessa modalidade, e codificar todas as funções do Socket.IO em um único arquivo ficaria terrivelmente assustador para fazer manutenções. O ideal é separar as funcionalidades de forma modular e escalável, sendo essa prática semelhante ao que fizemos com as *views*, *models* e *controllers*.

Com base nesse problema, crie o diretório `sockets` e adicione nele o arquivo `chat.js`. Para esse arquivo, vamos migrar o código Socket.IO do `app.js`. Veja o exemplo que segue:

```
module.exports = function(io) {
 var sockets = io.sockets;
 sockets.on('connection', function (client) {
 client.on('send-server', function (data) {
 var msg = "" + data.nome + " " + data.msg + "
";
 client.emit('send-client', msg);
 client.broadcast.emit('send-client', msg);
 });
 });
}
```

Dessa forma, deixamos o `app.js` com menos código. Um detalhe importante é que o `chat` será carregado por uma nova chamada à função `load()` para a qual passaremos como parâmetro a variável `io`:

```
// carregamento dos módulos..
// carregamento dos middlewares...

load('models')
 .then('controllers')
 .then('routes')
 .into(app);

load('sockets')
 .into(io);

// server.listen()...
```

Mais uma vez utilizamos boas práticas de código modular, organizando melhor o projeto e preparando-o para trabalhar com diversas funções do Socket.IO em conjunto com Express.

## 6.5 SOCKET.IO E EXPRESS USANDO A MESMA

## SESSÃO

O Socket.IO consegue acessar e manipular uma session criada pelo servidor web. Implementaremos esse controle de sessão compartilhada, pois será mais seguro do que passar os dados do usuário logado por meio da tag:

```
<input type="hidden" id="nome" value="<%- usuario.nome %>">
```

Como isso funciona? Na prática, quando logamos no sistema, o Express cria um ID de sessão para o usuário. Essa sessão é persistida em memória ou disco no servidor (essa decisão fica a critério dos desenvolvedores). O Socket.IO não consegue acessar esses dados, ele apenas possui um controle para autorizar uma conexão do cliente. Com isso, podemos usar as funções de sessão e cookie do Express dentro dessa função de autorização para validar uma sessão — se essa for uma válida, armazenaremos no cliente Socket.IO, autorizando sua conexão no sistema.

Resumindo, precisamos criar um controle para compartilhar sessão entre o Express e Socket.IO. Vamos configurar no Express para isolar em variáveis as funções `cookieParser()` e `expressSession()`. Também criaremos duas constantes chamadas `KEY` e `SECRET`, que serão utilizadas para buscar o ID da sessão e carregar os dados do usuário logado usando o objeto `expressSession.MemoryStore` que será reutilizado na autenticação do Socket.IO. Faremos essas modificações no `app.js`, seguindo o trecho do código:

```
const KEY = 'ntalk.sid', SECRET = 'ntalk';
var express = require('express')
, load = require('express-load')
, bodyParser = require('body-parser')
, cookieParser = require('cookie-parser')
```

```

, expressSession = require('express-session')
, methodOverride = require('method-override')
, error = require('./middlewares/error')
, app = express()
, server = require('http').Server(app)
, io = require('socket.io')(server)
, cookie = cookieParser(SECRET)
, store = new expressSession.MemoryStore()
;

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(cookie);
app.use(expressSession({
 secret: SECRET,
 name: KEY,
 resave: true,
 saveUninitialized: true,
 store: store
}));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
app.use(methodOverride('_method'));
app.use(express.static(__dirname + '/public'));

// continuação do app.js ...

```

Com esses recursos habilitados, será possível utilizar a sessão do Express no Socket.IO. Mas, para isso, vamos implementar um controle de autorização via função `io.use(callback)` para que, a cada conexão, o Socket.IO valide o `sessionID` permitindo ou não recuperar os dados do usuário presente no sistema. A seguir, implementamos diversas condicionais para essa validação no `app.js`:

```

// carregamento dos módulos..
// carregamento dos middlewares...

io.use(function(socket, next) {
 var data = socket.request;
 cookie(data, {}, function(err) {
 var sessionID = data.signedCookies[KEY];

```

```
store.get(sessionID, function(err, session) {
 if (err || !session) {
 return next(new Error('acesso negado'));
 } else {
 socket.handshake.session = session;
 return next();
 }
});
});
});

// load()...
// server.listen()...
```

A função `next()` é responsável pela autorização ou não autorização da conexão, e a variável `socket.request` contém informações da requisição cliente, ou seja, isso inclui *headers*, *cookies* e outras informações do HTTP. Buscamos o `sessionID` através da variável `data.signedCookies[KEY]` e, em seguida, buscamos os dados da sessão que estão na memória do servidor através da função `store.get(sessionID)`. Se obtemos sucesso, incluímos a sessão na variável `socket.handshake.session` e liberamos a conexão pela função `return next()`. Caso contrário, executamos a função `next()` passando como parâmetro um objeto de erro, com uma mensagem de erro.

Pronto! Agora o Socket.IO está habilitado para ler e manipular os objetos de uma sessão criada pelo Express. Com isso, podemos trafegar dados do usuário logado dentro do nosso chat de forma segura pelo back-end da aplicação. Para finalizar essa tarefa, faremos alguns *refactorings*.

Primeiro, vamos eliminar a tag `<input type="hidden" id="nome" value="<%- usuario.nome %>">` e, em seguida, modificar a função `enviar()` para que ela somente envie o conteúdo da mensagem. Para isso, abra e edite o código

views/chat/index.ejs :

```
<% include ../header %>
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io('http://localhost:3000');
 socket.on('send-client', function (msg) {
 var chat = document.getElementById('chat');
 chat.innerHTML += msg;
 });
 var enviar = function() {
 var msg = document.getElementById('msg');
 socket.emit('send-server', msg.value);
 };
</script>
<header>
 <h2>Ntalk - Chat</h2>
</header>
<section>
 <pre id="chat"></pre>
 <input type="text" id="msg" placeholder="Mensagem">
 <input type="button" onclick="enviar();" value="Enviar">
</section>
<% include ../exit %>
<% include ../footer %>
```

Também removeremos do controllers/chat.js a variável referente aos dados da sessão do usuário:

```
module.exports = function(app) {
 var ChatController = {
 index: function(req, res){
 res.render('chat/index');
 }
 };
 return ChatController;
};
```

Com a *view* e *controller* atualizados, vamos adaptar no sockets/chat.js dentro do evento sockets.on('connection') a mensagem que será emitida pelo servidor, concatenando o nome e conteúdo da mensagem do

usuário. A única diferença é que agora serão carregados pela variável `client.handshake.session.usuario` que mantém dados da sessão:

```
module.exports = function(io) {
 var sockets = io.sockets;
 sockets.on('connection', function (client) {
 var session = client.handshake.session
 , usuario = session.usuario;
 client.on('send-server', function (msg) {
 msg = "" + usuario.nome + ": " + msg + "
";
 client.emit('send-client', msg);
 client.broadcast.emit('send-client', msg);
 });
 });
}
```

Que tal testar essas modificações? Faça o seguinte: reinicie o servidore acesse o browser em `http://localhost:3000`. **Antes de logar no sistema, apague os cookies do browser**, pois foram configurados uma nova chave do cookie e sessão da aplicação. Após esse procedimento, teste o seu chat iniciando uma nova conversa entre dois usuários e veja se tudo esta funcionando corretamente.

## 6.6 GERENCIANDO SALAS DO CHAT

Para finalizar o nosso chat, vamos aprimorá-lo implementando um controle de sala *one-to-one* para assegurar que cada conversa seja entre dois usuários no nosso bate-papo, prevenindo que outros usuários entrem no meio de uma conversa a dois.

Desenvolver essa funcionalidade é muito simples: apenas temos que criar uma string que será o nome da sala e utilizá-la através da função `sockets.in('nome_da_sala').emit()`. Outro

detalhe dessa função é que ela emite um evento para todos os usuários da sala, incluindo o próprio usuário emissor.

Para implementar uma sala *one-to-one*, temos de garantir que em uma sala entrem, no máximo, dois clientes. Para implementar de forma segura, criaremos nomes de salas difíceis de serem decifrados, usando um módulo nativo do Node.js para criptografar o nome de uma sala. Ele será um *Hash MD5* de um *timestamp*, criado pelo primeiro usuário que começar uma conversa.

#### DICAS SOBRE CRIPTOGRAFIA NO NODE.JS

Cada nova versão da API Crypto do Node.js traz novas melhorias e novas funções. Porém, essa API trabalha com recursos básicos de criptografia. Caso você seja um desenvolvedor focado em segurança, utilize o módulo BCrypt, que possui uma série de funcionalidades e algoritmos de criptografia mais difíceis de se quebrar.

Para conhecer o BCrypt em detalhes e suas funcionalidades acesse sua documentação em <https://github.com/ncb000gt/node.bcrypt.js>.

Veja na prática como gerenciaremos as salas dos usuários. Primeiro, carregaremos os módulos de criptografia para gerar o valor *hash* da sala.

```
module.exports = function(io) {
 var crypto = require('crypto')
 , sockets = io.sockets;
 // continuação do sockets/chat.js ...
```

```
}
```

Dentro do evento `sockets.on('connection')` , criaremos um novo evento `client.on('join')` , que será emitido quando um usuário entrar na tela do chat. Implementaremos um simples condicional para criar o *hash* da sala quando não existir e, em seguida, armazenamos na sessão do usuário via `session.sala = sala` .

```
client.on('join', function(sala) {
 if(!sala) {
 var timestamp = new Date().toString()
 , md5 = crypto.createHash('md5');
 sala = md5.update(timestamp).digest('hex');
 }
 session.sala = sala;
 client.join(sala);
});
```

Para controlar a saída de usuários na sala, vamos utilizar o evento *default* do Socket.IO chamado de `client.on('disconnect')` , que será utilizado para excluir um usuário da sala quando o mesmo se desconectar dela. Para descobrirmos em qual sala o usuário está conectado, usaremos a variável `session.sala = sala` para recuperar o ID da sala e, em seu callback, vamos remover o usuário através da função `client.leave(session.sala)` .

```
client.on('disconnect', function () {
 client.leave(session.sala);
});
```

Para finalizar, vamos atualizar o evento `client.on('send-server')` , para que ele envie uma mensagem somente para usuários de uma sala através da função `sockets.in(sala).emit('send-client', msg)` . Também

vamos criar um novo evento chamado `client.broadcast.emit('new-message', data);`. Sua variável `data` terá como parâmetros o e-mail e o ID da sala do usuário, e ele será executado para atualizar a URL do botão **Conversar** do contato que receber uma mensagem.

```
client.on('send-server', function (msg) {
 var sala = session.sala
 , data = {email: usuario.email, sala: sala};
 msg = "" + usuario.nome + " " + msg + "
";
 client.broadcast.emit('new-message', data);
 sockets.in(sala).emit('send-client', msg);
});
```

Para finalizar nosso back-end, edite o `controllers/chat.js` para que ele envie uma query string do ID da sala quando existir.

```
module.exports = function(app) {
 var ChatController = {
 index: function(req, res){
 var params = {sala: req.query.sala};
 res.render('chat/index', params);
 }
 };
 return ChatController;
};
```

Com o nosso back-end implementado, resta-nos preparar o front-end. Primeiro, temos de atualizar o `views/chat/index.ejs` para que, quando um usuário entrar nessa tela, seja automaticamente dispare o evento `socket.emit('join', '<%- sala %>')` para criar uma nova sala no Socket.IO.

```
<% include ../header %>
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io('http://localhost:3000');
 socket.emit('join', '<%- sala %>');
```

```

socket.on('send-client', function (msg) {
 document.getElementById('chat').innerHTML += msg;
});
var enviar = function() {
 var msg = document.getElementById('msg');
 socket.emit('send-server', msg.value);
};
</script>
<header>
 <h2>Ntalk - Chat</h2>
</header>
<section>
 <pre id="chat"></pre>
 <input type="text" id="msg" placeholder="Mensagem">
 <button onclick="enviar();">Enviar</button>
</section>
<% include ../exit %>
<% include ../footer %>

```

Agora vamos incluir no botão **Conversar** um meio de ele atualizar o *hash* de sua URL que levará um usuário para uma sala do chat quando ele for chamado por um de seus contatos. Com isso, vamos criar um novo ponto de conexão do Socket.IO. Este ponto será implementado em `views/contatos/index.ejs`. Crie um novo *partial* chamado `views/contatos/notify_script.ejs` com o seguinte código:

```

<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io('http://localhost:3000');
 socket.on('new-message', function(data) {
 var chat = document.getElementById('chat_' + data.email);
 chat.href += '?sala=' + data.sala;
 });
</script>

```

Agora em `views/contatos/index.ejs`, inclua o *partial* anterior através da tag `<% include notify_script %>`. Também inclua no botão **Conversar** um ID para o script anterior identificar onde será atualizada a URL com o *hash* de uma sala.

Veja:

```
<% include ../header %>
<header>
 <h2>Ntalk - Agenda de contatos</h2>
</header>
<section>
 <!-- Formulário de novo contato -->
 <table>
 <thead>
 <tr>
 <th>Nome</th>
 <th>E-mail</th>
 <th>Ação</th>
 </tr>
 </thead>
 <tbody>
 <% contatos.forEach(function(contato, index) { %>
 <tr>
 <td><%= contato.nome %></td>
 <td><%= contato.email %></td>
 <td>
 <a href="/contato/<%= index %>">Detalhes
 <a href="/chat" id="chat_<%= contato.email %>">
 Conversar

 </td>
 </tr>
 <% }) %>
 </tbody>
 </table>
</section>
<% include notify_script %>
<% include ../exit %>
<% include ../footer %>
```

Reinic peace o servidor e faça o teste! Se tudo der certo, o chat *one-to-one* estará funcionando perfeitamente.

## 6.7 NOTIFICADORES NA AGENDA DE CONTATOS

Para finalizar este capítulo com chave de ouro, vamos criar um simples notificador na agenda de contatos. Esse notificador vai informar o status de cada contato, que terá apenas três estados: **Online**, **Offline** e **Mensagem**. Visualmente, ele será um novo campo na tabela de contatos, e vamos explorar novas funções do Socket.IO para torná-lo real-time.

