

Node.js

Aplicações web real-time com Node.js

Edição atualizada



© 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

[2018]

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.

ISBN

Impresso e PDF: 978-85-66250-14-5

EPUB: 978-85-66250-93-0

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

AGRADECIMENTOS

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

Um agradecimento para Aline Brandariz Santos, por ser o sorriso em minha vida, por estar ao meu lado em todos os momentos, por me fazer feliz, e por me apoiar e me incentivar a arriscar na vida.

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, pois foi graças a ele que entrei nesse mundo da tecnologia. Ele foi a primeira pessoa a me apresentar ao 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 e aprender sempre. Foi uma honra trabalhar junto com eles.

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 a todos os leitores do blog *Underground WebDev* (<https://udgwebdev.com>). 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 está prestes a ler.

"Parabéns pelo Post! Adorei, muito explicativo. A comunidade brasileira agradece." – Rafael Henrique Moreira

"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

"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

"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

"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

"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

"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

"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

SOBRE O AUTOR



Figura 1: Caio Ribeiro Pereira

Empreendedor, desenvolvedor, entusiasta bitcoin, escritor e blogueiro.

Formado em bacharel de Sistemas de Informação pela Universidade Católica de Santos, é blogueiro nos tempos livres e apaixonado por programação, principalmente quando o assunto é sobre o universo JavaScript. Adora compartilhar conhecimento em seu blog, e está sempre testando e acompanhando novas tendências e tecnologias.

Site pessoal: <https://crpwebdev.github.io>

Blog: <https://udgwebdev.com>

PREFÁCIO

As mudanças do mundo web

Tudo na web trata-se 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, a necessidade dessa interação aumenta 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 a interação em tempo real entre cliente e servidor seja eficaz. E mais, isto precisa acontecer em uma escala massiva, suportando entre 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 os sistemas trabalharem em tempo real, os servidores precisam enviar e receber dados utilizando comunicação bidirecional em vez de usar intensamente requisição e resposta do modelo HTTP pelo *Ajax*. Também temos de manter esse tipo comunicação de forma leve e rápida para continuar escalável, reutilizável e com um desenvolvimento fácil de ser mantido a longo prazo.

A quem se destina este livro?

Este 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, 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.

Vale lembrar que este livro tem como pré-requisito saber programar e, principalmente, ter noções básicas de JavaScript, HTML e CSS.

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	9
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	19
2.4 Separando o HTML do JavaScript	21
2.5 Desafio: implementando um roteador de URL	24
3 Por que o assíncrono?	26

Sumário	Casa do Código
3.1 Desenvolvendo de forma assíncrona	26
3.2 Assincronismo versus sincronismo	30
3.3 Entendendo o event-loop	33
3.4 Evitando callbacks hell	35
4 Iniciando com o Express	38
4.1 Por que utilizá-lo?	38
4.2 Instalação e configuração	39
4.3 Criando um projeto de verdade	40
4.4 Gerando o scaffold do projeto	42
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	86
6.5 Compartilhando sessão entre Socket.IO e Express	87
6.6 Gerenciando salas do chat	93
6.7 Notificadores na agenda de contatos	99
6.8 Principais eventos do Socket.IO	104

7 Integração com banco de dados	106
7.1 Bancos de dados mais adaptados para Node.js	106
7.2 Instalando o MongoDB	108
7.3 MongoDB no Node.js utilizando Mongoose	110
7.4 Modelando com Mongoose	112
7.5 Implementando um CRUD na agenda de contatos	113
7.6 Persistindo estruturas de dados usando Redis	121
7.7 Mantendo um histórico de conversas do chat	122
7.8 Persistindo lista de usuários online	125
8 Preparando um ambiente de testes	128
8.1 Mocha, o framework de testes para Node.js	128
8.2 Criando um ambiente para testes	129
8.3 Instalando e configurando o Mocha	132
8.4 Rodando o Mocha no ambiente de testes	133
8.5 Testando as rotas	135
8.6 Deixando seus testes mais limpos	144
9 Aplicação Node em produção – Parte 1	148
9.1 Configurando clusters	148
9.2 Redis controlando as sessões da aplicação	152
9.3 Monitorando aplicação por meio de logs	155
9.4 Otimizações no Express	158
10 Aplicação Node em produção – Parte 2	161
10.1 Mantendo a aplicação protegida	161
10.2 Mantendo o sistema no ar com Forever	167
10.3 Externalizando variáveis de configurações	170

11 Node.js e Nginx	175
11.1 Servindo arquivos estáticos do Node.js usando o Nginx	175
12 Continuando os estudos	180
13 Bibliografia	182

Versão: 21.8.13

CAPÍTULO 1

BEM-VINDO AO MUNDO NODE.JS

1.1 O PROBLEMA DAS ARQUITETURAS BLOQUEANTES

Os sistemas para web desenvolvidos nativamente sobre plataforma **.NET**, **Java**, **PHP**, **Ruby** ou **Python** 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 uma fila de requisições ociosas por um período de tempo.

Esta é uma arquitetura clássica, existente em diversos sistemas e que possui um design ineficiente. É gasta 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 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 criou o Node.js, com a ajuda inicial de 14 colaboradores. 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, já que 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*, porque o Node.js trabalha apenas em *single-thread* (única thread por processo). 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 baseados 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 uma terá uma 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, podemos usar *clusters* (assunto a ser explicado na seção *Configurando clusters*, no capítulo *Aplicação Node em produção – Parte 1*), módulos nativos do Node.js e muito fáceis 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. Neles,

serão executadas funções em *background*, em paralelo, que aguardam o retorno por meio de funções de *callback*. E tudo isso é trabalhado de forma não bloqueante.

1.4 EVENT-LOOP

Node.js é orientado a eventos e 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: 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 foi emitido. Quando ocorre, é emitido um evento. Ele executa-o e envia-o para a fila de executados. Quando um evento está em execução, 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) e Twisted (do Python). Porém, o *event-loop* do Node é mais performático, porque 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 usar, as dicas serão as mesmas. É claro que os procedimentos serão diferentes para cada sistema (principalmente para o Windows, mas não será nada grave).

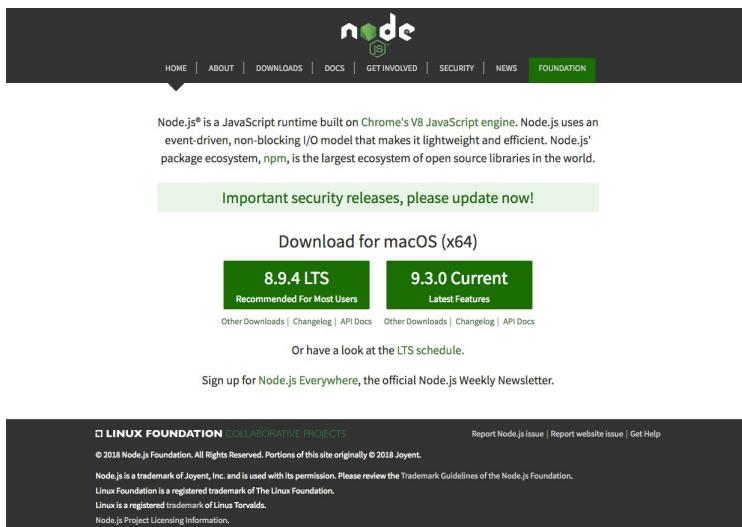
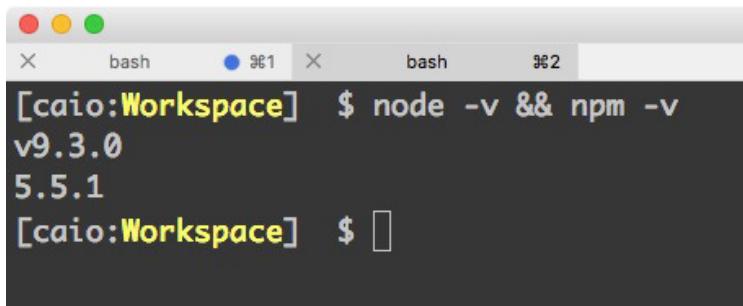


Figura 1.2: Página de download do Node.js

Instalando Node.js

O primeiro passo é acessar o site oficial (<https://nodejs.org>) e clicar no botão **v9.3.0 Current**. Para usuários do *Windows* e *MacOS*, basta baixar os seus instaladores e executá-los normalmente. Já para quem já usa *Linux* com *Package Manager* instalado, acesse <https://nodejs.org/en/download/package-manager>, referente às instruções sobre como instalá-lo em diferentes sistemas.

Instale o Node.js de acordo com seu sistema e, caso não ocorram 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.

A screenshot of a macOS terminal window titled "bash". It shows two tabs: tab 1 is active and titled "bash", while tab 2 is titled "bash". The command `[caio:Workspace] $ node -v && npm -v` is entered and its output is displayed: "v9.3.0" and "5.5.1".

```
[caio:Workspace] $ node -v && npm -v
v9.3.0
5.5.1
[caio:Workspace] $
```

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

A última versão estável usada neste livro é **Node 9.3.0**, junto do **NPM 5.5.1**.

DICA

Todo o conteúdo deste livro será compatível com versões do Node.js **6.0.0** ou superiores.

Configurando o ambiente de desenvolvimento

Para configurá-lo, basta adicionar uma variável de ambiente `NODE_ENV` ao sistema operacional. Em sistemas Linux ou OSX, basta acessar o arquivo `.bash_profile` ou `.bashrc` com um editor de texto qualquer e em modo *superuser* (`sudo`), então

adicionar o seguinte comando: export NODE_ENV='development' .

No Windows 7, o processo é um pouco diferente:

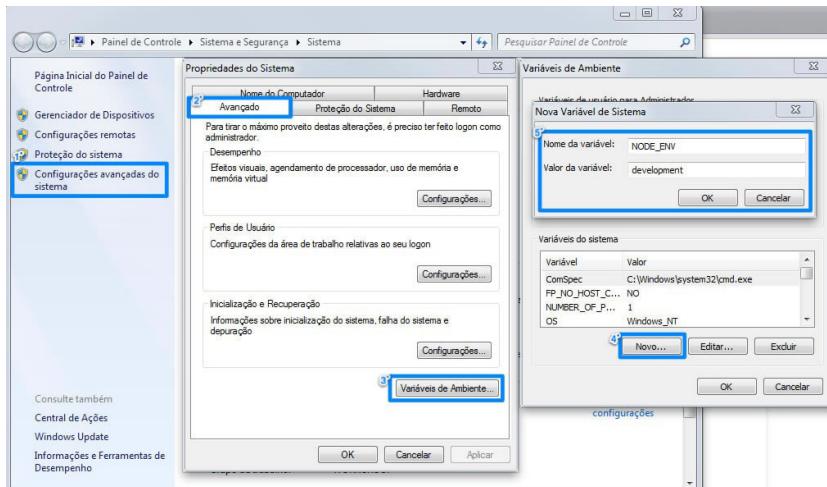


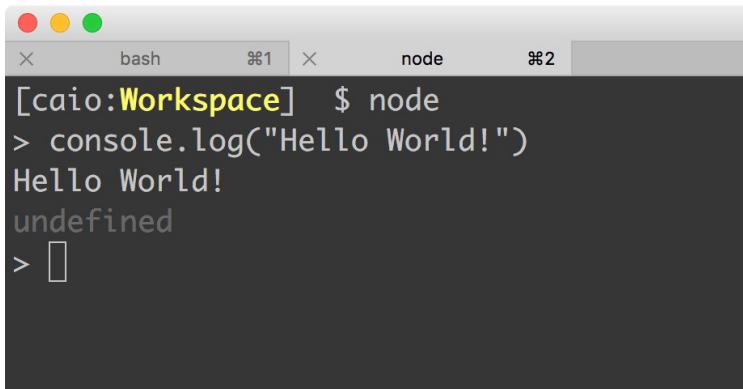
Figura 1.4: Configurando a variável NODE_ENV no Windows 7

Clique com o botão direito no ícone Meu Computador e selecione a opção Propriedades . Ao 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 em 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 macOS terminal window. The window title bar shows three colored dots (red, yellow, green) on the left, followed by the word "bash" in a grey font, then two tabs: "⌘1" and "node". The "node" tab is currently active, indicated by a grey background. The main terminal area displays the following text:

```
[caio:Workspace] $ node
> console.log("Hello World!")
Hello World!
undefined
> []
```

The text is white on a dark grey background. The cursor is visible at the end of the last line.

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 tornou-se tão popular pela comunidade, que foi a partir da versão **0.6.0** do Node.js que se integrou ao instalador, tornando-se o gerenciador *default*. Isso simplificou a vida dos desenvolvedores na época, pois fez diversos projetos convergirem para essa plataforma.

Não listarei todos os comandos, mas apenas os principais para que você tenha noção 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, pelo 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 os

termos *módulo*, *biblioteca* e *framework*. 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 descriptor, 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, 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 o e-

mail do autor. Utilize o formato `Nome <email>` para que sites (como <https://npmjs.org>) reconheçam corretamente esses dados.

Outro atributo principal é o `version`, com o qual definimos a versão atual do módulo. É extremamente recomendado que se 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 ao projeto. O primeiro, o `modulo-1`, somente será incluído em sua versão fixa, a `1.0.0`. Utilize este tipo de versão para instalar dependências cujas 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 `~` (til), 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 maiores ou iguais 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. Isso comprometerá seu funcionamento e exigirá que você atualize todo o código até voltar ao normal.

O último, o `modulo-4`, usa o caractere `*` (asterisco), e 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 usar `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 existe na maioria das linguagens de programação, assim como seu uso. 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 traz um módulo e, para criar um código JavaScript que seja modular e que possa ser carregado pelo `require`, usam-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 = (msg) => {
  console.log(msg);
};
```

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

```
exports.hello = (msg) => {
  console.log(msg);
};
```

A diferença entre o `hello.js` e o `human.js` está na maneira como eles serão carregados. Em `hello.js`, instanciamos uma única função para carregar todo o seu módulo. Já em `human.js`, é carregado um objeto com funções modulares, sendo estas carregadas individualmente. Essa é a grande diferença entre eles.

Para entender melhor na prática, crie o código `app.js` para carregar esses módulos:

```
const hello = require('./hello');
const 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, foram executados: a função `hello()`, que imprimiu a mensagem `Olá pessoal!`; e o objeto `human`, que executou sua função `human.hello('Olá galera!')`.