Abra o código `sockets/chat.js` e implemente no início do evento `sockets.on('connection')` uma nova regra que seguirá a seguinte lógica: armazenar o e-mail dos usuários online e, depois, rodar um loop desses e-mails para que em cada iteração seja executado as funções `client.emit('notify-onlines', email)` e `client.broadcast.emit('notify-onlines', email)`. Isso notificará que esse usuário está on-line para os outros usuários que o possuírem em sua agenda de contatos. Veja como faremos isso no código a seguir:

```
module.exports = function(io) {
 var crypto = require('crypto')
 , sockets = io.sockets
 , onlines = {}
 ;
 sockets.on('connection', function (client) {
 var session = client.handshake.session
 , usuario = session.usuario
 ;

 onlines[usuario.email] = usuario.email;
 for (var email in onlines) {
 client.emit('notify-onlines', email);
 client.broadcast.emit('notify-onlines', email);
 }

 // continuação do chat.js ...
 });
}
```

Já temos um notificador de usuários online. Agora, precisamos incrementar esse notificador para que ele informe quando um usuário está **offline**, ou quando está enviando uma **mensagem** em sua agenda. Não há segredo para implementar essas regras, apenas temos de reutilizar a função `client.broadcast.emit('new-message')` para atualizar o status de **mensagem** no usuário, e implementar a função `client.broadcast.emit('notify-offlines')` dentro do evento `client.on('disconnect')` que garantirá que um usuário saiu do chat. Por último, removemos o e-mail do usuário da variável `onlines` (através do código `delete onlines[usuario.email]`) para garantir que, na próxima iteração, ele não esteja mais online.

```
client.on('send-server', function (msg) {
 var sala = session.sala
 , data = {email: usuario.email, sala: sala};
 msg = "" + usuario.nome + " " + msg + "
";
 client.broadcast.emit('new-message', data);
 sockets.in(sala).emit('send-client', msg);
});

client.on('disconnect', function() {
 var sala = session.sala
 , msg = "" + usuario.nome + ": saiu.
";
 client.broadcast.emit('notify-offlines', usuario.email);
 sockets.in(sala).emit('send-client', msg);
 delete onlines[usuario.email];
 client.leave(sala);
});
```

Com o back-end desenvolvido, vamos finalizar essa tarefa codificando como serão renderizados os status de cada contato na agenda. Para fazer essas modificações, edite o arquivo `views/contatos/notify_script.ejs`:

```
<script src="/socket.io/socket.io.js"></script>
<script>
```

```

var socket = io('http://localhost:3000');
var notify = function(data) {
 var id = 'notify_' + data.email;
 var notify = document.getElementById(id);
 if (notify) {
 notify.textContent = data.msg;
 }
};
socket.on('notify-onlines', function(email) {
 notify({email: email, msg: 'Online'});
});
socket.on('notify-offlines', function(email) {
 notify({email: email, msg: 'Offline'});
});
socket.on('new-message', function(data) {
 notify({email: data.email, msg: 'Mensagem'});
 var id = 'chat_' + data.email;
 var chat = document.getElementById(id);
 chat.href += '?sala=' + data.sala;
});
</script>

```

Com isso implementado, basta atualizar a lista de contatos para visualizar as mensagens de notificação. Edite o arquivo `views/contatos/index.ejs` incluindo uma tag `<span id="notify_<%- contato.email %>">` para que o código JavaScript anterior faça toda magia!

```

<table>
 <thead>
 <tr>
 <th>Nome</th>
 <th>E-mail</th>
 <th>Status</th>
 <th>Ação</th>
 </tr>
 </thead>
 <tbody>
 <% contatos.forEach(function(contato, index) { %>
 <tr>
 <td><%- contato.nome %></td>
 <td><%- contato.email %></td>

```

```
<td>
 <span id="notify_<%- contato.email %>">Offline
</td>
<td>
 <a href="/contato/<%- index %>">Detalhes
 <a href="/chat" id="chat_<%- contato.email %>">
 Conversar

</td>
</tr>
<% }) %>
</tbody>
</table>
```

Agora temos o nosso notificador pronto! Para testá-lo, reinicie o servidor, crie 3 contas no Ntalk cadastrando os e-mails de cada conta como contato entre elas, para possibilitar uma conversa no chat. Por exemplo, cadastro da conta A, B, C e os contatos da conta A são os usuários da conta B e C, e assim faça o mesmo com as demais para criar uma rede de contatos.

Depois disso, converse no chat entre a conta A com a B, e repare que agora os status vão se alterar em tempo real de acordo com a interação do usuário.

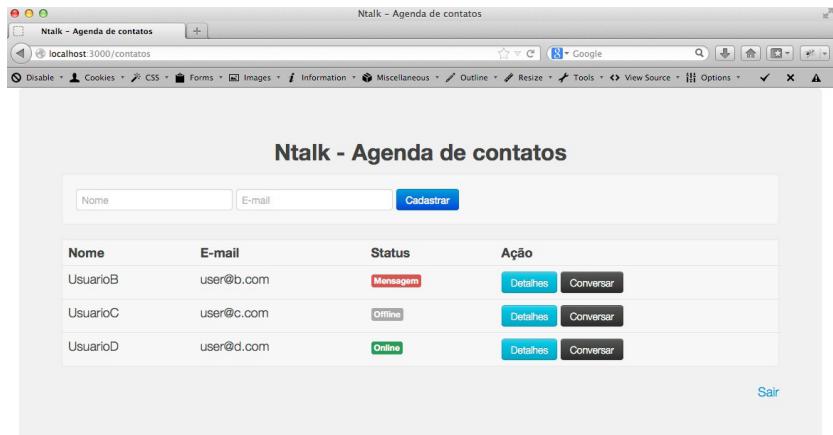


Figura 6.7: Notificações da agenda de contatos do Usuário A

## 6.8 PRINCIPAIS EVENTOS DO SOCKET.IO

Para complementar seus estudos com este módulo, apresentarei os principais eventos do *Socket.IO*, tanto no servidor como para cliente.

No lado do servidor:

- `io.sockets.on('connection', function(client))` – evento que acontece quando um novo cliente se conecta no servidor.
- `client.on('message', function(mensagem, callback))` – ocorre quando um cliente se comunica através da função `send()`. O callback desse evento responde automaticamente o cliente no final de sua execução.
- `client.on('qualquer-nome-de-evento', function(data))` – são eventos criados pelo desenvolvedor; qualquer nome pode ser apelidado

aqui, exceto os nomes dos eventos principais e o seu comportamento é de apenas receber objetos através da variável `data`. Em nosso chat, criamos o evento '`send-server`'.

- `client.on('disconnect', callback)` – quando um cliente sai do sistema é emitido o evento '`disconnect`' para o servidor. Também é possível emitir esse evento no cliente sem precisar sair do sistema.

No lado do cliente:

- `client.on('connect', callback)` – ocorre quando o cliente se conecta no servidor.
- `client.on('connecting', callback)` – ocorre quando o cliente está se conectando no servidor.
- `client.on('disconnect', callback)` – ocorre quando o cliente se desconecta do servidor.
- `client.on('connect_failed', callback)` – ocorre quando o cliente não conseguiu se conectar no servidor devido a falhas de comunicação entre cliente com servidor.
- `client.on('error', callback)` – ocorre quando o cliente já se conectou, porém um erro no servidor ocorreu durante as trocas de mensagens.
- `client.on('message', function(message, callback))` – ocorre quando o cliente envia uma mensagem de resposta rápida ao servidor, cujo o retorno acontece através da função de callback.
- `client.on('qualquer-nome-de-evento', function(data))` – evento customizado pelo

desenvolvedor. No exemplo do web chat, criamos o evento 'send-client' , que envia mensagem para o servidor.

- `client.on('reconnect_failed', callback)` – ocorre quando o cliente não consegue se reconectar no servidor.
- `client.on('reconnect', callback)` – ocorre quando o cliente se reconecta ao servidor.
- `client.on('reconnecting', callback)` – ocorre quando o cliente está se reconectando no servidor.

Mais uma vez, implementamos uma incrível funcionalidade em nosso sistema. No próximo capítulo, vamos otimizar a agenda de contatos adicionando um banco de dados para persistir os contatos dos usuários, e também incluiremos um histórico de conversas no chat.

## CAPÍTULO 7

# INTEGRAÇÃO COM BANCO DE DADOS

## 7.1 BANCOS DE DADOS MAIS ADAPTADOS PARA NODE.JS

Nos capítulos anteriores (em especial, os capítulos *Iniciando com o Express*, *Dominando o Express* e *Programando sistemas real-time*), aplicamos um modelo simples de banco de dados, mais conhecido como *MemoryStore*. Ele não é um modelo adequado de persistência de dados, pois quando o usuário sair da aplicação ou o servidor for reiniciado, todos os dados serão apagados. Utilizamos esse modelo apenas para apresentar os conceitos sobre os módulos Express e Socket.IO.

Neste capítulo, vamos aprofundar nossos conhecimentos trabalhando com um banco de dados de verdade para Node.js. Algo fortemente ligado ao Node.js são os banco de dados NoSQL. É claro que existem módulos de banco de dados SQL, mas de fato, módulos NoSQL são mais populares nesta plataforma, visto que alguns bancos de dados permitem armazenar objetos do tipo JSON, que facilita (e muito) manipular estes objetos no Node.js.

A grande vantagem de trabalhar com esse modelo de banco de

dados é a grande compatibilidade e suporte mantido pela comunidade própria Node.js.

Os NoSQL populares são:

- MongoDB – <http://www.mongodb.org>
- Redis – <http://redis.io/>
- CouchDB – <http://couchdb.apache.org>
- RiakJS – <http://riakjs.com>

Dos bancos de dados SQL, existem alguns módulos para MySQL (<http://www.mysql.com>), SQLite (<http://www.sqlite.org>) e PostgreSQL (<http://www.postgresql.org>).

Caso queira ver todos os *drivers* compatíveis com Node.js, acesse <https://github.com/joyent/node/wiki/Modules#wiki-database>.



Figura 7.1: NoSQL MongoDB

Neste livro, usaremos o MongoDB, que é um banco de dados NoSQL, mantido pela empresa 10gen e foi escrito em linguagem C/C++. Ele utiliza JavaScript como interface para manipulação de dados, e a persistência dos dados é feita através de objetos JSON, em que no MongoDB é conceitualmente chamado de documentos.

Nele, trabalhamos com o conceito *schema-less*, ou seja, não existe relacionamentos de tabelas, nem chaves primárias ou estrangeiras, mas sim documentos que possuem (ou não) documentos embutidos – que são subdocumentos dentro de um mesmo documento –, sendo tudo isso mantido dentro de uma *collection* (coleção de documentos, que é o mesmo que chamar de tabelas no modelo relacional).

Outra vantagem do *schema-less* é que os atributos são inseridos ou removidos em *runtime*, sem a necessidade de travar o banco de dados quando um registro inválido for inserido. Isso faz com que o MongoDB seja flexível a grandes mudanças.

Como disse antes, com o MongoDB podemos persistir subdocumentos dentro de um documento, que seria o mesmo que criar relacionamento entre tabelas. Porém, neste conceito, tudo é inserido em um mesmo documento. Isso diminui e muito o número de consultas complexas no banco de dados e, principalmente, elimina consultas que usam *joins* entre documentos.

## 7.2 INSTALANDO O MONGODB

Não há segredos para instalar o MongoDB, tanto é que utilizaremos suas configurações padrões. Primeiro, acesse <http://www.mongodb.org/downloads> e faça o download do MongoDB compatível com seu sistema operacional.

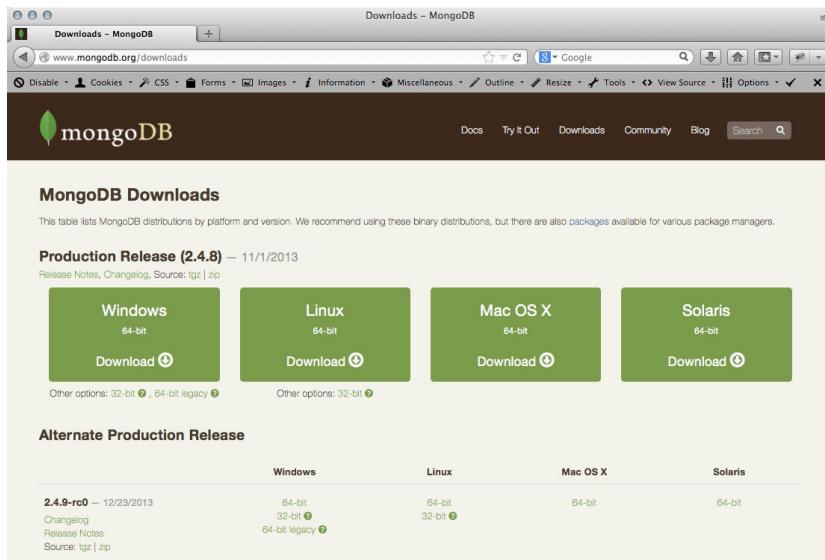


Figura 7.2: Página de download do MongoDB

Neste livro, usaremos sua última versão estável, a **2.4.8**. O MongoDB será instalado via HomeBrew (<http://mxcl.github.com/homebrew>), que é um gerenciador de pacotes para Mac. Para instalá-lo, execute os comandos:

```
brew update
brew install mongodb
```

Se você estiver no Linux Ubuntu, terá um pouco mais de trabalho; porém, é possível instalá-lo via comando `apt-get`. Primeiro, em modo `sudo` (super usuário), edite o arquivo `/etc/apt/sources.list.d/mongodb.list`, adicionando no final do arquivo o comando:

```
deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
dist 10gen
```

Salve-o e execute os próximos comandos:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
 --recv 7F0CEB10
sudo apt-get update
sudo apt-get install mongodb-10gen
```

Já no Windows, o processo é tão mais trabalhoso que nem apresentarei neste livro... brincadeira! Instalar no Windows também é fácil, apenas baixe o MongoDB, crie as pastas C:\data\db e C:\mongodb , e descompacte o conteúdo do arquivo zip do MongoDB dentro de C:\mongodb .

## 7.3 MONGODB NO NODE.JS UTILIZANDO MONGOOSE

O Mongoose possui uma interface muito fácil de aprender. Em poucos códigos, você vai conseguir criar uma conexão no banco de dados e executar uma query, ou persistir dados. Com o MongoDB instalado e funcionando em sua máquina, vamos instalar o módulo mongoose , um framework responsável por mapear objetos do Node.js para MongoDB. Para isso, execute o comando:

```
npm install mongoose --save
```

Para a aplicação se conectar com o banco de dados, no app.js , usaremos a variável db em modo global para manter uma conexão com o banco de dados compartilhando seus recursos em todo projeto:

```
const KEY = 'ntalk.sid', SECRET = 'ntalk';
var express = require('express')
, load = require('express-load')
, bodyParser = require('body-parser')
, cookieParser = require('cookie-parser')
, expressSession = require('express-session')
, methodOverride = require('method-override')
, error = require('./middlewares/error')
```

```
, app = express()
, server = require('http').Server(app)
, io = require('socket.io')(server)
, cookie = cookieParser(SECRET)
, store = new expressSession.MemoryStore()
, mongoose = require('mongoose')
;
global.db = mongoose.connect('mongodb://localhost:27017/ntalk')
;
```

Quando é executada a função `mongoose.connect` , cria-se uma conexão com o banco de dados MongoDB para o Node.js. Como o MongoDB é *schema-less*, na primeira vez que a aplicação se conecta com o banco através da URL `'mongodb://localhost:27017/ntalk'` , automaticamente será criada a base de dados com o nome **ntalk**. Caso queira conferir, abra o terminal e acesse o CLI do MongoDB:

```
mongo
```

Em seguida, dentro do console MongoDB, execute o comando `show dbs` e veja se a base de dados foi criada semelhante à seguinte figura:

```
[vagrant:ntalk] $ mongo
MongoDB shell version: 2.4.9
connecting to: test
> show dbs
local 0.078125GB
ntalk 0.203125GB ←
test (empty)
> █
```

Figura 7.3: Base do Ntalk no MongoDB

## 7.4 MODELANDO COM MONGOOSE

O Mongoose é um módulo focado para a criação de *models*; isso significa que, com ele, criaremos objetos persistentes modelando seus atributos através do objeto `mongoose.Schema`. Após a implementação de um modelo, temos de registrá-lo no banco de dados utilizando a função `db.model('nome-do-model', modelSchema)`, que recebe um modelo e cria sua respectiva collection no MongoDB.

Vamos explorar as principais funcionalidades do Mongoose aplicando-o na prática em nosso projeto. Com isso, faremos diversos *refactorings* em toda aplicação para substituir o modelo de persistência de sessão para o modelo do Mongoose que armazena dados no MongoDB.

Para começar, vamos criar o modelo `models/usuario.js`. Ele será o modelo principal e terá os seguintes atributos: `nome`,

`email` e `contatos`. A modelagem dos atributos acontece através do objeto `require('mongoose').Schema`. Veja a seguir como ficará esta modelagem:

```
module.exports = function(app) {
 var Schema = require('mongoose').Schema;

 var contato = Schema({
 nome: String
 , email: String
 });
 var usuario = Schema({
 nome: {type: String, required: true}
 , email: {type: String, required: true
 , index: {unique: true}}
 , contatos: [contato]
 });

 return db.model('usuarios', usuario);
};
```

Rpare que foram criados dois objetos: `usuario` e `contato`, e apenas o modelo `usuario` foi registrado como *collection* para o MongoDB. Faremos isso, pois o `contato` será um subdocumento de `usuario` e o registro ocorre via função `db.model('usuarios', usuario)`.

Outro detalhe importante é que incluímos dois tipos de validações neste modelo: `required` e `unique`. Para quem não conhece, o `required: true` valida se o seu atributo possui algum valor, ou seja, ele não permite persistir um campo vazio. Já o `unique: true` cria um índice de valor único em seu atributo, semelhante nos bancos de dados SQL. Essas validações geram um erro que é enviado no callback de qualquer função de persistência do modelo, por exemplo, `usuario.create()` ou `usuario.update()`.

## 7.5 IMPLEMENTANDO UM CRUD NA AGENDA DE CONTATOS

Com o modelo implementado, o que nos resta a fazer é alterar os controllers para usarem suas funções. Começando do mais fácil, vamos modificar o `controllers/home.js`. Nele, vamos executar a função `findOne`, que retorna apenas um objeto, e a função `select('name email')` para filtrar esse objeto retornando um novo, contendo apenas os atributos `name` e `email` do usuário. Com isso, evita-se que seja carregado o subdocumento `contatos` ao efetuar o login no sistema.

Na prática, essa query seria algo que, em banco de dados SQL, faríamos com o seguinte comando:

```
SELECT name, email FROM usuarios LIMIT 1
```

Caso não seja encontrado um usuário, cadastraremos um novo através da função `Usuario.create`. Veja o código-fonte:

```
module.exports = function(app) {
 var Usuario = app.models.usuario;

 var HomeController = {
 index: function(req, res) {
 res.render('home/index');
 },
 login: function(req, res) {
 var query = {email: req.body.usuario.email};
 Usuario.findOne(query)
 .select('nome email')
 .exec(function(error, usuario){
 if (usuario) {
 req.session.usuario = usuario;
 res.redirect('/contatos');
 } else {
 var usuario = req.body.usuario;
 Usuario.create(usuario, function(error, usuario){
```

```

 if(error){
 res.redirect('/');
 } else {
 req.session.usuario = usuario;
 res.redirect('/contatos');
 }
 });
}
);
},
logout: function(req, res) {
 req.session.destroy();
 res.redirect('/');
}
};

return HomeController;
};

```

Repare que, com o *Mongoose*, é possível criar queries complexas chamando funções que em seu retorno permitem chamar uma outra função. Isso forma um encadeamento de funções, cujo fim ocorre quando chamamos a função `exec()`. Um bom exemplo disso é a query a seguir:

```

Usuario.findOne(req.body.usuario)
 .select('nome email')
 .exec(function(error, usuario) {
 // continuação do código...
 });

```

A função `Usuario.create` persiste um objeto que tenha os mesmo atributos de seu modelo. Caso contrário, ocorrerá um erro e os seus detalhes serão enviados para variável `erro` presente no callback. Por esse motivo, é recomendável sempre tratar esses erros para garantir integridade dos dados no sistema.

Parabéns! Com o controle de login funcionando corretamente, só falta modificar o `controllers/contatos.js` e suas respectivas views. Nesta etapa, exploraremos novas funções do

MongoDB através do modelo `Usuario`. Veja a seguir as mudanças para cada *action*:

```
module.exports = function(app) {
 var Usuario = app.models.usuario;

 var ContatosController = {
 index: function(req, res) {
 var _id = req.session.usuario._id;
 Usuario.findById(_id, function(erro, usuario) {
 var contatos = usuario.contatos;
 var resultado = { contatos: contatos };
 res.render('contatos/index', resultado);
 });
 },
}
```

Na *action* `index`, utilizamos a função `Usuario.findById()` que retorna apenas um usuário baseado no `_id` em parâmetro. Essa função será o suficiente para retornar os dados do usuário e todos os seus contatos.

```
create: function(req, res) {
 var _id = req.session.usuario._id;
 Usuario.findById(_id, function(erro, usuario) {
 var contato = req.body.contato;
 var contatos = usuario.contatos;
 contatos.push(contato);
 usuario.save(function() {
 res.redirect('/contatos');
 });
 });
},
```

Já na *action* `create`, temos apenas de atualizar a lista de contatos incluindo um novo contato. Para isso, buscamos o usuário via função `Usuario.findById()` e, em seu callback, atualizamos o array `usuario.contatos` com a função `contatos.push(contato)`. Em seguida, é executada a função `usuario.save()`.

```

show: function(req, res) {
 var _id = req.session.usuario._id;
 Usuario.findById(_id, function(erro, usuario) {
 var contatoID = req.params.id;
 var contato = usuario.contatos.id(contatoID);
 var resultado = { contato: contato };
 res.render('contatos/show', resultado);
 });
},
edit: function(req, res) {
 var _id = req.session.usuario._id;
 Usuario.findById(_id, function(erro, usuario) {
 var contatoID = req.params.id;
 var contato = usuario.contatos.id(contatoID);
 var resultado = { contato: contato };
 res.render('contatos/edit', resultado);
 });
},

```

As *actions* `show` e `edit` possuem o mesmo comportamento. Eles retornam os dados de um específico contato do usuário através da função `usuario.contatos.id(contatoID)` , que é uma função do subdocumento `contatos` que retorna um contato baseado em seu `_id` .

```

update: function(req, res) {
 var _id = req.session.usuario._id;
 Usuario.findById(_id, function(erro, usuario) {
 var contatoID = req.params.id;
 var contato = usuario.contatos.id(contatoID);
 contato.nome = req.body.contato.nome;
 contato.email = req.body.contato.email;
 usuario.save(function() {
 res.redirect('/contatos');
 });
 });
},

```

Nessa *action*, a implementação do `update` não tem segredos. Como `contatos` é um subdocumento, seu tratamento é semelhante às funções de um `array` , por isso praticamente

buscamos o usuário de acordo com seu `_id` via função `Usuario.findById()`. Em seguida, buscamos o seu respectivo contato com base em seu `contatoID`, e atualizamos seus atributos normalmente. Quando executamos a função `usuario.save()`, o contato será atualizado na base de dados.

```
destroy: function(req, res) {
 var _id = req.session.usuario._id;
 Usuario.findById(_id, function(erro, usuario) {
 var contatoID = req.params.id;
 usuario.contatos.id(contatoID).remove();
 usuario.save(function() {
 res.redirect('/contatos');
 });
 });
}
```

Em `destroy`, temos três ações para excluir um contato. Primeiro, buscamos um contato via função `Usuario.findById()`. Em seguida, buscamos um específico contato do usuário já o excluindo através da linha `usuario.contatos.id(contatoID).remove()` e, para finalizar, atualizamos os dados do usuário executando `usuario.save()`.

Para terminar, vamos atualizar as views do contato. A edição será bem simples, vamos apenas trocar a maneira como eles renderizam o `id` do contato nas URLs. Veja a seguir como faremos essa atualização. Abra o arquivo `views/contatos/edit.ejs`, nele mude apenas a URL da `action` de seus respectivos `form`, para que seja renderizado o atributo `contato._id`:

```
<form action="/contato/<%- contato._id %>?_method=put"
 method="post">
```

Já em `views/contatos/show.ejs`, faça as seguintes

modificações:

```
<% include ../header %>
<header>
 <h2>Ntalk - Dados do contato</h2>
</header>
<section>
 <form action="/contato/<%- contato._id %>?_method=delete"
 method="post">
 <p><label>Nome:</label><%- contato.nome %></p>
 <p><label>E-mail:</label><%- contato.email %></p>
 <p>
 <button type="submit">Excluir</button>
 <a href="/contato/<%- contato._id %>/editar">Editar
 </p>
 </form>
</section>
<% include ../exit %>
<% include ../footer %>
```

Agora em `views/contatos/index.ejs` alteraremos as URLs dos link gerados na iteração do loop de contatos:

```
<% contatos.forEach(function(contato) { %>
 <tr>
 <td><%- contato.nome %></td>
 <td><%- contato.email %></td>
 <td>
 <span id="notify_<%- contato.email %>" class="label">
 Offline

 </td>
 <td>
 <a href="/contato/<%- contato._id %>">Detalhes
 <a href="/chat" id="chat_<%- contato.email %>">Conversar

 </td>
 </tr>
<% }) %>
```

Com as views finalizadas, terminamos o nosso *refactoring* na agenda de contatos. Para verificar se tudo ocorreu bem, reinicie o

servidor e confira as novidades no sistema. Dessa vez, temos um sistema integrado ao MongoDB, persistindo seus contatos, cadastrando usuário e fazendo login corretamente.

## 7.6 PERSISTINDO ESTRUTURAS DE DADOS USANDO REDIS

Com nossa agenda integrada ao MongoDB e persistindo contatos no banco de dados, precisamos agora persistir dados das conversas do chat da aplicação, para que ele mantenha um histórico de conversas.

Como o chat será uma área de maior acesso, precisamos armazenar seus dados em uma estrutura simples de chave-valor que permita leituras muito rápido. Para manter esse tipo de estrutura, o MongoDB não seria uma boa solução, pois precisamos de um banco de dados que mantenha dados frequentemente em memória para garantir uma leitura rápida deles. Ele se chama Redis.



Figura 7.4: NoSQL Redis

O Redis guarda e busca, em sua base de dados, elementos chave-valor, de maneira extremamente rápida, pois mantém os dados em grande parte do tempo na memória, fazendo em curtos

períodos a sincronização dos dados com disco rígido. Ele é considerado um NoSQL do tipo em chave-valor, em que a chave é o identificador e o valor pode ser diversos tipos de estruturas de dados.

As estruturas de dados com que ele trabalha são: *Strings*, *Hashes*, *Lists*, *Sets* e *Sorted Sets*. Ele possui um CLI (através do comando `redis-cli`) que permite em *runtime* brincar com seus inúmeros comandos — aliás, são vários comandos que realizam operações com os dados, vale a pena dar uma olhada em sua documentação, disponível em <http://redis.io/commands>.

## 7.7 MANTENDO UM HISTÓRICO DE CONVERSAS DO CHAT

Usaremos o Redis para implementar o histórico de conversas do nosso chat. Basicamente, vamos persistir cada mensagem em uma lista, agrupando-a em uma chave que será o `id` da sala do chat. Isso vai fazer com que o usuário que receber uma mensagem consiga visualizá-la após clicar no botão **Conversar**.

Não entraremos em detalhes sobre como instalar e configurar o Redis, visto que sua instalação é muito simples, você só precisa baixar e instalar. E fique tranquilo! Os exemplos aplicados neste livro utilizam a configuração padrão dele. Para começar com o Redis, recomendo que visite seu site oficial: <http://redis.io/download>.

Já considerando que o Redis está instalado e funcionando corretamente em sua máquina, instalaremos seu *driver* compatível com Node.js e um módulo adicional, que é um *wrapper* que

aumenta performance do Redis com Node.js, permitindo processamento assíncrono e outras otimizações de baixo nível nas operações do Redis: ele se chama *Hi-Redis*. Apenas o instale, que o próprio módulo Redis o utilizará automaticamente. No terminal, execute o comando:

```
npm install redis hiredis --save
```

**Obs.:** Caso você esteja usando Windows, não será necessário instalar o HiRedis, pois ele não é compatível com Windows, ele é apenas um módulo que melhora performance em sistemas Linux, Unix e MacOSX.

Com o Redis e seu respectivo driver instalados corretamente, vamos ao que interessa: implementar o histórico do chat. Abra o código `sockets/chat.js`; será nele que vamos conectar o driver com o servidor Redis para habilitar seus comandos na aplicação.

É a partir da execução de `redis = require('redis').createClient()` que tudo começa. Praticamente, ele carrega o driver e retorna um cliente Redis. Como o banco Redis e a aplicação Node.js estão hospedados na mesma máquina e utilizamos as configurações padrões do Redis, então não há necessidade de enviar parâmetros extras para função `createClient()`. Caso você necessite conectar de um local diferente, apenas inclua o seguinte parâmetro: `createClient(porta, ip)`. Veja como será o nosso código:

```
module.exports = function(io) {

 var crypto = require('crypto')
 , redis = require('redis').createClient()
 , sockets = io.sockets
 , onlines = {}
 ;
};
```

```
// continuação dos eventos do socket.io...
}
```

Agora com um cliente Redis em ação, vamos implementar algumas de suas funções para pesquisar e persistir as mensagens. Usaremos a estrutura de lista para armazenar as mensagens. Cada lista terá uma sala como chave para pesquisa. Primeiro, vamos implementá-la no evento `client.on('join')` :

```
client.on('join', function(sala) {
 if(!sala) {
 var timestamp = new Date().toString()
 , md5 = crypto.createHash('md5')
 ;
 sala = md5.update(timestamp).digest('hex');
 }
 session.sala = sala;
 client.join(sala);

 var msg = "" + usuario.nome + " entrou.
";
 redis.lpush(sala, msg, function(erro, res) {
 redis.lrange(sala, 0, -1, function(erro, msgs) {
 msgs.forEach(function(msg) {
 sockets.in(sala).emit('send-client', msg);
 });
 });
 });
});
```

Basicamente, utilizamos duas de suas funções: primeiro executamos a função `redis.lpush(sala, msg)` que adiciona na lista a mensagem, e depois em seu callback utilizamos a função `redis.lrange(sala, 0, -1)`, que retorna um array contendo os elementos a partir de um range inicial e final da lista.

O range utiliza dois índices e, neste caso, o índice inicial é `0` e o final é `-1`. Quando informamos o valor `-1` no índice final, indicamos que o range será total, retornando todos os elementos da lista. Por último, no callback do `redis.lrange()`, iteramos o

array de mensagens emitindo mensagem por mensagem para o cliente.

Agora para finalizar o nosso histórico do chat, implementaremos a função `redis.lpush` nos eventos `send-server` e `disconnect`. Como estes eventos enviam uma única mensagem, simplificaremos o código incluindo a função `redis.lpush(sala, msg)` sem utilizar callbacks:

```
client.on('disconnect', function () {
 var sala = session.sala
 , msg = ""+ usuario.nome +": saiu.
";
 redis.lpush(sala, msg);
 client.broadcast.emit('notify-offlines', usuario.email);
 sockets.in(sala).emit('send-client', msg);
 delete onlines[usuario.email];
 client.leave(sala);
});

client.on('send-server', function (msg) {
 var sala = session.sala
 , data = {email: usuario.email, sala: sala};
 msg = ""+usuario.nome+": "+msg+"
";
 redis.lpush(sala, msg);
 client.broadcast.emit('new-message', data);
 sockets.in(sala).emit('send-client', msg);
});
```

## 7.8 PERSISTINDO LISTA DE USUÁRIOS ONLINE

Para finalizar a utilização do Redis no nosso chat, vamos fazer uma simples, porém robusta, modificação na lista de usuários online. Em vez de guardarmos os e-mails dos usuários online em memória na aplicação, vamos persistir esses e-mails também no Redis.

Para realizar essa alteração, vamos usar uma estrutura de dados que só mantém dados distintos, conhecida pelo nome `Sets`. Para trabalhar com essa estrutura, utilizaremos as funções:  
`redis.sadd('onlines')` para incluir um e-mail;  
`redis.srem('onlines')` para remover um e-mail; e  
`redis.smembers('onlines')` para listar todos os e-mails existentes na estrutura.

Abra novamente o `sockets/chat.js`, nele remova todas as referências a variável `onlines` e altere para as chamadas das funções do `redis`. Dessa forma, o loop de e-mails `onlines` vai ficar assim:

```
sockets.on('connection', function (client) {
 var session = client.handshake.session
 , usuario = session.usuario
 ;
 redis.sadd('onlines', usuario.email, function(error) {
 redis.smembers('onlines', function(error, emails) {
 emails.forEach(function(email) {
 client.emit('notify-onlines', email);
 client.broadcast.emit('notify-onlines', email);
 });
 });
 });
 // continuação do chat.js ...
});
```

Agora, dentro do evento `client.on('disconnect')`, substitua a o trecho `delete onlines[usuario.email]` para a função `redis.srem('onlines', usuario.email)`, deixando o código semelhante a este:

```
client.on('disconnect', function () {
 var sala = session.sala
 , msg = "" + usuario.nome + ": saiu.
";
 redis.lpush(sala, msg);
 client.broadcast.emit('notify-offlines', usuario.email);
```

```
 sockets.in(sala).emit('send-client', msg);
 redis.srem('onlines', usuario.email);
 client.leave(sala);
});
```

Mais uma vez, terminamos um excelente capítulo! Agora temos uma aplicação 100% funcional que utiliza dois banco de dados NoSQL. Acredite, o que já temos aqui já é o suficiente para colocar a nossa aplicação no ar.

Mas continue lendo! Afinal, nos próximos capítulos vamos aprofundar nossos conhecimentos codificando testes e também otimizando o sistema para que ele entre em um ambiente de produção de forma eficiente.

## CAPÍTULO 8

# PREPARANDO UM AMBIENTE DE TESTES

## 8.1 MOCHA, O FRAMEWORK DE TESTES PARA NODE.JS

Teste automatizado é algo cada vez mais adotado no mundo de desenvolvimento de sistemas. Existem diversos tipos de testes: teste unitário, teste funcional, teste de aceitação, entre outros. Neste capítulo, focaremos apenas no teste de aceitação, para o qual vamos utilizar alguns frameworks do Node.js.

O mais recente e que anda ganhando visibilidade pela comunidade é o Mocha, cujo site é <https://mochajs.org/>.



Figura 8.1: Mocha – Framework para testes

Ele é mantido pelo mesmo criador do Express (TJ Holowaychuk), e foi construído com as seguintes características: teste no estilo TDD, testes no estilo BDD, cobertura de código, relatório em HTML, teste de comportamento assíncrono, e integração com os módulos `should` e `assert`. Praticamente, ele é um ambiente completo para desenvolvimento de testes, e possui diversas interfaces de apresentação do resultado dos testes.

Nas seções a seguir, apresentarei o Mocha, desde a sua configuração até a implementação de testes no projeto Ntalk.

## 8.2 CRIANDO UM ENVIRONMENT PARA TESTES

Antes de entrarmos a fundo nos testes, primeiro temos de criar um novo ambiente com configurações específicas para testes. Isso envolve criar uma função que contenha informações para se conectar em um banco de dados de testes e de desenvolvimento. Com isso, vamos migrar a função `mongoose.connect` para um novo arquivo, para que ele retorne uma conexão de banco de dados de acordo com o ambiente, seja ambiente de testes ou de desenvolvimento.

Para identificar em qual ambiente está o projeto, usamos a variável `process.env.NODE_ENV`, que vai procurar se existe essa variável de ambiente no seu sistema operacional.

Vamos criar um novo diretório, responsável por manter bibliotecas externas, ele se chamará de `libs`. Nele, vamos criar o arquivo `libs/db_connect.js` e inserir a seguinte lógica:

```
module.exports = function() {
```

```
var mongoose = require('mongoose');
var env_url = {
 "test": "mongodb://localhost:27017/ntalk_test"
 , "development": "mongodb://localhost:27017/ntalk"
};
var url = env_url[process.env.NODE_ENV || "development"];
return mongoose.connect(url);
};
```

### LENDÔ VARIÁVEIS DE AMBIENTE NO NODE

Quando se trabalha com variáveis de ambiente é muito comum persistir dados de configurações ou dados sensitivos no sistema operacional. A URL de acesso a banco de dados ou outros serviços, assim como senhas e chaves importantes de acesso a sistemas externos, são alguns exemplos de variáveis de ambiente. Esses dados são configurados em um arquivo no próprio sistema operacional. No capítulo *Bem-vindo ao mundo Node.js*, foi explicado como criar a variável `NODE_ENV` ; para outras variáveis se faz o mesmo procedimento. E no Node.js podemos lê-las através do `process.env["VARIAVEL"]` , sendo que `process.env` é um objeto JSON que contém todas as variáveis do sistema operacional.

Com essa função preparada, **remova do arquivo** `app.js` o carregamento do `mongoose` pela sua variável `global.db` . Afinal, quanto menos variáveis globais existirem na aplicação, melhor. Como preparamos uma biblioteca que retorna uma conexão de acordo com a variável de ambiente `NODE_ENV` , o uso do `db_connect.js` será no modelo `models/usuario.js` :

```
module.exports = function(app) {
 var db = require('../lib/db_connect')()
 , Schema = require('mongoose').Schema;

 var contato = Schema({
 nome: String
 , email: String
 });
 var usuario = Schema({
 nome: { type: String, required: true }
 , email: { type: String, required: true
 , index: {unique: true} }
 , contatos: [contato]
 });

 return db.model('usuarios', usuario);
};
```

Dessa forma, mantemos nossa aplicação com nenhuma variável global, e a variável `db` que mantém uma conexão com MongoDB será gerenciada pelo `libs/db_connect.js`.

Pronto! Essa foi uma demonstração simples de como configurar uma aplicação utilizando variáveis de ambiente. Com essas configurações, ela estará preparada para funcionar em multiambientes. A princípio, só criamos uma biblioteca que retorna uma instância de conexão com banco de dados de acordo com sua variável de ambiente, mas em aplicações mais complexas, usa-se muito desse conceito para criar outros tipos de bibliotecas.

## 8.3 INSTALANDO E CONFIGURANDO O MOCHA

Para começarmos com o Mocha, vamos instalá-lo em modo global para a utilização do seu CLI. Em seu console, execute o comando:

```
npm install -g mocha
```

O Mocha é um módulo focado em testes e vamos adicioná-lo no `package.json`, porém ele não será incluído dentro de `dependencies`, e sim como `devDependencies`. Para adicionar um `devDependencies`, você precisa instalar um módulo utilizando a *flag* `--save-dev`. O motivo é que ele é muito pesado, e não é um framework para ser carregado em um ambiente de produção. Neste caso, instale-o rodando o comando:

```
npm install mocha --save-dev
```

Na seção seguinte, implementaremos alguns testes utilizando interface BDD (*Behavior Driven-Development*) para usar funções como `describe`, `it`, `beforeEach`, `should` e outras. A função `should` faz verificações em cima dos resultados de cada testes, porém ela não é nativa do framework Mocha. Com isso, teremos que habilitá-la o módulo `should`.

```
npm install should --save-dev
```

Para explorarmos o Mocha, neste livro apenas implementaremos testes em cima das rotas da aplicação. Para realizar esse tipo de teste precisamos de um módulo que faça requisições em nosso servidor. Testar requisições sobre as rotas é muito útil, pois permite verificar como será o comportamento de uma requisição feita por um usuário. Para realizar esses testes, utilizaremos o módulo `supertest` que também nasceu pelo os mesmos criadores do Mocha. Instale-o rodando o comando:

```
npm install supertest --save-dev
```

Para finalizar esta seção, crie o diretório `test`, pois é lá o local em que codificaremos os testes da aplicação.

## 8.4 RODANDO O MOCHA NO AMBIENTE DE TESTES

Assim como criamos o `libs/db_connect.js`, que é um simples biblioteca que retorna uma conexão MongoDB baseado no valor da variável `NODE_ENV`, temos de executar os testes utilizando essa variável com valor '`test`' para que os testes rodem no seu devido ambiente. Para isso, um simples comando no terminal, `mocha test`, já será o suficiente, mas neste caso ele não será executado no ambiente de testes. O comando correto é `NODE_ENV=test mocha test`, pois ele define o valor de `NODE_ENV` para `test`.

Mas executar esse comando longo seria um pouco cansativo, não é? E um bom programador, tem que ser preguiçoso! Que tal simplificá-lo?

Uma boa prática para simplificar este comando é utilizar o `package.json` para definir comandos dentro do atributo `"scripts"`. Ele será convertido para um comando executável via `npm`. Abra o `package.json` e crie os seguintes comandos:

```
"scripts": {
 "start": "node app.js",
 "test": "NODE_ENV=test ./node_modules/.bin/mocha test/**/*.js"
}
```

Caso esteja no Windows, faça o seguinte:

```
"scripts": {
 "start": "node app.js",
 "test": "SET NODE_ENV=test && ./node_modules/.bin/mocha test/**/*.js"
}
```

Foram adicionados dois comandos dentro do atributo

`scripts` : o `start` e o `test`. Estes são os comandos atalhos do `npm`, ou seja, agora os testes serão executados via comando `npm test` e a aplicação será iniciada via comando `npm start`. Dentro de `scripts`, você pode criar quantos comandos quiser, porém para os demais comandos, todos eles serão executados via comando `npm run [nome-do-comando]`, inclusive `npm run test` e `npm run start`.

#### **POR QUE USAMOS O MOCHA DA PASTA NODE\_MODULES?**

Dentro do nosso projeto, cada dependência dele fica localizada dentro da pasta `node_modules`. Apenas **módulos globais** ficam fora desta pasta, afinal, eles são armazenados em uma pasta específica do seu sistema operacional. Quando utilizamos os comandos `npm test` ou `npm start`, qualquer uso de módulo deve obrigatoriamente ser chamado dentro do diretório `node_modules` de seu projeto, para que esta chamada interna funcione em qualquer sistema operacional.

Um bom exemplo de serviço é o Travis CI (<http://travis-ci.org>), um serviço de *deploy contínuo* compatível com diversas linguagens, inclusive Node.js. Ele roda os testes do seu projeto através do comando `npm test`, e não instala módulos globais, apenas utiliza os módulos existentes da pasta `node_modules` do projeto.

## 8.5 TESTANDO AS ROTAS

O módulo `supertest` será intensivamente utilizado e, antes de criarmos os testes, temos de exportar a variável `app` do `app.js` para que automaticamente levante uma instância de servidor para rodar os testes. Este *refactoring* é muito simples, **apenas inclua na última linha** do `app.js` o seguinte trecho:

```
// Trecho final do app.js...
module.exports = app;
```

Feito isso, agora podemos criar os primeiros testes. Crie o arquivo `test/requests/home.js`. No primeiro teste, simularemos uma requisição para a rota principal `"/"` esperando que retorne o status `200` como sucesso da requisição.