Perceba 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 que são disponíveis por meio de módulos não oficiais criados pela comunidade.

Um dos mais usados 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, vamos desenvolver um sistema web usando o módulo nativo **HTTP**, para mostrar em baixo-nível como tratar roteamento e carregamento das páginas. Com isso, também mostraremos as vantagens e desvantagens de trabalhar usando módulo nativo *versus* um framework web. Vamos apresentar 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 sua infraestrutura.

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 o sistema em detalhes, permitindo desenvolver algo performático (ou não), mas algo mais controlado pelo programador por trabalhar diretamente com protocolo HTTP.

Caso sua aplicação seja grande e você precise de um framework com diversas facilidades prontas para serem usadas, ou não haja a necessidade de trabalhar no baixo-nível (programando diretamente no protocolo HTTP), recomendo que use algum framework web que já venha com o mínimo necessário de configurações prontas. Assim, você não perderá tempo trabalhando com tudo isso. Alguns módulos conhecidos são:

- **Express** – <https://expressjs.com>
- **CompoundJS** – <https://compoundjs.com>
- **Sails** – <https://sailsjs.com>

Esses módulos já são preparados para lidar com uma infraestrutura mínima ou até uma mais completa, permitindo desenvolver facilmente aplicações *RESTful/WebSocket*, adotando um padrão MVC (*Model-View-Controller*) e, em alguns casos, até possuem integrações com drivers de alguns bancos de dados.

Primeiro, usaremos apenas o módulo nativo **HTTP**, pois

precisamos entender todo o seu conceito, visto que todos os frameworks citados o utilizam como estrutura base em seus projetos. Isso porque esses frameworks são voltados para a construção de aplicações web, e o protocolo HTTP/HTTPS é a base essencial para esse tipo de aplicação. A seguir, mostro uma clássica aplicação *Hello World*.

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

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

const server = http.createServer((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 um resultado em formato HTML pela **rota raiz /**, com a mensagem `Hello World!`.

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

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 `(request, response) => {}` é executado apenas quando o servidor recebe uma requisição. Para

isso, o *event-loop* verifica constantemente se o servidor foi requisitado e, quando recebe uma requisição, ele emite um evento para que seja executado o callback.

O Node.js trabalha muito com chamadas assíncronas que respondem por meio de callbacks do JavaScript. Por exemplo, se quisermos notificar que o servidor está de pé, mudamos a linha `server.listen` para receber no parâmetro uma função que faz esse aviso:

```
server.listen(3000, () => {
  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 ter de passar como parâmetro uma função 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:

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

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

const servidorLigou = () => {
  console.log('Servidor Hello World rodando!');
}
```

```
server.listen(3000, servidorLigou);
```

2.3 TRABALHANDO COM DIVERSAS ROTAS

Até agora respondemos apenas ao endereço / , mas queremos possibilitar que nosso servidor também responda a outros. Utilizando um palavreado comum entre desenvolvedores Rails, vamos adicionar novas **rotas**, para entender mais a fundo como organizar o roteamento de conteúdo de múltiplas páginas.

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

```
const http = require('http');
const server = http.createServer((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, () => {
  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?

Repare na complexidade do nosso código: o roteamento foi

tratado pelos comandos `if` e `else`. Já a leitura de URL é obtida por meio 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 de grande porte, torna-se cansativo e inviável tratar todas as URLs dessa maneira, principalmente se você precisar construir um tratamento de rotas dinâmicas. Afinal, o código ficaria gigante com diversos `ifs` e `elses` tratando cada caso específico, sendo que existem maneiras melhores de se resolver isso com menos código.

No Node.js, existe o módulo nativo chamado `url`, responsável por fazer *parser* e formatação de URLs. Vamos usá-lo para capturar informações de *query string* que o usuário informar e, no final, mostrar esses dados na tela. Para isso, crie o novo arquivo `url_server.js` com o seguinte código:

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

const server = http.createServer((request, response) => {
  response.writeHead(200, {'Content-Type': 'text/html'});
  response.write('<h1>Dados da query string</h1>');
  const 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, () => {
  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`) . Por meio do retorno da função `url.parser()` , esse módulo identifica 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 os 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 a âncora da URL: `#hash`

Resumindo, o módulo URL permite capturar diversas informações a respeito das URLs da aplicação. Com base nele, torna-se viável tratar o roteamento de páginas, além de pegar informações via path ou query string. Ela é usada como base em diversos frameworks web, pois auxilia no tratamento de rotas de uma aplicação web.

2.4 SEPARANDO O HTML DO JAVASCRIPT

Agora precisamos organizar os códigos HTML. 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*), 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 próximo capítulo, explicaremos melhor as diferenças, vantagens e desvantagens de funções assíncronas *versus* síncrona.

No exemplo a seguir, apresento duas maneiras de ler um arquivo usando o *File System*:

```
const fs = require('fs');
fs.readFile('/index.html', (erro, arquivo) => {
  if (erro) throw erro;
  console.log(arquivo);
});
const 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 os sistemas Linux, OS X e Unix, e outras são apenas para Windows.

Para melhores informações, leia sua documentação em <https://nodejs.org/dist/latest-v8.x/docs/api/fs.html>.

Voltando ao desenvolvimento da nossa aplicação, usaremos a função `fs.readFile()` para renderizar o HTML de forma assíncrona. Dessa forma, será possível separar conteúdo HTML do JavaScript, cada um em seu devido arquivo. Crie um novo arquivo, chamado `site_pessoal.js`, com o seguinte código:

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

const server = http.createServer((request, response) => {
  // __dirname retorna o diretório raiz da aplicação
  fs.readFile(`$__dirname__/index.html`, (erro, html) => {
    response.writeHead(200, {'Content-Type': 'text/html'});
    response.write(html);
    response.end();
  });
});
server.listen(3000, () => {
  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 este 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 o path no browser, `/artigos` deve renderizar `artigos.html`;
- A regra anterior também se aplica ao 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. Use a função `fs.existsSync(html)` para verificar se existe o HTML com o mesmo nome do `pathname` digitado.

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

CAPÍTULO 3

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 como e quando elas são invocadas. Isso porque ela traz a vantagem de não deixar a CPU ociosa quando faz uma operação de I/O.