```
var app = require('../app')
, should = require('should')
, request = require('supertest')(app);

describe('No controller home', function() {
 it('deve retornar status 200 ao fazer GET /',function(done){
 request.get('/')
 .end(function(err, res){
 res.status.should.eql(200);
 done();
 });
 });
 // continuação...
```

No mesmo arquivo, implementaremos o teste a seguir, que faz uma requisição para rota `"/sair"`, e seu comportamento de sucesso é receber um redirecionamento para rota principal `"/"`, que será o retorno da variável `res.headers.location`.

```
it('deve ir para rota / ao fazer GET /sair', function(done){
 request.get('/sair')
 .end(function(err, res){
 res.headers.location.should.eql('/');
 done();
 });
});
```

```
});
```

Aqui já complicamos o teste, simulamos um POST enviando parâmetros válidos (nome e email) , que faz um login e é redirecionado para rota "/contatos" .

```
it('deve ir para rota /contatos ao fazer POST /entrar',
function(done){
 var login = {
 usuario: {nome: 'Teste', email: 'teste@teste'}
 };
 request.post('/entrar')
 .send(login)
 .end(function(err, res){
 res.headers.location.should.eql('/contatos');
 done();
 });
});
```

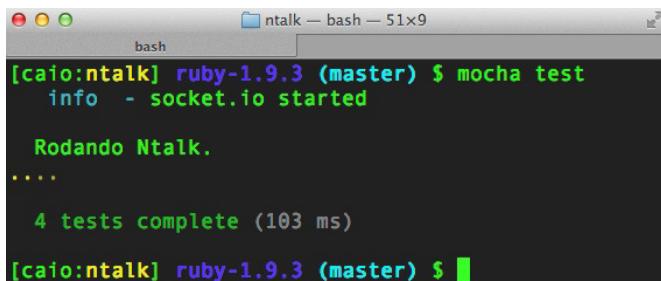
Este último teste é semelhante ao anterior, a diferença é que enviamos parâmetros inválidos de login, para que seja testado o comportamento de redirecionamento da rota.

```
it('deve ir para rota / ao fazer POST /entrar',
function(done){
 var login = {usuario: {nome: '', email: ''}};
 request.post('/entrar')
 .send(login)
 .end(function(err, res){
 res.headers.location.should.eql('/');
 done();
 });
});
```

}); // fim da função describe()

Esse foi o nosso teste com a rota routes/home.js . Deixei cada teste bem descriptivo e atômico, realizando apenas uma única verificação através da função should . Vamos rodar os testes para ver se tudo deu certo?

Para executar os testes, é simples! Como já configuramos no package.json o script para execução de testes, execute no terminal o comando `npm test`. Veja o resultado dos seus testes semelhantes ao da figura:



```
[caio:ntalk] ruby-1.9.3 (master) $ mocha test
 info - socket.io started

 Rodando Ntalk.

 ...
 4 tests complete (103 ms)

[caio:ntalk] ruby-1.9.3 (master) $
```

Figura 8.2: Os testes em home.js passaram com sucesso

Reparam que surgiram dois prints do sistema que só aparecem quando levantamos o servidor e, em seguida, aparece o resultado dos testes. Isso acontece porque em `app.js` exportamos a variável que contém uma instância de um servidor da aplicação, o `module.exports = app`. Nos testes, quando carregamos este módulo e injetamos dentro do `require('supertest')(app)`, ele se encarrega de iniciar o servidor para que o módulo `supertest` possa emular as requisições neste servidor e assim permitir que façamos os testes funcionais em cima das rotas.

Agora vamos explorar novas funções do Mocha, testando as rotas de `routes/contatos.js`. O *controller* dessas rotas possui um filtro que verifica se existe um usuário logado no sistema. Implementaremos dois casos de testes: um caso de testes para usuário logado, e um caso para um usuário não logado.

Crie o arquivo `test/requests/contatos.js`. Nele vamos

criar um `describe` com duas funções `describe` interna, que serão usadas para descrever os casos de testes para usuário logado e não logado.

```
var app = require('../app')
, should = require('should')
, request = require('supertest')(app);

describe('No controller contatos', function() {

 describe('o usuario nao logado', function() {
 // testes aqui...
 });

 describe('o usuario logado', function() {
 // testes aqui...
 });

});
```

No caso de testes para usuário não logado, implementaremos um simples teste de verificação, semelhante ao que utilizamos no `test/requests/home.js`. Afinal, em `routes/contatos.js`, existe um filtro que barrará um usuário não logado, fazendo com que a requisição seja redirecionada para a rota da homepage. Precisamos testar este comportamento.

Para simplificar, seguem todos os testes que serão inseridos dentro de `describe('o usuario nao logado')`:

```
describe('deve voltar para /', function() {
 it('ao fazer GET /contatos', function(done){
 request.get('/contatos').end(function(err, res) {
 res.headers.location.should.eql('/');
 done();
 });
 });
 it('ao fazer GET /contato/1', function(done){
 request.get('/contato/1').end(function(err, res) {
 res.headers.location.should.eql('/');
 });
 });
});
```

```

 done();
 });
});
it('ao fazer GET /contato/1/editar', function(done){
 request.get('/contato/1/editar').end(function(err, res) {
 res.headers.location.should.eql('/');
 done();
 });
});
it('ao fazer POST /contato', function(done){
 request.post('/contato').end(function(err, res) {
 res.headers.location.should.eql('/');
 done();
 });
});
it('ao fazer DELETE /contato/1',function(done){
 request.del('/contato/1').end(function(err, res) {
 res.headers.location.should.eql('/');
 done();
 });
});
it('ao fazer PUT /contato/1', function(done){
 request.put('/contato/1').end(function(err, res) {
 res.headers.location.should.eql('/');
 done();
 });
});
});
});

```

Reparam que o teste dentro de `describe('o usuario nao logado')` foi tão repetitivo que foi criado mais um `describe ( describe('deve voltar para /') )` para descrever o resultado repetitivo. Isso ocorre pois o filtro que aplicamos no capítulo *Iniciando com o Express* vai barrar qualquer requisição de usuário não autenticado pelo login. Por isso, o resultado correto é que todos os testes redirecionem para a rota principal do sistema, que é a tela de login.

Mais uma vez, vamos rodar os testes para ter a certeza de que tudo ocorreu bem na implementação dos testes. Dessa vez,

veremos em mais detalhes os resultados. Para isso, execute o comando `NODE_ENV=test mocha test --reporter spec`, e os resultados serão apresentados no formato semelhante ao framework RSpec da linguagem Ruby, igual à figura seguinte:

The screenshot shows a terminal window titled "ntalk — bash — 68x24". The command entered is "[caio:ntalk] ruby-1.9.3 (master) \$ mocha --reporter spec test". The output is as follows:

```
Rodando Ntalk.
No controller contatos
 o usuario nao logado
 ✓ deve ir para rota / ao fazer GET /contatos (27ms)
 ✓ deve ir para rota / ao fazer GET /contato/1
 ✓ deve ir para rota / ao fazer GET /contato/1/editar
 ✓ deve ir para rota / ao fazer POST /contato
 ✓ deve ir para rota / ao fazer DELETE /contato/1
 ✓ deve ir para rota / ao fazer PUT /contato/1

No controller home
 ✓ deve retornar status 200 ao fazer GET / (15ms)
 ✓ deve ir para rota / ao fazer GET /sair
 ✓ deve ir para rota /contatos ao fazer POST /entrar (12ms)
 ✓ deve ir para rota / ao fazer POST /entrar (13ms)

10 tests complete (127 ms)
```

Figura 8.3: Resultado dos testes utilizando Reporter Spec

O mais legal do Mocha é que ele possui diversos *reporters*, permitindo que você tenha várias opções para customizar o resultado de seus testes. Para conhecer outros formatos de *reporters*, acesse <http://visionmedia.github.io/mocha/#reporters>.

No `describe('o usuario logado')`, teremos de emular um usuário autenticado, ou seja, um usuário que fez um login no sistema. Para isso, usaremos duas estratégias:

1. Cada teste deve, **antes**, fazer uma requisição `POST` na rota de login;
2. O mesmo teste deve manter um **cookie** válido, gerado após o

login obter sucesso, para pular o filtro.

Para implementar essa rotina, vamos usar a função `beforeEach()`, que é executada antes de cada teste. Dentro dessa função, faremos um login no sistema para capturar o seu `cookie`, que é encontrado dentro de `res.headers['set-cookie']`. Nisso, armazenaremos seu resultado em uma variável que estará no mesmo escopo da função `describe('o usuario logado')` para que seja reutilizado em cada um de seus testes. Para entender melhor, veja o seguinte código:

```
describe('o usuario logado', function() {
 var login = {usuario: {nome: 'Teste', email: 'teste@teste'}}
 , contato =
 {contato: {nome: 'Teste', email: 'teste@teste'}}
 , cookie = {};

 beforeEach(function(done) {
 request.post('/entrar')
 .send(login)
 .end(function(err, res) {
 cookie = res.headers['set-cookie'];
 done();
 });
 });
 // implementação dos testes...
});
```

Com a variável `cookie` recebendo os dados de um usuário autenticado, fica viável testar o comportamento das rotas pós-filtro. Para executar os testes, temos de injetar o `cookie` em cada requisição, e isso se faz via `req.cookies = cookie`. Vamos implementá-los?

A seguir, utilizaremos essa técnica, mas infelizmente testaremos apenas algumas rotas. Após a apresentação dos testes, explicarei o motivo.

Aqui testamos uma requisição GET na rota /contatos :

```
// função beforeEach()...
it('deve retornar status 200 em GET /contatos',function(done){
 var req = request.get('/contatos');
 req.cookies = cookie;
 req.end(function(err, res) {
 res.status.should.eql(200);
 done();
 });
});
```

No seguinte teste, emulamos uma requisição POST na rota /contato , testando o comportamento do cadastro de um contato:

```
it('deve ir para rota /contatos em POST /contato',
function(done){
 var contato =
 {contato: {nome: 'Teste', email: 'teste@teste'}};
 var req = request.post('/contato');
 req.cookies = cookie;
 req.send(contato).end(function(err, res) {
 res.headers.location.should.eql('/contatos');
 done();
 });
});
}); // fim da função describe()
```

Então vem a pergunta: *por que não foram testadas todas as rotas para um usuário logado?*

Testes de aceitação verificam o comportamento do sistema simulando uma requisição real, pelo qual testamos as rotas. Esse tipo de teste aqui está automatizado; entretanto, por ele ser um típico *teste de caixa-preta*, também é possível realizá-lo manualmente, como um usuário acessando o sistema.

As rotas de routes/contatos.js , que passam um id como

parâmetro, precisam de um `id` válido que retorne um objeto do banco de dados. Para fins didáticos, vamos parar por aqui, mas caso queira testar essas rotas, será necessário criar objetos *fakes* no banco de dados, que utilizem `ids` fixos para possibilitar a elaboração desses testes.

## 8.6 DEIXANDO SEUS TESTES MAIS LIMPOS

Algo que polui muito os códigos de teste é o excesso de variáveis, que são carregadas no topo de cada teste. É claro que, por questões de legibilidade e entendimento do teste, é necessário declará-las. Porém, é possível criar um arquivo de configuração do próprio Mocha para que sejam centralizados em um único arquivo o carregamento do framework `should` e alguns outros parâmetros iniciais de execução.

Esse arquivo deve ser criado com o nome `test/mocha.opts`. Basicamente, ele permite utilizar os parâmetros de configuração do seu próprio CLI, além de carregar alguns módulos auxiliares, como o `should`. Isso será o suficiente para deixar os testes mais limpos.

Crie o arquivo `test/mocha.opts`, seguindo os parâmetros a seguir:

```
--require should
--reporter spec
```

Além de carregarmos o `should`, também foi adicionado um novo parâmetro, o `--reporter spec`, que define um novo layout de resultado dos testes. Caso queria incluir outros parâmetros em seu `mocha.opts`, execute no terminal o comando `mocha -h` para visualizar todas as opções possíveis de configuração.

The screenshot shows a terminal window with multiple tabs, all titled "bash". The active tab displays the help output for the "mocha" command. The output includes usage instructions, command descriptions, and a detailed list of options with their meanings. The text is in Portuguese.

```
[caio:ntalk] ruby-1.9.3 (master) $ mocha -h
Usage: mocha [debug] [options] [files]
 como também permite carregar alguns módulos auxiliares: should. E
 e para deixar mais limpo os testes.
init <path>
 initialize a client-side mocha setup at <path> mocha.opts, seguindo os parâmetros

Options:
 [code bash]
 -h, --help ou should output usage information
 -V, --version sync output the version number
 -r, --require <name> require the given module
 -R, --reporter <name> specify the reporter to use
 -U, --ui <name> specify user-interface (bdd|tdd|exports)
 -g, --grep <pattern> only run tests matching <pattern>
 -i, --invert inverts --grep matches
 -t, --timeout <nms> set test-case timeout in milliseconds [2000]
 -S, --slow <nms> slow" test threshold in milliseconds [75]
 -W, --watch watch files for changes
 -c, --colors force enabling of colors
 -C, --no-colors force disabling of colors
 -G, --growl enable growl notification support
 -d, --debug enable node's debugger, synonym for node --debug
 -b, --bail bail after first test failure
 -A, --async-only --N para
 --recursive force all tests to take a callback (async)
 --debug-brk include sub directories
 --globals <names> executando os
 --check-leaks allow the given comma-delimited global [names]
 --interfaces check for global variable leaks
 --reporters display available interfaces
 --compilers <ext>:<module>,... use the given module(s) to compile files
```

Figura 8.4: Parâmetros opcionais do Mocha

Agora, só para finalizar, remova a função `var should = require('should')` de todos os testes criados, pois agora, eles serão automaticamente carregados via `mocha.opts`.

# APLICAÇÃO NODE EM PRODUÇÃO – PARTE 1

## 9.1 O QUE VAMOS FAZER?

Nas próximas seções, serão abordados temas importantes para preparar o nosso projeto para o ambiente de produção. O objetivo aqui é apresentar alguns conceitos e ferramentas para manter uma aplicação Node.js de forma segura e com boa performance. Otimizaremos o projeto Ntalk, preparando-o para entrar em ambiente de produção, além de garantir toda a monitoria do sistema por meio de *loggings*.

## 9.2 CONFIGURANDO CLUSTERS

Infelizmente, o Node.js não trabalha com *threads*. Isso é algo que, na opinião de alguns desenvolvedores, é considerado como um ponto negativo, e que provoca um certo desinteresse em aprender ou levar a sério essa tecnologia. Entretanto, apesar de ele ser *single-thread*, é possível, sim, prepará-lo para trabalhar com processamento paralelo. Para isso, existe nativamente um módulo chamado *cluster*.

Ele basicamente instancia novos processos de uma aplicação,

trabalhando de forma distribuída e, quando trabalhamos com uma aplicação web, esse módulo se encarrega de compartilhar a mesma porta da rede entre os *clusters* ativos. O número de processos a serem criados é você quem determina, e é claro que a boa prática é instanciar um total de processos relativo à quantidade de núcleos do processador do servidor ou também uma quantidade relativa a núcleos X processadores.

Por exemplo, se tenho um único processador de oito núcleos, então posso instanciar oito processos, criando assim uma rede de oito *clusters*. Mas, caso tenha quatro processadores de oito núcleos cada, é possível criar uma rede de trinta e dois *clusters* em ação.

Para garantir que os *clusters* trabalhem de forma distribuída e organizada, é necessário que exista um processo pai, mais conhecido como *cluster master*. Ele é o processo responsável por balancear a carga de processamento entre os demais *clusters*, distribuindo a carga para os processos filhos que são chamados de *cluster slave*. Implementar essa técnica no Node.js é muito simples, visto que toda distribuição de processamento é executada de forma abstruída para o desenvolvedor.

Outra vantagem é que os *clusters* são independentes uns dos outros. Ou seja, caso um *cluster* saia do ar, os demais continuarão servindo a aplicação mantendo o sistema no ar. Porém, é necessário gerenciar as instâncias e encerramento desses *clusters* manualmente.

Com base nesses conceitos, vamos aplicar na prática a implementação de *clusters*. Crie no diretório raiz o arquivo `clusters.js`, para que, por meio dele, seja carregado clusters da nossa aplicação. Veja o código a seguir:

```

var cluster = require('cluster')
, cpus = require('os').cpus()
;
if (cluster.isMaster) {
 cpus.forEach(function(cpu) {
 cluster.fork();
 });
 cluster.on('listening', function(worker) {
 console.log("Cluster %d conectado", worker.process.pid);
 });
 cluster.on('disconnect', function(worker) {
 console.log('Cluster %d esta desconectado.', worker.process.pid);
 });
 cluster.on('exit', function(worker) {
 console.log('Cluster %d caiu fora.', worker.process.pid);
 });
} else {
 require('./app');
}

```

Dessa vez, para levantar o servidor, vamos rodar o comando `node clusters.js` para que a aplicação rode de forma distribuída e para comprovar que deu certo. Veja no terminal quantas vezes se repetiu a mensagem "Ntalk no ar".

```

[caio:ntalk] ruby-1.9.3 (master) $ node server.js
 info - socket.io started
 info - socket.io started
 info - socket.io started
 info - socket.io started
Rodando Ntalk.
Rodando Ntalk.
Cluster 95230 conectado
Cluster 95227 conectado
Rodando Ntalk.
Cluster 95228 conectado
Rodando Ntalk.
Cluster 95229 conectado

```

Figura 9.1: Rodando Node.js em clusters

Basicamente, carregamos o módulo `cluster` e primeiro

verificamos se ele é o *cluster master* via função `cluster.isMaster`. Caso ele seja, rodamos um loop cujas iterações são baseadas no total de núcleos de processamento (*CPUs*) que ocorrem através do trecho `cpus.forEach()`, que retorna o total de núcleos do servidor. Em cada iteração, rodamos o `cluster.fork()` que, na prática, instancia um processo filho *cluster slave*.

Quando nasce um novo processo (neste caso, um processo filho), consequentemente ele não cai na condicional `if(cluster.isMaster)`. Com isso, é iniciado o servidor da aplicação via `require('./app')` para este processo filho.

Também foram incluídos alguns eventos que são emitidos pelo *cluster master*. No código anterior, utilizamos apenas os principais eventos:

- `listening` : acontece quando um *cluster* está escutando uma porta do servidor. Neste caso, a nossa aplicação está escutando **a porta 3000**;
- `disconnect` : executa seu callback quando um *cluster* se desconecta da rede;
- `exit` : ocorre quando um processo filho é fechado no sistema operacional.

## DESENVOLVIMENTO EM CLUSTERS

Muito pode ser explorado no desenvolvimento de *clusters* no Node.js. Aqui apenas aplicamos o essencial para manter nossa aplicação rodando em paralelo, mas caso tenha a necessidade de implementar mais detalhes que explorem ao máximo os *clusters*, recomendo que leia a documentação (<http://nodejs.org/api/cluster.html>) para ficar por dentro de todos os eventos e funções deste módulo.

Para finalizar e deixar automatizado o *start* do servidor em modo *cluster* (via comando `npm`), atualize em seu `package.json` no atributo `scripts` de acordo com o código a seguir:

```
"scripts": {
 "start": "node clusters.js",
 "test": "NODE_ENV=test ./node_modules/.bin/mocha test/**/*.js"
}
```

Pronto! Agora você pode levantar uma rede de *clusters* de sua aplicação através do comando `npm start` !

## 9.3 REDIS CONTROLANDO AS SESSÕES DA APLICAÇÃO

No Node.js, quando desenvolvemos uma aplicação orientada a *clusters*, aliado aos frameworks Express e Socket.IO, o mecanismo de persistência de sessão em memória para de funcionar corretamente. Não acredita? Então veja você mesmo! Execute o servidor via `npm start` e faça um login no sistema. Até agora esta

tudo ok, correto? Tente cadastrar um novo contato ou ver os detalhes de um existente. Repare que automaticamente você foi redirecionado para tela de login. Mas que estranho! Por que aconteceu isso?

No momento, usamos um controle de sessão em memória (através do objeto `expressSession.MemoryStore`). A natureza desse tipo de controle não consegue compartilhar dados entre os *clusters*, pois ele foi projetado para trabalhar com apenas um único processo. O Express e Socket.IO são frameworks que utilizam, por padrão, sessão em memória. A solução para esse problema é adotar um novo tipo de *storing* de sessão, e o Redis é uma ótima alternativa. Como já o utilizamos dentro do chat da aplicação, teremos agora apenas de adaptá-lo para o mecanismo *session store* do Express e do Socket.IO.

Essa adaptação é simples, e seu resultado visa manter a aplicação rodando perfeitamente em clusters. Implantaremos esse upgrade no mecanismo de sessão do Express e Socket.IO. Antes de começar os *refactorings*, será necessário instalar dois novos módulos: `connect-redis` para tratar sessões do Express no Redis e o `socket.io-redis` para o Socket.IO usar o Redis. Instale-os rodando no terminal o seguinte comando:

```
npm install connect-redis socket.io-redis --save
```

Agora, vamos modificar o `app.js` para usar estes módulos:

```
const KEY = 'ntalk.sid', SECRET = 'ntalk';
var express = require('express')
, load = require('express-load')
, bodyParser = require('body-parser')
, cookieParser = require('cookie-parser')
, expressSession = require('express-session')
, methodOverride = require('method-override')
```

```
, error = require('./middlewares/error')
, redisAdapter = require('socket.io-redis')
, RedisStore = require('connect-redis')(expressSession)
, app = express()
, server = require('http').Server(app)
, io = require('socket.io')(server)
, cookie = cookieParser(SECRET)
, store = new RedisStore({prefix: KEY})
;
// carregamento dos middlewares ...

io.adapter(redisAdapter({host: 'localhost', port: 6379}));

// continuação do app.js ...
```

Com esse upgrade implementado, agora o sistema vai rodar perfeitamente em *clusters*, sem causar bugs no controle de sessão, e controle de sessão será compartilhada entre os *clusters* existentes.

## 9.4 MONITORANDO APLICAÇÃO ATRAVÉS DE LOGS

Quando colocamos um sistema em produção, aumentamos os riscos de acontecerem bugs que não foram identificados durante os testes no desenvolvimento. Isso é normal, e toda aplicação já passou ou vai passar por esta situação. O importante neste caso é ter um meio de monitorar todo comportamento da aplicação, através de arquivos de *logs*.

Tanto o Express quanto o Socket.IO possuem *middlewares* para gerar *logs*. Para configurar um modo debug em nossa aplicação, para que seja gerado *logs* da aplicação, é muito simples! Você precisa apenas iniciar o servidor usando a variável `DEBUG` , ou seja, você pode rodar a aplicação com este comando:

```
DEBUG=* node app.js
```

No Windows:

```
SET DEBUG=* && node app.js
```

Agora o seu sistema está gerando logs com maior detalhe de todo comportamento da aplicação. O único problema aqui é que esses logs serão impressos na tela de console. Em um sistema em produção, seria muito tedioso ficar com console do sistema aberto para ver os logs, afinal, você também tem uma vida para viver no mundo real!

Se do nada acontecer um problema no sistema e você não estiver presente para ver o erro gerado no console, você perderá informações úteis de debug da aplicação. Para resolver esse problema, é necessário um rodar simples comando no terminal para gerar arquivos de logs. Ao executar o comando `DEBUG=* node clusters >> app.log`, o terminal para de imprimir logs na tela e passa escrever os logs dentro do arquivo `app.log`. E para simplificar ainda mais, vamos manter esse comando dentro do alias `npm start`. Edite o `package.json` na seguinte linha:

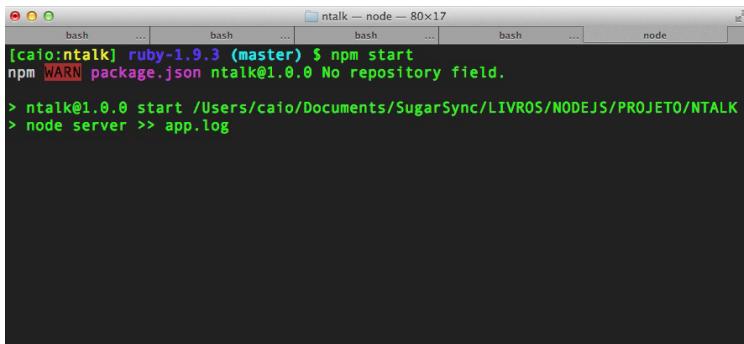
```
"scripts": {
 "start": "DEBUG=* node clusters >> app.log",
 "test": "NODE_ENV=test ./node_modules/.bin/mocha test/**/*.js"
}
```

No Windows:

```
"scripts": {
 "start": "SET DEBUG=* && node clusters >> app.log",
 "test": "SET NODE_ENV=test && ./node_modules/.bin/mocha test/**/*.js"
}
```

Agora sua aplicação está preparada para gerar arquivo de logs. Para testar as alterações, execute o comando `npm start`. Repare

que, desta vez, o terminal vai ficar congelado sem exibir nenhuma mensagem na tela. Veja a figura:

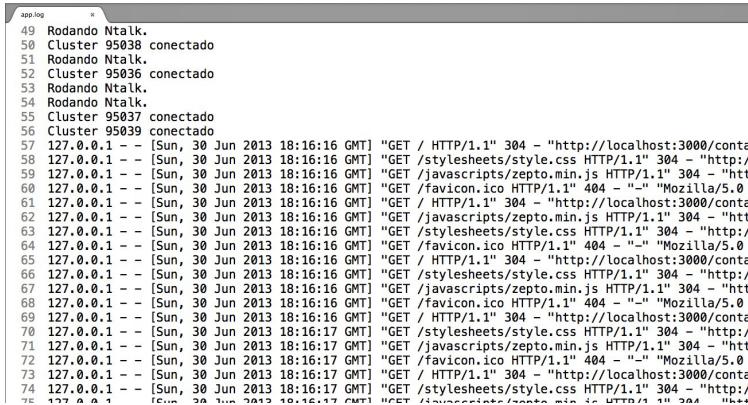


```
[caio@ntalk] ruby-1.9.3 (master) $ npm start
npm WARN package.json ntalk@1.0.0 No repository field.

> ntalk@1.0.0 start /Users/caio/Documents/SugarSync/LIVROS/NODEJS/PROJETO/NTALK
> node server >> app.log
```

Figura 9.2: Tela do terminal não emitindo logs

Em contrapartida, todas as mensagens serão persistidas dentro do arquivo `app.log`.



```
app.log
49 Rodando Ntalk.
50 Cluster 95038 conectado
51 Rodando Ntalk.
52 Cluster 95036 conectado
53 Rodando Ntalk.
54 Rodando Ntalk.
55 Cluster 95037 conectado
56 Cluster 95039 conectado
57 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET / HTTP/1.1" 304 -
58 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 -
59 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /javascripts/zepto.min.js HTTP/1.1" 304 -
60 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /favicon.ico HTTP/1.1" 404 -
61 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET / HTTP/1.1" 304 -
62 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /javascripts/zepto.min.js HTTP/1.1" 304 -
63 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 -
64 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /favicon.ico HTTP/1.1" 404 -
65 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET / HTTP/1.1" 304 -
66 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 -
67 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /javascripts/zepto.min.js HTTP/1.1" 304 -
68 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET /favicon.ico HTTP/1.1" 404 -
69 127.0.0.1 - [Sun, 30 Jun 2013 18:16:16 GMT] "GET / HTTP/1.1" 304 -
70 127.0.0.1 - [Sun, 30 Jun 2013 18:16:17 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 -
71 127.0.0.1 - [Sun, 30 Jun 2013 18:16:17 GMT] "GET /javascripts/zepto.min.js HTTP/1.1" 304 -
72 127.0.0.1 - [Sun, 30 Jun 2013 18:16:17 GMT] "GET /favicon.ico HTTP/1.1" 404 -
73 127.0.0.1 - [Sun, 30 Jun 2013 18:16:17 GMT] "GET / HTTP/1.1" 304 -
74 127.0.0.1 - [Sun, 30 Jun 2013 18:16:17 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 -
75 127.0.0.1 - [Sun, 30 Jun 2013 18:16:17 GMT] "GET /stylesheets/style.css HTTP/1.1" 304 -
```

Figura 9.3: Logs da aplicação no arquivo `app.log`

## 9.5 OTIMIZAÇÕES NO EXPRESS

Nesta seção, pretendo passar algumas dicas que visam

aumentar a performance do sistema. Serão adicionadas algumas configurações tanto para o Express como o Mongoose, com o objetivo de otimizar tanto server-side como o client-side da aplicação.

Toda a otimização será feita dentro do `app.js`, afinal, ele faz o *boot* da nossa aplicação, motivo pelo qual ele carrega e executa todos seus submódulos. Vamos começar otimizando o Express. Faremos nele 2 otimizações: habilitar compactação **Gzip** e adicionar *cache* para os arquivos estáticos incluindo dentro de `express.static` o atributo `maxAge`.

Primeiro, instale o módulo `compression` para trabalhar com Gzip:

```
npm install compression --save
```

Em seguida, edite o `app.js`, implementando as seguintes alterações:

```
const KEY = 'ntalk.sid', SECRET = 'ntalk';
var express = require('express')
, load = require('express-load')
, bodyParser = require('body-parser')
, cookieParser = require('cookie-parser')
, expressSession = require('express-session')
, methodOverride = require('method-override')
, compression = require('compression')
, error = require('./middlewares/error')
, redisAdapter = require('socket.io-redis')
, RedisStore = require('connect-redis')(expressSession)
, app = express()
, server = require('http').Server(app)
, io = require('socket.io')(server)
, cookie = cookieParser(SECRET)
, store = new RedisStore({prefix: KEY})
;
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
```

```
app.use(compression());
app.use(cookie());
app.use(expressSession({
 secret: SECRET,
 name: KEY,
 resave: true,
 saveUninitialized: true,
 store: store
}));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
app.use(methodOverride('_method'));
app.use(express.static(__dirname + '/public', {
 maxAge: 3600000 // milissegundos
}));

// continuação do app.js ...
```

Calma! Falta pouco para terminar este livro! Nas próximas páginas, continuaremos a segunda parte deste capítulo, apresentando mais dicas importantes para você preparar sua aplicação para rodar em ambiente de produção.

## CAPÍTULO 10

# APLICAÇÃO NODE EM PRODUÇÃO – PARTE 2

## 10.1 MANTENDO A APLICAÇÃO PROTEGIDA

### Prevenindo ataques XSS

Também vamos aplicar algumas boas práticas de segurança da informação para que nossa aplicação não seja alvo de vulnerabilidades básicas, que podem ser evitadas. Para isso, vamos adicionar pequenos ajustes que vão fazer uma grande diferença!

Primeiro, adicionaremos um novo *middleware* para proteger nossa aplicação contra ataques do tipo XSS (*Cross-Site Scripting*). Seu nome é `csurf` e, para instalá-lo, rode o comando:

```
npm install csurf --save
```

Com esse módulo instalado, agora teremos algumas pequenas alterações a fazer. Primeiro, edite o `app.js` e adicione esse *middleware* no último lugar com o seguinte código:

```
const KEY = 'ntalk.sid', SECRET = 'ntalk';
var express = require('express')
, load = require('express-load')
, bodyParser = require('body-parser')
, cookieParser = require('cookie-parser')
```

```

, expressSession = require('express-session')
, methodOverride = require('method-override')
, compression = require('compression')
, csurf = require('csurf')
, error = require('./middlewares/error')
, redisAdapter = require('socket.io-redis')
, RedisStore = require('connect-redis')(expressSession)
, app = express()
, server = require('http').Server(app)
, io = require('socket.io')(server)
, cookie = cookieParser(SECRET)
, store = new RedisStore({prefix: KEY})
;

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(compression());
app.use(cookie);
app.use(expressSession({
 secret: SECRET,
 name: KEY,
 resave: true,
 saveUninitialized: true,
 store: store
}));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
app.use(methodOverride('_method'));
app.use(express.static(__dirname + '/public', {
 maxAge: 3600000 // milissegundos
}));
app.use(csurf());
app.use(function(req, res, next) {
 res.locals._csrf = req.csrfToken();
 next();
});

// continuação do app.js ...

```

Após essa inclusão do `csurf`, a cada requisição realizada por qualquer rota de nossa aplicação (exceto as rotas de arquivos estáticos) será gerada uma variável local para as views. Isto acontece através do código `res.locals._csrf`, que será visível

pelas views por meio da chamada `<%-_csrf %>`. Com isso, será possível adicionar uma nova tag em todos os formulários existentes em nosso projeto para aplicar um token de proteção contra ataques XSS.

Para aplicar esse token nos formulários, adicionaremos a tag `<input type="hidden" name="_csrf" value="<%- _csrf %>">`, para que a cada submissão de formulário o servidor valide esse token.

Então vamos lá! Abra o arquivo `views/home/index.ejs` e faça a seguinte alteração:

```
<% include ../header %>
<header>
 <h1>Ntalk</h1>
 <h4>Bem-vindo!</h4>
</header>
<section>
 <form action="/entrar" method="post">
 <input type="hidden" name="_csrf" value="<%- _csrf %>">
 <input type="text" name="usuario[nome]" placeholder="Nome">

 <input type="text" name="usuario[email]" placeholder="E-mail">

 <button type="submit">Entrar</button>
 </form>
</section>
<% include ../footer %>
```

Faça o mesmo no arquivo `views/contatos/index.ejs`:

```
<% include ../header %>
<header>
 <h2>Ntalk - Agenda de contatos</h2>
</header>
<section>
 <form action="/contato" method="post">
 <input type="hidden" name="_csrf" value="<%- _csrf %>">
```

```

<input type="text" name="contato[nome]"
 placeholder="Nome">
<input type="text" name="contato[email]"
 placeholder="E-mail">
<button type="submit">Cadastrar</button>
</form>
<!-- continuação da view ... -->

```

Aplique a mesma técnica também no arquivo views/contatos/show.ejs :

```

<% include ../header %>
<header>
 <h2>Ntalk - Dados do contato</h2>
</header>
<section>
 <form action="/contato/<%- contato._id %>?_method=delete"
 method="post">
 <input type="hidden" name="_csrf" value="<%- _csrf %>">
 <p><label>Nome:</label><%- contato.nome %></p>
 <p><label>E-mail:</label><%- contato.email %></p>
 <p>
 <button type="submit">Excluir</button>
 <a href="/contato/<%- contato._id %>/editar">Editar
 </p>
 </form>
</section>
<% include ../exit %>
<% include ../footer %>

```

Para finalizar, edite também o código views/contatos/edit.ejs :

```

<% include ../header %>
<header>
 <h2>Ntalk - Editar contato</h2>
</header>
<section>
 <form action="/contato/<%- contato._id %>?_method=put"
 method="post">
 <input type="hidden" name="_csrf" value="<%- _csrf %>">
 <label>Nome:</label>
 <input type="text" name="contato[nome]">

```

```

 value="<%- contato.nome %>">
<label>E-mail:</label>
<input type="text" name="contato[email]"
 value="<%- contato.email %>">
<button type="submit">Atualizar</button>
</form>
</section>
<% include .../exit %>
<% include .../footer %>
```

Pronto! Para testar essas alterações, reinicie o servidor e acesse a página inicial da aplicação: <http://localhost:3000>. Para conferir se deu certo, basta inspecionar o código-fonte do formulário de login, verificando se a tag de csrf gerou um token aleatório semelhante ao da figura a seguir:

```

<!DOCTYPE html>
▼ <html>
 ▶ <head>...</head>
 ▼ <body>
 ▶ <header>...</header>
 ▼ <section>
 ▼ <form action="/entrar" method="post">
 <input type="hidden" name="_csrf" value="Y220ieFG-ALozZSJ9W4z0FKyx0kZYxH-qEJE">
 <input type="text" name="usuario[name]" placeholder="Nome">

 <input type="text" name="usuario[email]" placeholder="E-mail">

 <button type="submit">Entrar</button>
 </form>
 </section>
 ▶ <footer>...</footer>
 </body>
</html>
```

Figura 10.1: Token CSRF contra ataque XSS

## Removendo o header X-Powered-By da requisição

Uma técnica extremamente simples de aplicar, que a primeira vista pode ser algo muito bobo de se preocupar, é a remoção do *header* X-Powered-By das requisições da aplicação. Este *header* apenas informa qual é o nome do servidor que a aplicação está sendo hospedada.

No nosso caso, estamos usando o servidor Express, e apenas

essa informação é o suficiente para um hacker experiente ter noções iniciais de como procurar brechas em sua aplicação. Afinal, o Express é um framework open-source, e nada vai impedir que ele explore possíveis falhas de segurança em seu sistema.

Para remover esse *header*, basta apenas adicionar a função `app.disable('x-powered-by')` no início do carregamento dos middlewares do Express, assim como será apresentado a seguir como fazer esta alteração no `app.js`:

```
// carregamento dos módulos ...

app.disable('x-powered-by');

// continuação do app.js ...
```

## 10.2 EXTERNALIZANDO VARIÁVEIS DE CONFIGURAÇÕES

Para deixar mais *clean* nosso `app.js`, vamos adotar uma boa prática de externalizar variáveis de configurações da nossa aplicação. Essa prática visa centralizar em um único arquivo variáveis de chaves secretas, senhas, URL de hosts de banco de dados, e outros itens que visam configurar a aplicação. Ela segue o mesmo conceito das variáveis de sistema operacional, dessa forma fica fácil alterar dados de configuração do nosso servidor sem precisar mexer em código.

Vamos migrar as variáveis de configurações para um arquivo externo de formato `json`. Fazendo isso, será possível carregar esse arquivo através da função `require` (isso mesmo! Além de carregar módulos JavaScript, a função `require` também carrega arquivos de formato `json`).

Para isso, crie na raiz do projeto o arquivo config.json com os seguintes itens:

```
{
 "SECRET": "Ntalk",
 "KEY": "ntalk.sid",
 "MONGODB": {
 "test": "mongodb://localhost:27017/ntalk_test",
 "development": "mongodb://localhost:27017/ntalk"
 },
 "REDIS": {
 "host": "localhost",
 "port": 6379
 },
 "CACHE": {
 "maxAge": 3600000
 }
}
```

Agora, voltando no app.js , carregaremos o config.json para utilizar seus atributos no cookie, sessão e cache estático:

```
var express = require('express')
, cfg = require('./config.json')
, load = require('express-load')
, bodyParser = require('body-parser')
, cookieParser = require('cookie-parser')
, expressSession = require('express-session')
, methodOverride = require('method-override')
, error = require('./middlewares/error')
, compression = require('compression')
, csurf = require('csurf')
, redisAdapter = require('socket.io-redis')
, RedisStore = require('connect-redis')(expressSession)
, app = express()
, server = require('http').Server(app)
, io = require('socket.io')(server)
, cookie = cookieParser(cfg.SECRET)
, store = new RedisStore({prefix: cfg.KEY})
;
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(compression());
```

```

app.use(cookie);
app.use(expressSession({
 secret: cfg.SECRET,
 name: cfg.KEY,
 resave: true,
 saveUninitialized: true,
 store: store
}));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
app.use(methodOverride('_method'));
app.use(express.static(__dirname + '/public', cfg.CACHE));
app.use(csrf());
app.use(function(req, res, next) {
 res.locals._csrf = req.csrfToken();
 next();
});

```

Também utilizaremos o config.json no trecho de código responsável por configurar o Socket.IO no app.js :

```

io.adapter(redisAdapter(cfg.REDIS));
io.use(function(socket, next) {
 var data = socket.request;
 cookie(data, {}, function(err) {
 var sessionID = data.signedCookies[cfg.KEY];
 store.get(sessionID, function(err, session) {
 if (err || !session) {
 return next(new Error('not authorized'));
 } else {
 socket.handshake.session = session;
 return next();
 }
 });
 });
});

```

Para finalizar, adotaremos essa prática em libs/db\_connect.js , fazendo as seguintes modificações:

```

module.exports = function() {
 var mongoose = require('mongoose')
 , config = require('../config.json')

```

```
, env = process.env.NODE_ENV || "development"
, url = config.MONGODB[env]
;
return mongoose.connect(url);
};
```

## 10.3 APPLICANDO SINGLETON NAS CONEXÕES DO MONGOOSE

No Mongoose, apenas aplicaremos o *design pattern Singleton* no carregamento dessa biblioteca de conexão com banco de dados. Isso será implementado com o objetivo de garantir que apenas uma única conexão seja instanciada e compartilhada por toda aplicação. No arquivo `libs/db_connect.js`, aplique as seguintes mudanças:

```
var mongoose = require('mongoose')
, config = require('../config.json')
, env = process.env.NODE_ENV || "development"
, url = config.MONGODB[env]
, single_connection
;
module.exports = function() {
 if(!single_connection) {
 single_connection = mongoose.connect(url);
 }
 return single_connection;
};
```

## 10.4 MANTENDO O SISTEMA NO AR COM FOREVER

O Node.js é praticamente uma plataforma de baixo nível. Com ele, temos bibliotecas com acesso direto aos recursos do sistema operacional, e programamos entre diversos protocolos, como por exemplo, o protocolo HTTP.

Para trabalhar com HTTP, temos que programar como será o servidor HTTP e também sua aplicação. Quando colocamos uma aplicação Node.js em produção, diversos problemas e bugs são encontrados com o passar do tempo e, quando surge um bug grave, o servidor cai, deixando a aplicação fora do ar. De fato, programar em Node.js requer lidar com esses detalhes de servidor.

O `Forever` é uma ferramenta que surgiu para resolver esse problema de queda do servidor. Seu objetivo é monitorar um servidor realizando *pings* a cada curto período que for determinado pelo desenvolvedor. Quando ele detecta que a aplicação está fora do ar, automaticamente ele reinicia-a. Ele consegue fazer isso porque mantém seu programa de verificação do servidor rodando em *background* no sistema operacional.

Existem duas versões deste framework: uma é a versão CLI em que toda tarefa é realizada no terminal, e a outra maneira é de forma programável em código Node.js, utilizando o módulo `forever-monitor`. Este último é o mais recomendado a se usar quando sua aplicação está hospedada em um ambiente cujo terminal possui restrições de segurança que não permitem instalar programas do tipo CLI. O mecanismo do `forever-monitor` é o mesmo que o `forever`, sendo sua única diferença sua configuração ser feita via JavaScript, e ele instanciar uma aplicação Node.js via `child process`.

Sua instalação é simples, basta executar o comando:

```
npm install forever-monitor --save
```

Algo interessante do `forever-monitor` é que, além de ele reiniciar o servidor, é possível configurá-lo para gerar arquivos de logs da aplicação separados por logs do `forever` (`logFile`), logs

da aplicação ( `outFile` ) e logs de erros da aplicação ( `errFile` ).

Vamos criar um novo código chamado de `server.js` dentro do diretório raiz do projeto, e também crie um diretório vazio na raiz chamado de `logs` . Em seguida, abra o `server.js` e implemente o seguinte código:

```
var forever = require('forever-monitor');
var Monitor = forever.Monitor;

var child = new Monitor('clusters.js', {
 max: 10,
 silent: true,
 killTree: true,
 logFile: 'logs/forever.log',
 outFile: 'logs/app.log',
 errFile: 'logs/error.log'
});

child.on('exit', function () {
 console.log('O servidor foi finalizado.');
});

child.start();
```

Tudo começa quando instanciamos o objeto `forever.Monitor` . Nele, passamos dois parâmetros em seu construtor: o primeiro é o código da aplicação que desejamos executar (no nosso caso é o `cluster.js` ) e, no segundo parâmetro, passamos um objeto com os atributos de configuração do `forever-monitor` .

Em `max` , definimos o total de vezes que poderá reiniciar o servidor quando ele cair, lembrando que, ao passar deste total, sua aplicação será totalmente finalizada. Infelizmente, essa é uma limitação do `forever-monitor` , pois o `forever` , por ser um CLI, permite reiniciar infinitamente sua aplicação.

O atributo `silent` apenas oculta a exibição de logs no terminal. Ao habilitar o atributo `killTree`, todos os processos filhos da sua aplicação serão finalizados a cada `restart` do servidor.

Agora que temos o `forever-monitor` configurado e gerando logs por conta própria, vamos atualizar o comando `npm start` para que ele execute diretamente o `server.js` em vez do atual `clusters.js`. Com base no código seguinte, edite o seu `package.json`:

```
"scripts": {
 "start": "DEBUG=* node server",
 "test": "NODE_ENV=test ./node_modules/.bin/mocha test/**/*.js"
}
```

ou se você estiver no Windows, faça o seguinte:

```
"scripts": {
 "start": "SET DEBUG=* && node server",
 "test": "SET NODE_ENV=test && ./node_modules/.bin/mocha test/**/*.js"
}
```

Essa foi uma configuração básica para utilizar o `forever-monitor`. Caso sua necessidade seja ir além do que foi apresentado aqui, visite o GitHub oficial dos projetos, em <https://github.com/nodejitsu/forever> e em <https://github.com/nodejitsu/forever-monitor>.

Cada um possui suas vantagens e desvantagens, basta saber qual alternativa terá melhor resultado de acordo com o ambiente em que será hospedado sua aplicação Node.js. Vale lembrar que, em ambientes limitados que não permitem instalar CLIs, a melhor alternativa é usar o `forever-monitor`.

Agora, nossa aplicação está otimizada para o ambiente de

produção. No próximo capítulo, faremos uma integração interessante com o servidor Nginx, que é considerado um ótimo servidor de arquivos estáticos.

# NODE.JS E NGINX

## 11.1 SERVINDO ARQUIVOS ESTÁTICOS DO NODE.JS USANDO O NGINX

Enfim, estamos no último capítulo técnico. Durante todo o percurso implementamos uma aplicação Node.js que utiliza o web framework Express, implementamos um meio de persistência de dados usando o MongoDB e também o Redis, e também criamos um meio de nossa aplicação interagir em tempo-real através de comunicação bidirecional com o Socket.IO.

Também configuramos clusters, logs e codificamos testes de aceitação utilizando o Mocha com Supertest. Em especial, tivemos dois capítulos dedicados ao Express; afinal, ele é a base principal da nossa aplicação, com ele desenvolvemos rotas e incluímos diversos middlewares que visam otimizar o fluxo do servidor da nossa aplicação. Nesta seção, vamos integrar o Node.js com o servidor HTTP Nginx.



Figura 11.1: Servidor Nginx

O objetivo dessa integração é aumentar performance da aplicação, por isso criaremos um *proxy* do Nginx com Node.js. Também delegaremos todo processamento de arquivos estático para o Nginx, deixando apenas que o Node.js cuide do processamento de suas rotas. Isso visa diminuir o número de requisições diretas em nossa aplicação.

## ATENÇÃO

Não entraremos em detalhes sobre como instalar o Nginx em sua máquina. Para instalá-lo, recomendo que acesse seu site oficial (<http://nginx.org>). Também recomendo que leia sua *Wiki* que contém diversas dicas de como configurá-lo (<http://wiki.nginx.org/Main>).

A versão usada neste livro é a 1.5.2 . Recomendo que **não utilize versões anteriores** a esta, pois é provável que não funcione a dica de configuração que explicarei adiante.

Outro detalhe importante é que foi a partir da versão 1.3.13 que o Nginx passou a dar suporte ao protocolo WebSocket, e nisso o seu servidor estará habilitado para trabalhar com requisições do WebSocket que são geradas pelo Socket.IO.

Agora que temos o Nginx instalado e funcionando corretamente em sua máquina, vamos configurá-lo para que ele comece a servir arquivos estáticos de nossa aplicação. Tudo isso será feito dentro de seu arquivo principal chamado `nginx.conf` . A localização desse arquivo varia de acordo com o sistema operacional, então, recomendo que leia sua documentação oficial (<http://nginx.org/en/docs>) para descobrir onde ele se encontra.

A seguir, apresento uma versão simplificada de configuração do Nginx. Esta configuração fará o Nginx servir os arquivos estáticos em vez do Express. Para finalizar, aplicamos um *proxy* do Nginx para as rotas da nossa aplicação.

```

worker_processes 1;

events {
 worker_connections 1024;
}

http {
 include mime.types;
 default_type application/octet-stream;
 sendfile on;
 keepalive_timeout 65;
 gzip on;

 server {
 listen 80;
 server_name localhost;
 access_log logs/access.log;

 location ~ ^/(javascripts|stylesheets|images) {
 root /var/www/ntalk/public;
 expires max;
 }

 location / {
 proxy_pass http://localhost:3000;
 }
 }
}

```

Praticamente adicionamos algumas melhorias em cima das configurações padrões do `nginx.conf`. Com o objetivo de otimizar o servidor estático, habilitamos compactação `gzip` nativa, através do trecho `gzip on;`, e criamos dois `locations` dentro de `server`. O primeiro `location` é o responsável por servir conteúdo estático.

```

location ~ ^/(javascripts|stylesheets|images) {
 root /var/www/ntalk/public;
 expires max;
}

```

É dentro dele que definimos a localização da pasta `public` da

nossa aplicação através do trecho `root /var/www/ntalk/public;` . Essa localização definida no item `root` baseia-se no endereço onde fica a pasta `public` do projeto em seu sistema operacional, seja ele Unix, MacOS ou Linux.

Se o seu sistema é Windows, altere o endereço para o padrão de diretórios dele, que é algo semelhante a `root C:/www/ntalk/public` , ou para qualquer outro endereço em que você deseja manter esse projeto.

Também aplicamos dentro desse `location` um *cache* dos arquivos através do item `expires max;` .

No último `location` , aplicamos um simples controle de *proxy*. O item `proxy_pass` praticamente redireciona as demais rotas para nossa aplicação, que estará ativa através do endereço `http://localhost:3000` .

```
location / {
 proxy_pass http://localhost:3000;
}
```

Com o Nginx já configurado e rodando, que tal testar essa integração? Reinicie o Nginx por meio do comando `nginx -s reload` , e também nossa aplicação, via comando `npm start` . Se até agora nenhum problema aconteceu, basta acessar sua aplicação através do novo endereço: `http://localhost` .

## CAPÍTULO 12

# CONTINUANDO OS ESTUDOS

Finalmente chegamos ao fim deste livro! Mas como esta tecnologia está constantemente em evolução, com certeza é sempre bom se manter atualizado com seus novos recursos, ler e acompanhar novas referências sobre essa plataforma.

Sendo assim, listarei algumas excelentes referências para você continuar estudando mais e mais Node.js!

### Sites, blogs, fóruns e slides

- Site oficial Node.js — <http://nodejs.org>
- GitHub do Node.js — <http://github.com/joyent/node>
- Documentação do Node.js — <http://nodejs.org/api>
- Blog do Node.js — <http://blog.nodejs.org>
- NPM Registry — <http://npmjs.org>
- Blog da comunidade NodeBR — <http://nodebr.com>
- Fórum do NodeBR — <http://groups.google.com/forum/#!forum/nodebr>
- Underground WebDev (autor deste livro) — <http://udgwebdev.com>
- How to Node — <http://howtonode.org>

- Scoop.it do Node.js — <http://scoop.it/t/nodejs-code>
- TJ Holowaychuk (autor do Express) — <https://medium.com/@tjholowaychuk>
- NodeCloud — <http://nodecloud.org>
- NodeSecurity — <https://nodesecurity.io>
- DailyJS — <http://dailyjs.com>
- NetTuts+ — <http://net.tutsplus.com>
- Blog metaduck — <http://metaduck.com>
- Blog Waib — <http://www.waib.com.br/Blog>
- Blog Caelum — <http://blog.caelum.com.br>
- NodeSchool — <http://nodeschool.io>
- Blog Nomadev — <http://nomadev.com.br>
- GUJ (discuta sobre JavaScript e Node.js) — <http://guj.com.br>
- Grupo de discussão deste livro — <http://forum.casadocodigo.com.br>

## Eventos, screencasts, podcasts e cursos

- NodeSchool — <http://nodeschool.io>
- EggHead.IO — <https://egghead.io/technologies/node>
- TagTree — <http://tagtree.tv>
- Node Knockout (Hackaton Node.js inspirado em Rails Rumble) — <http://nodeknockout.com>
- DevCast – Node.js e MongoDB, o casamento perfeito — <http://youtube.com/watch?v=-OpUd1Rov2c>
- DevCast – JavaScript dos novos tempos — <http://youtube.com/watch?v=LlUTi5DM4ws>
- GrokPodcast Node.js – Parte 1 — <http://grokpodcast.com/2011/02/17/episodio-19-node-js-parte-1>

- GrokPodcast Node.js – Parte 2 —  
[http://grokpodcast.com/2011/02/24/episodio-20-nodejs-parte-2](http://grokpodcast.com/2011/02/24/episodio-20-node-js-parte-2)
- GrokPodcast Node.js – Parte 3 —  
[http://grokpodcast.com/2011/03/03/episodio-21-nodejs-parte-3](http://grokpodcast.com/2011/03/03/episodio-21-node-js-parte-3)
- Nodecasts (inspirado no Railscast) —  
<http://nodecasts.net>
- SailsCasts (screencasts do Framework Sails.js) —  
<http://irlnathan.github.io/sailscasts/blog/archives>
- Nodetuts screencasts — <http://nodetuts.com>
- Nodeup podcasts — <http://nodeup.com>
- Codeschool: real time with Node.js —  
<http://codeschool.com/courses/real-time-web-with-nodejs>
- Tuts+ Courses: building web apps with Node.js and Express — <http://tutsplus.com/course/building-web-apps-in-node-and-express>

Enfim, espero que essas referências sejam de grande utilidade.

Tenha bons estudos e obrigado por ler este livro!

## CAPÍTULO 13

# BIBLIOGRAFIA

FAUSAK, Taylor. *Testing a Node.js HTTP server with Mocha*. Disponível em: <http://taylor.fausak.me/2013/02/17/testing-a-nodejs-http-server-with-mocha/>. 2012.

GARLAPATI, Shravya. *Blazing fast node.js: 10 performance tips from LinkedIn Mobile*. Disponível em: <https://engineering.linkedin.com/nodejs/blazing-fast-nodejs-10-performance-tips-linkedin-mobile>. 2011.

HOLOWAYCHUK, T. J. *Modular web apps with Node.js and Express*. Disponível em: <http://tjholowaychuk.com/post/38571504626/modular-web-applications-with-node-js-and-express>. 2012.

KIESSLING, Manuel. *Node Beginner Book*. Leanpub, 2013.

MARDANOV, Azat. *Javascript and Node Fundamentals*. Leanpub, 2013.

MCMAHON, Caolan. *Node.js: Style and structure*. Disponível em: [http://caolanmcmahon.com/posts/nodejs\\_style\\_and\\_structure/](http://caolanmcmahon.com/posts/nodejs_style_and_structure/). 2012.

MEANS, Garann. *Node for front-end developers*. O'Reilly

Media, 2012.

MOREIRA, Rafael Henrique. *Gerando seu app automaticamente com Express em Node.js*. Disponível em: <http://nodebr.com/gerando-seu-app-automaticamente-com-express-1-em-node-js/>. 2013.

MOREIRA, Rafael Henrique. *Callbacks em Node*. Disponível em: <http://nodebr.com/callbacks-em-node/>. 2013.

MOREIRA, Rafael Henrique. *JavaScript no servidor com Node.js*. Disponível em: <http://nodebr.com/javascript-no-servidor-com-node-js/>. 2013.

OLIVEIRA, Eric. *Aprendendo Padrões de Projeto JavaScript*. Leanpub, 2013.

PEREIRA, Caio Ribeiro. *Node.js para leigos - Instalação e configuração*. Disponível em: <http://udgwebdev.com/node-js-para-leigos-instalacao-e-configuracao/>. 2012.

PEREIRA, Caio Ribeiro. *Node.js para leigos - Trabalhando com HTTP*. Disponível em: <http://udgwebdev.com/node-js-para-leigos-trabalhando-com-http>. 2012.

PEREIRA, Caio Ribeiro. *Compartilhando Sessions entre Socket.IO e Express*. Disponível em: <http://udgwebdev.com/nodejs-express-socketio-e-sessions/>. 2013.

RAUCH, Guilhermo. *Smashing Node.js: Javascript Everywhere*. Wiley, 2012.

RAUCH, Guilhermo. *NPM tricks*. Disponível em: <http://www.devthought.com/2012/02/17/npm-tricks/>. 2012.

SANFILIPPO, Salvatore. *Introduction to Redis*. Disponível em: <http://redis.io/topics/introduction>. 2011.

SENCHALABS. *Connect* - High quality middleware for node.js. Disponível em: <http://www.senchalabs.org/connect/>. 2010.

TEIXEIRA, Pedro. *Hands-on Node.js*. Leanpub, 2012.

TEIXEIRA, Pedro. *The Callback Pattern*. Disponível em: <http://nodetuts.com/02-callback-pattern.html>. 2012.

VICENT, Seth. *Making 2D Games with Node.js and Browserfy*. Leanpub, 2013.

VISIONMEDIA. *Mocha simple, flexible, fun*. Disponível em: <http://visionmedia.github.io/mocha/>. 2012.

WILSON, Jim R. *Node.js the Right Way: Practical, Server-Side JavaScript That Scales*. The Pragmatic Bookshelf, 2013.