O código a seguir exemplifica as diferenças entre uma função síncrona e assíncrona em relação à linha do tempo na qual elas são executadas. Isso vai mostrar na prática o quanto de tempo é economizado ao realizar uma operação de I/O com Node.js. 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, responsável por carregar de forma síncrona, ou seja, carregamento totalmente sequencial dos arquivos. Assim, ao terminar a leitura completa de um arquivo, será realizada a leitura do próximo até terminar a execução desse código. Para isso, crie o arquivo `text_sync.js` com o código a seguir:

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

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

Agora vamos criar o arquivo `text_async.js`, com o seu respectivo código, diferente apenas na forma de chamar a função `writeFileSync`. Esta será a versão assíncrona `writeFile`. Nele todo carregamento será assíncrono e, com isso, múltiplos carregamentos serão realizados ao mesmo tempo. Isso otimiza o uso de CPU enquanto ocorre uma execução de I/O dos arquivos. Veja como o código ficará:

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

for (let i = 1; i <= 5; i++) {
  const file = `async-txt${i}.txt`;
  fs.writeFile(file, 'Hello Node.js!', (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:

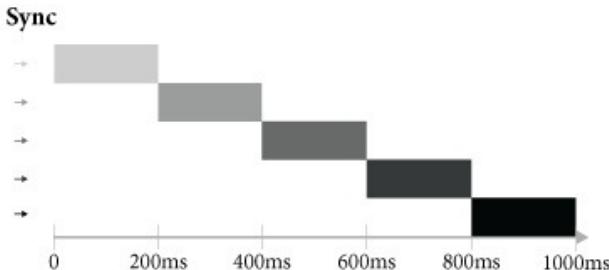


Figura 3.1: Timeline síncrona bloqueante

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:

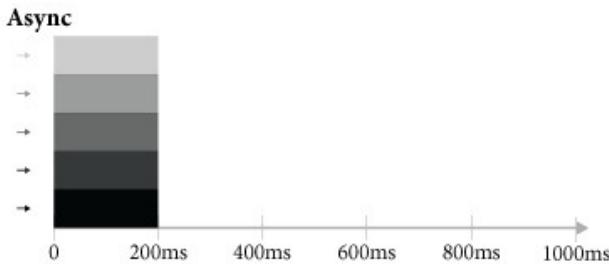


Figura 3.2: Timeline assíncrona não bloqueante

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

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, 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 ter pouco domínio sobre elas 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 suas funções em paralelo sem travar o 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 (funções síncronas) para se obter uma performance melhor no uso de CPU, já que funções síncronas de I/O deixam a CPU ociosa, e há momentos em uma aplicação em que a CPU poderia realizar outras tarefas em paralelo.

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, ela vai bloquear o sistema inteiro, deixando o processador ocioso enquanto ele usa outros recursos de hardware – como leitura em disco, utilização da rede etc.

Fazendo uma analogia com uma aplicação web, se você criar uma rota que carrega um arquivo de conteúdo pesado de forma síncrona, caso múltiplos usuários consumam essa mesma rota, a CPU do servidor ficará ociosa. Nesse caso, o primeiro usuário vai esperar e receber tal conteúdo, enquanto os demais ficarão em uma espera maior, até que termine o processamento do primeiro. Já na situação de realizar um *I/O* assíncrono desse conteúdo pesado, a CPU teria uma maior autonomia ao tratar o carregamento desse conteúdo para múltiplos usuários ao mesmo tempo.

Sempre que puder, use funções assíncronas para aproveitar essa característica principal do Node.js. Caso você ainda não esteja convencido, a próxima seção lhe mostrará como e quando usar bibliotecas assíncronas, tudo por meio de um 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**. Com isso, será possível entender melhor quanto tempo a CPU ficou ociosa esperando o carregamento de um arquivo pesado (operação de *I/O*).

Para isso, crie 3 arquivos: `processamento.js` , `leitura_async.js` e `leitura_sync.js` . Criaremos isoladamente o código `leitura_async.js` , que faz a leitura

assíncrona:

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

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

```
const fs = require('fs');
const leituraSync = (arquivo) => {
  console.log('Fazendo leitura síncrona');
  const inicio = new Date().getTime();
  fs.readFileSync(arquivo);
  const fim = new Date().getTime();
  console.log(`Bloqueio síncrono: ${fim - inicio}ms`);
};
module.exports = leituraSync;
```

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

```
const http = require('http');
const fs = require('fs');
const leituraAsync = require('./leitura_async');
const leituraSync = require('./leitura_sync');
const arquivo = './node.exe';
const stream = fs.createWriteStream(arquivo);
const download = 'http://nodejs.org/dist/latest/node.exe';
http.get(download, (res) => {
  console.log('Fazendo download do Node.js');
  res.on('data', (data) => stream.write(data));
  res.on('end', () => {
    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 `const conteudo = fs.readFileSync()`. Mas qual é o problema dessa abordagem? **Elá 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:

- **Processador:** Core i5 2.0GHz
- **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*, ele terá o mesmo bloqueio perceptível pelo usuário. A diferença é que um servidor estará lidando com milhares de usuários requisitando I/O, com a grande probabilidade de ser ao mesmo tempo.

É por isso que o Node.js trabalha com o 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>). Elas são bibliotecas internas responsáveis por prover o processamento de I/O assíncrono 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 verifica se existem novos eventos em sua **fila de eventos**, em cada iteração. 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 herda suas funcionalidades de evento desse módulo. 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 a sua aplicação mais robusta e estruturada para lidar com eventos que são executados de forma assíncrona não bloqueante. Para conhecer mais sobre as funcionalidades do `EventEmitter`, acesse sua documentação, em <https://nodejs.org/dist/latest-v8.x/docs/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 por meio das suas funções de callback. Assim, tal processamento terá uma sequência de execução. No código a seguir, apresentarei um exemplo desse caso.

Crie um arquivo chamado `callback_hell.js`. Nele vamos implementar um código com excesso de callbacks sequenciais para exemplificar esse problema:

```
const fs = require('fs');
const path = require('path');

fs.readdir(__dirname, (erro, contents) => {
  if (erro) { throw erro; }
  contents.forEach((content) => {
    const dir = path.join(__dirname, content);
    fs.stat(dir, (erro, stat) => {
```

```
    if (erro) { throw erro; }
    if (stat.isFile()) {
      console.log('%s %d bytes', content, stat.size);
    }
  });
});
});
```

Repare na quantidade de callbacks encadeados que existe 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? O seu código seria praticamente 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 um erro for identificado.

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

```
const fs = require('fs');
const path = require('path');

const ler = (arquivo) => {
  const dir = path.join(__dirname, arquivo);
  fs.stat(dir, (erro, stat) => {
    if (erro) return erro;
    if (stat.isFile()) {
      console.log('%s %d bytes', arquivo, stat.size);
    }
  });
};

const lerDiretorio = () => {
  fs.readdir(__dirname, (erro, diretorio) => {
    if (erro) return erro;
    diretorio.forEach((arquivo) => ler(arquivo));
  });
};

lerDiretorio();
```

Veja o quanto melhorou a legibilidade do seu código. Dessa forma, deixamos o nome das funções mais semântico e legível, 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

A partir deste capítulo até o final do livro, vamos construir na prática uma aplicação web usando Node.js. Com isso, explicaremos linha por linha o que faz cada código e, principalmente, boas práticas de estruturação de projetos.

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 <https://expressjs.com>.

express

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 baseada em variáveis de ambiente;
- Suporte a *helpers* dinâmicos;
- Integração com *Template Engines*;
- Integração com SQL e NoSQL.

4.2 INSTALAÇÃO E CONFIGURAÇÃO

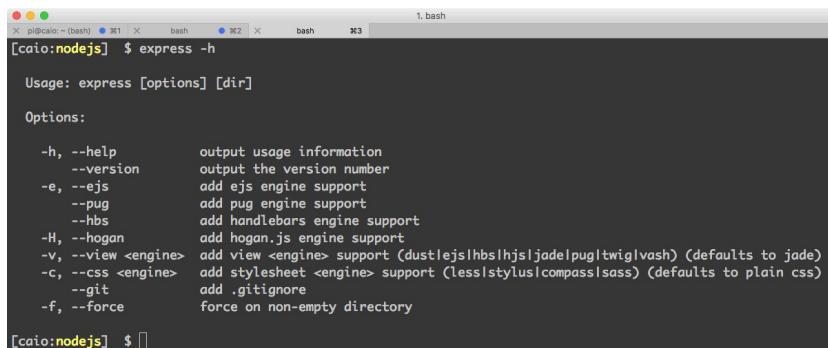
Instalar o Express é muito simples e existem meios de criar um projeto Express com alguns recursos pré-configurados, visando economizar nosso tempo ao organizar esses detalhes. Para aproveitar disso, opcionalmente, recomendo que instale um gerador de projeto `express` , que se chama `express-generator` :

```
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 o Hogan) e um CSS engine (por padrão, ele utiliza CSS puro, mas é possível escolher LESS, Stylus ou Compass).

Para visualizar todas as opções, execute o comando `express -h`.



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

Options:
  -h, --help           output usage information
  --version           output the version number
  -e, --ejs            add ejs engine support
  --pug               add pug engine support
  --hbs               add handlebars engine support
  -H, --hogan          add hogan.js engine support
  -v, --view <engine>  add view <engine> support (dust|ejs|hbs|hjs|jade|pug|twig|vash) (defaults to jade)
  -c, --css <engine>   add stylesheet <engine> support (less|stylus|compass|sass) (defaults to plain css)
  --git               add .gitignore
  -f, --force          force on non-empty directory

[caio: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* (tempo real).

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, utilizado no teste de integração;
- **Nginx**: servidor web de alta performance para arquivos estáticos.

Exploraremos essas tecnologias no decorrer destes capítulos, então, muita calma e boa leitura! Caso você esteja com pressa de ver o projeto rodando, você pode cloná-lo em 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, acesse o endereço no seu navegador favorito:
<http://localhost:3000>.

Agora, se você quer aprender o passo a passo para desenvolver esse projeto, continue lendo o livro e seguindo todas as dicas apresentadas.

4.4 GERANDO O SCAFFOLD DO PROJETO

Criaremos o diretório da aplicação já com alguns recursos do Express que são gerados 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 tudo isso foi gerado:

```
pi@caio: ~ (bash) 361 X bash 362 X bash 363 X 1. bash
[caio:nodejs] $ express ntalk --ejs
warning: option '--ejs' has been renamed to '--view=ejjs'

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/views
create : ntalk/views/index.ejjs
create : ntalk/views/error.ejjs
create : ntalk/public
create : ntalk/bin
create : ntalk/bin/www
create : ntalk/public/javascripts
create : ntalk/public/images
create : ntalk/public/stylesheets
create : ntalk/public/stylesheets/style.css

install dependencies:
$ cd ntalk && npm install

run the app:
$ DEBUG=ntalk:* npm start
[caio:nodejs] $
```

Figura 4.3: Estrutura do Express

Veja a seguir uma breve explicação sobre os arquivos e pastas criados:

- 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 pelo 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 `cookie-parser`.

Agora faremos algumas alterações nos códigos gerados pelo comando `express`. O primeiro passo será criar uma **descrição sobre o que será esse projeto**, por meio do atributo `"description"`. Por último, modificaremos o atributo `scripts` para `"start": "node app.js"` com intuito de customizar nosso comando inicializador de nosso projeto. 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": true,  
  "scripts": {  
    "start": "node app.js"  
  },  
  "dependencies": {  
    "body-parser": "~1.17.1",  
    "cookie-parser": "~1.4.3",  
    "debug": "~2.6.3",  
    "ejs": "~2.5.6",  
    "express": "~4.15.2",  
    "morgan": "~1.8.1",  
    "serve-favicon": "~2.4.2"  
}
```

```
}
```

Para simplificar nosso trabalho, vamos remover algumas coisas que foram geradas pelo comando `express`. Primeiro, **exclua o diretório `bin`**, pois não vamos utilizá-lo, afinal, vamos usar uma maneira mais simples para iniciar, que ocorre pelo comando `npm start`. Ele já foi incluído no `package.json` por meio do atributo `"scripts": { "start": "node app.js" }`.

Em seguida, vamos remover os seguintes módulos: `morgan`, `debug` e `static-favicon`, que não serão utilizados em nosso projeto por enquanto.

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

No final, seu `package.json` será algo semelhante a este:

```
{
  "name": "ntalk",
  "description": "Node talk - Agenda de contatos",
  "version": "0.0.1",
  "private": false,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "body-parser": "~1.17.1",
    "cookie-parser": "~1.4.3",
    "ejs": "~2.5.6",
    "express": "~4.15.2"
  }
}
```

Pronto, agora podemos começar. Vamos modificar o `app.js`, deixando-o com o mínimo de código possível para explicarmos em *baby steps* o que realmente será adotado no desenvolvimento do nosso projeto. Recomendo que **apague todo o código gerado**, e coloque o código simplificado do que será nossa aplicação:

```
const express = require('express');
const path = require('path');
const routes = require('./routes/index');
const users = require('./routes/users');
const app = express();

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

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

app.listen(3000, () => {
  console.log('Ntalk no ar.');
});
```

Como se inicia um servidor Express?

Essa versão inicialmente atende aos requisitos mínimos de uma aplicação Express. A brincadeira começa quando executamos a função `express()`, pois o seu retorno instancia e habilita todas as suas funcionalidades de um servidor web, e as principais funcionalidades desse framework fica disponível e acessível pela variável `app`, por meio do trecho: `const app = express();`.

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

Como se configuram os middlewares?

O `app.set(chave, valor)` é uma estrutura segura de chave e valor, que fica na variável `app`. Seria o mesmo que criar o código `app['chave'] = 'valor'`; no JavaScript. Um exemplo prático são as configurações de `views`, definidas no código anterior:

```
app.set('views', path.join(__dirname,
```

```
'views')) e app.set('view engine', 'ejs') .
```

Por enquanto, a nossa aplicação possui apenas duas configurações internas: a primeira indica o diretório das **views**, e a segunda indica qual *template engine* será usado para renderizar HTMLs dinâmicos – no nosso caso, usaremos o EJS (*Embedded JavaScript*).

Também contamos com três middlewares por meio das funções `app.use()`. O primeiro indica uma pasta para servir arquivos estáticos, e as outras duas são rotas específicas da aplicação para acessar uma determinada página dinâmica, que será renderizada pelas views do *EJS*. No decorrer deste livro, serão incluídos e explicados novos middlewares e novas rotas.

Como criar rotas no Express?

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

Repare como são executados seus respectivos módulos; eles vieram das variáveis `const routes = require('./routes/index')` e `const users = require('./routes/users')`. Apenas carregamos seus respectivos módulos do diretório `routes`, cada um contendo suas regras de negócio para renderizar uma página.

4.5 ORGANIZANDO OS DIRETÓRIOS DO PROJETO

Quando o assunto é organização de códigos, o Express comporta-se 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 o back-end dentro do código `app.js`, ou se o projeto possuir diversas rotas, *views*, *models* e *controllers*. O ideal é que seja montada uma estrutura modularizável permitindo a utilização do *pattern* que melhor se encaixe nele.

Em nosso projeto, vamos implementar o padrão MVC (*Model-View-Controller*). Para isso, vamos criar os seguintes diretórios `models` e `controllers`, deixando sua estrutura dessa forma:

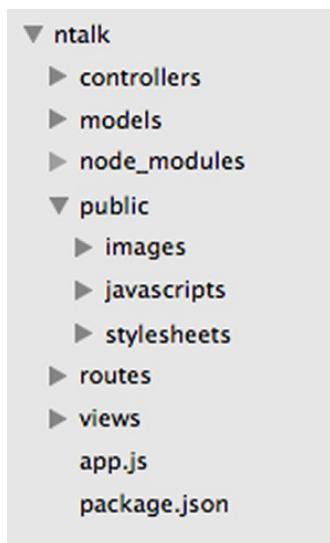


Figura 4.4: Estrutura de diretórios do ntalk

Cada *model* que for usado em um determinado *controller* realizará uma chamada à 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 no carregamento dos módulos em um código.

Com base nesse problema, surgiu um *plugin* que visa minimizar essas terríveis chamadas, o `consign` . Ele é responsável por mapear diretórios para carregar e injetar módulos dependentes dentro de uma variável que definirmos na função `consign({}).include('models').then('controllers').then('routes').into(app)` .

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

```
npm install consign --save
```

Vamos substituir todo o tipo de carregamento de routes que usa função `require()` por um carregamento mais estruturado e automatizado utilizando o módulo `consign` . Neste caso, vamos aplicar alguns *refactorings*, então, edite o `app.js` , deixando-o desse jeito:

```
const express = require('express');
const path = require('path');
const app = express();

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

app.listen(3000, () => {
  console.log('Ntalk no ar.');
});
```

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

```
const express = require('express');
const path = require('path');
const consign = require('consign');
const app = express();

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

consign({})
  .include('models')
  .then('controllers')
  .then('routes')
  .into(app)
;

app.listen(3000, () => {
  console.log('Ntalk no ar.');
});
```

ATENÇÃO

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

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

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

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

Repare que, por causa do `consign`, a variável `app` já possui como atributo o objeto `controllers`. Neste caso, o `app.controllers.home` está referenciando-se 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 = (app) => {
  const HomeController = {
    index: function(req, res) {
      res.render('home/index');
    }
  };
  return HomeController;
};
```

Para terminar o fluxo entre `controllers` e `routes`, temos de renderizar uma `view`, que basicamente é uma página HTML, mostrando algum resultado para o usuário. Exclua o arquivo `views/index.ejs`, criado pelo `express-generator`, e crie o `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">
      <br>
      <input type="text" name="usuario[email]"
        placeholder="E-mail">
      <br>
      <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 `npm start` e, em seguida, acesse em seu *browser* o endereço: <http://localhost:3000>. Se tudo deu certo, você verá uma tela com um formulário semelhante à figura:

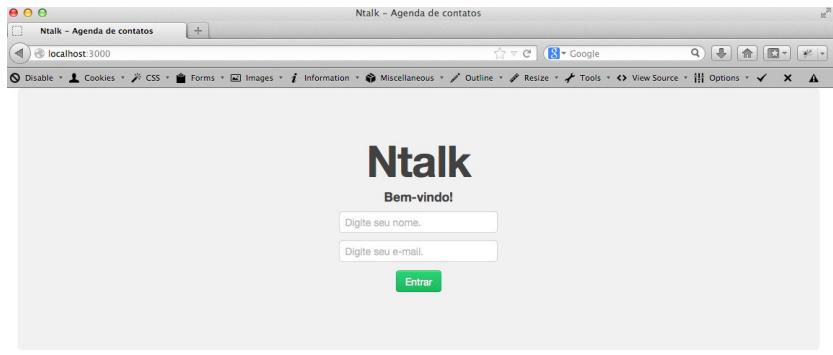


Figura 4.5: Tela de login do Ntalk

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 apresentadas 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 <https://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, pois temos 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, pois 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* pela 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">  
    <br>  
    <input type="text" name="usuario[email]"  
      placeholder="E-mail">  
    <br>  
    <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 utilizando a aplicação.

Trabalhar com sessão é muito simples, e os dados são

manipulados por meio de um objeto JSON, dentro da estrutura `req.session`. Para aplicar a 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 para `logout` via função `app.get('/sair', home.logout)`.

```
module.exports = (app) => {
  const { home } = app.controllers;
  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`, utilizados no `routes/home.js`.

Agora temos de codificar suas respectivas `actions` no `controllers/home.js`. Na `action login`, será implementada uma simples regra para validar se existem 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 pela 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 a homepage. Veja como ficarão essas lógicas no arquivo `controllers/home.js`:

```
module.exports = (app) => {
  const HomeController = {
    index(req, res) {
      res.render('home/index');
    },
    login(req, res) {
      const { usuario } = req.body;
      const { email, nome } = usuario;
      if (email && nome) {
        usuario.contatos = [];
        req.session.usuario = usuario;
        res.redirect('/contatos');
      } else {
        res.redirect('/');
      }
    },
    logout(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 que existir na sessão da aplicação. Para isso, enviaremos a variável `req.session.usuario` na renderização de sua *view* por meio da função `res.render()`.

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

```
module.exports = (app) => {
  const ContatosController = {
    index(req, res) {
      const { usuario } = req.session;
      res.render('contatos/index', { usuario });
    },
  };
  return ContatosController;
};
```

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

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

Para finalizar, vamos criar uma simples *view* para a tela de contatos, que melhoraremos 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 adicionada a tag `<% include ../exit %>`. Esta 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>
  <a href="/contatos">Contatos</a> | <a href="/sair">Sair</a>
</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 **npm start** e, depois, em seu browser, tente fazer um login no sistema.

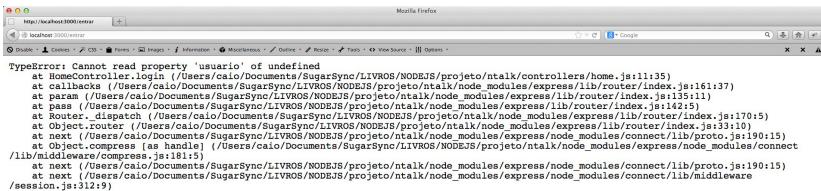


Figura 5.1: Infelizmente, deu mensagem de erro

"O que houve? Eu implementei tudo certo!". Então, meu amigo, esse erro aconteceu porque faltou usar um middleware que já tínhamos instalado 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 um objeto JSON, afinal, o objeto `req.body.usuario` não foi reconhecido. Já adiantando, vamos habilitar o controle de sessão pelo módulo `express-session` e também o gerenciador de *cookies* do módulo `cookie-parser`, para que tudo funcione corretamente na próxima vez.

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

```
const express = require('express');
const path = require('path');
const consign = require('consign');
const bodyParser = require('body-parser');
```

```

const cookieParser = require('cookie-parser');
const expressSession = require('express-session');
const app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
app.use(cookieParser('ntalk'));
app.use(expressSession());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(express.static(path.join(__dirname, 'public')));

consign({})
  .include('models')
  .then('controllers')
  .then('routes')
  .into(app)
;

app.listen(3000, () => {
  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 o `bodyParser.urlencoded()`, responsáveis por criar objetos JSON vindos de um formulário HTML.

Esse middleware basicamente cria um objeto pelos 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 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 sobrescrever 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 está pronta para testes! Reinicie o servidor teclando no terminal `CTRL+C` (no Windows ou Linux) ou `Command+C` (no MacOS), e execute `npm start`. Por último, acesse em seu browser <http://localhost:3000>. Faça novamente um login no sistema. Dessa vez, temos uma nova tela, 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` :

```
const express = require('express');
const path = require('path');
const consign = require('consign');
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
const expressSession = require('express-session');
const methodOverride = require('method-override');
const app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
app.use(cookieParser('ntalk'));
app.use(expressSession());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(methodOverride('_method'));
app.use(express.static(path.join(__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 = (app) => {
```

```
const { contatos } = app.controllers;
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 um banco de dados, todos os dados serão persistidos na própria sessão do usuário, ou seja, todos os contatos serão gravados em memória e apagados ao sair da aplicação.

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 = (app) => {
  const ContatoController = {
    index(req, res) {
      const { usuario } = req.session;
      const { contatos } = usuario;
      res.render('contatos/index', { usuario, contatos });
    },
    // continuação do controller...
```

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

```

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

```

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

```

show(req, res) {
  const { id } = req.params;
  const { usuario } = req.session;
  const contato = usuario.contatos[id];
  res.render('contatos/show', { id, contato });
},
edit(req, res) {
  const { id } = req.params;
  const { usuario } = req.session;
  const contato = usuario.contatos[id];
  res.render('contatos/edit', { id, contato, usuario });
},
// 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(req, res) {
  const { contato } = req.body;
  const { usuario } = req.session;
  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(req, res) {
  const { id } = req.params;
  const { usuario } = req.session;
  usuario.contatos.splice(id, 1);
  res.redirect('/contatos');
}
};

return ContatoController;
};
```

Este 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. Vamos modificar a view `views/contatos/index.ejs` para renderizar uma lista de contatos, e um formulário para cadastrar novos:

```
<% 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((contato, index) => { %>
    <tr>
      <td><%= contato.nome %></td>
      <td><%= contato.email %></td>
      <td>
        <a href="/contato/<%= index %>">Detalhes</a>
      </td>
    </tr>
  <% }) %>
</tbody>
</table>
</section>
<% include ../exit %>
<% include ../footer %>

```

Agora, no mesmo diretório, vamos criar o views/contatos/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 views/contatos/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</a>
        </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 paliativas. Uma delas, que aplicamos em nossa aplicação, é a inclusão de uma query string com nome `_method=put`, ou `_method=delete`, para que seja reconhecida pelo middleware `methodOverride('_method')` e faça uma sobreescrita dessas rotas para o método `PUT` ou `DELETE` da tag `<form>`. Veja um exemplo a seguir:

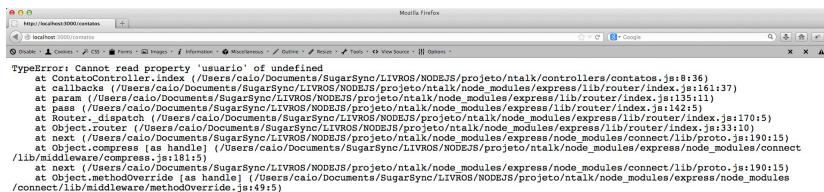
```
<form action="/contatos?_method=put" method="post">  
<form action="/contatos?_method=delete" method="post">
```

5.4 APlicando FILTROS ANTES DE ACESSAR AS ROTAS

Já percebeu que acontece um bug quando acessamos a rota `/contatos` sem logar no sistema? 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:



```
TypeError: Cannot read property 'usuario' of undefined
    at ContatoController.index (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/ntalk/controllers/contatos.js:8:36)
    at Object. (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/ntalk/node_modules/express/lib/router/index.js:151:37)
    at param (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/ntalk/node_modules/express/lib/router/index.js:135:11)
    at pass (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/ntalk/node_modules/express/lib/router/index.js:142:5)
    at Router.dispatch (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/ntalk/node_modules/express/lib/router/index.js:170:5)
    at Object.route (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/ntalk/node_modules/express/lib/router/index.js:133:10)
    at next (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/ntalk/node_modules/express/lib/router/index.js:190:15)
    at Object.compress [as handle] (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/ntalk/node_modules/express/node_modules/connect/lib/proto.js:190:15)
    at next (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/ntalk/node_modules/express/node_modules/connect/lib/proto.js:190:15)
    at Object.methodOverride [as handle] (/Users/caio/Documents/SugarSync/LIVROS/NODEJS/projeto/ntalk/node_modules/express/node_modules/connect/lib/middleware/methodoverride.js:49:5)
```

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 o seu path no primeiro parâmetro, 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 nela o arquivo *autenticador.js*. Nele, implemente a lógica a seguir:

```
module.exports = (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 ir para a função ao lado. Caso a autenticação não aconteça, executamos um simples `return res.redirect('/')`, que faz o usuário voltar à página inicial e impede que ocorra o bug anterior.

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

```
const autenticar = require('../middlewares/autenticador');  
  
module.exports = (app) => {  
    const { contatos } = app.controllers;  
  
    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.delete('/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 o roteamento e a renderização de erros do protocolo HTTP. Ele possui apenas duas funções: uma específica para tratamento do famoso erro **404** (página não encontrada), e uma genérica que recebe por parâmetro uma variável contendo detalhes sobre o status e a 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 o código de cada erro. Para ficar por dentro desse assunto, veja a especificação dos status de erros gerados por esse protocolo em <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

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

Primeiro, crie duas novas views. A primeira será 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 <a href="/">home page?</a> :)</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:
  <br>
  <%- error.message %>
</p>
<hr>
<p>Que tal voltar <a href="/">home page?</a> :)</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 = (req, res, next) => {
  res.status(404);
  res.render('not-found');
};

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

```

Em seguida, modifique o *stack* de middlewares editando o `app.js`. Repare que esse redirecionamento para a 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:

```
const express = require('express');
const path = require('path');
const consign = require('consign');
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
const expressSession = require('express-session');
const methodOverride = require('method-override');
const error = require('./middlewares/error');
const app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
app.use(cookieParser('ntalk'));
app.use(expressSession());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(methodOverride('_method'));
app.use(express.static(path.join(__dirname, 'public')));

consign({})
  .include('models')
  .then('controllers')
  .then('routes')
  .into(app)
;
// middleware de tratamento erros
app.use(error.notFound);
app.use(error.serverError);

app.listen(3000, () => {
  console.log('Ntalk no ar.');
});
```

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

encontrada (erro 404).

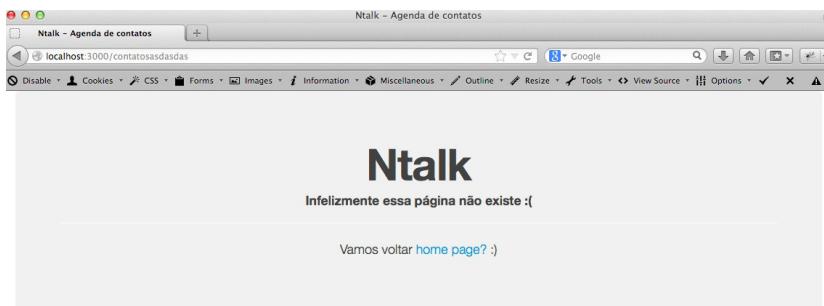


Figura 5.3: Tela de erro 404

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 nele, remova um filtro da rota `/contatos`, forçando o bug que ocorria na seção anterior, na qual falávamos sobre a implementação de filtros.

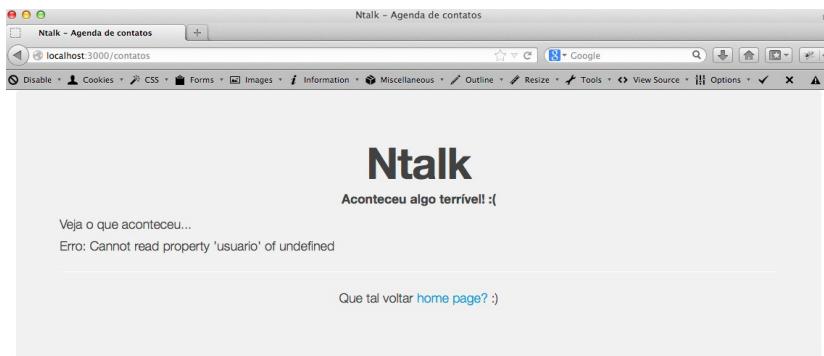


Figura 5.4: Tela de erro 500

Parabéns! Agora temos uma aplicação que cadastrá, 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 o 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.

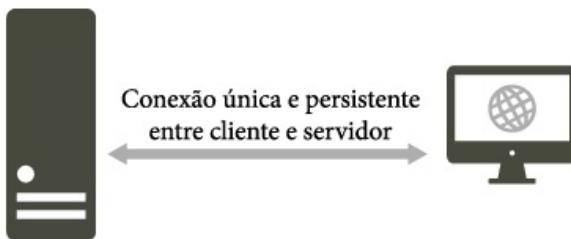


Figura 6.1: Imagem explicando sobre conexão bidirecional

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* e os navegadores antigos, emulando *Ajax long-polling* ou outros *transports* de comunicação em browsers que não possuem *WebSockets*, por exemplo, e tudo de forma totalmente abstraída para o desenvolvedor. Seu site oficial é <https://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. *AJAX long polling*;
3. *Forever iframe*;

Se o navegador do usuário possuir compatibilidade com *WebSockets*, será realizada uma comunicação bidirecional. Caso contrário, será emulada uma comunicação unidirecional, que faz requisições AJAX no servidor em curtos intervalos de tempo. 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 um 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 pela função `server.listen(3000)`, para que o Socket.IO use o mesmo listener para criar seu ponto de comunicação através do protocolo HTTP. Afinal, o WebSockets é um protocolo que funciona em conjunto com HTTP ou HTTPS.

Veja a seguir como ficarão essas modificações:

```
const express = require('express');
const path = require('path');
const http = require('http');
const socketIO = require('socket.io');
const consign = require('consign');
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
const expressSession = require('express-session');
const methodOverride = require('method-override');
const error = require('./middlewares/error');

const app = express();
const server = http.Server(app);
const io = socketIO(server);

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
app.use(cookieParser('ntalk'));
app.use(expressSession());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(methodOverride('_method'));
app.use(express.static(path.join(__dirname, 'public')));

consign({})
  .include('models')
  .then('controllers')
  .then('routes')
  .into(app)
;

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

server.listen(3000, () => console.log('Ntalk no ar.'));
```

Dessa forma, habilitamos o Socket.IO em nossa aplicação, mas até agora nenhuma funcionalidade dele foi implementada. Para isso, vamos criar nosso primeiro evento de mensageria real-time no servidor, por meio do callback da função: `io.on('connection')`.

Ainda no `app.js`, inclua essa função depois do carregamento das rotas da aplicação que utilizam a função `consign()`, usando o seguinte trecho de código:

```
// carregamento dos módulos..
// carregamento dos middlewares...
// carregamento das rotas...
io.on('connection', (client) => {
  client.on('send-server', (data) => {
    const resposta = `<b>${data.nome}</b> ${data.msg}<br>`;
    client.emit('send-client', resposta);
    client.broadcast.emit('send-client', resposta);
  });
});
// execução do server.listen() ...
```

Toda a brincadeira começa a partir do evento `io.on('connection')`. Essa função fica aguardando que um cliente conecte-se ao **socket.io** e envie uma mensagem para o servidor por meio de uma função de emissora. No nosso caso, o cliente terá uma função `emit('evento', dados)`. Qualquer nome de evento pode ser criado, exceto alguns nomes de eventos-chaves, que serão apresentados ao final deste capítulo.

Perceba que pouco código foi incluído e o servidor já responde com seus clientes por meio dos eventos `client.emit('send-client', resposta)` e `client.broadcast.emit('send-client', resposta)`. Resumindo, o fluxo básico de um chat é o cliente enviar uma mensagem para o servidor, e este responder ao

próprio cliente (via `client.emit()`) e aos demais conectados (via `client.broadcast.emit()`). Para compreender melhor, veja as figuras a seguir:

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

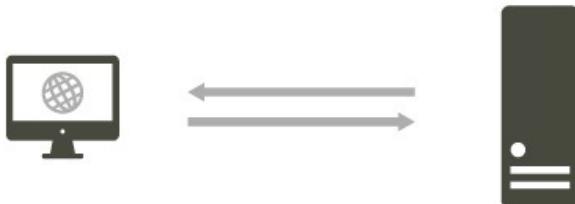


Figura 6.3: Envia mensagens para o cliente ou servidor

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

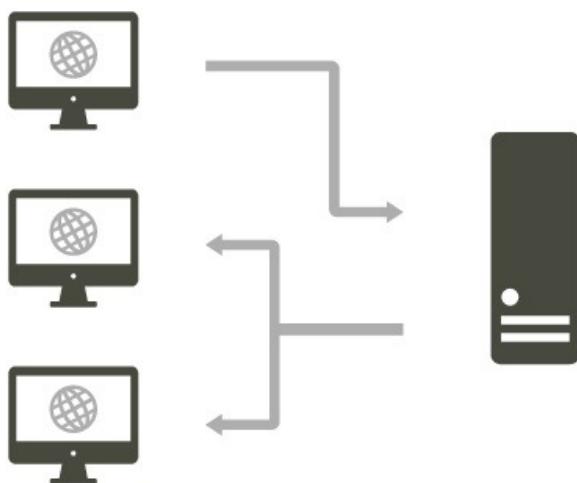


Figura 6.4: Envia mensagens para todos os clientes, exceto o próprio emissor

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 criaremos em seguida.

```
<% contatos.forEach((contato, index) => { %>
  <tr>
    <td><%- contato.nome %></td>
    <td><%- contato.email %></td>
    <td>
      <a href="/contato/<%- index %>">Detalhes</a>
      <a href="/chat">Conversar</a>
    </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, é preciso criar seu controller em `controllers/chat.js`:

```
module.exports = (app) => {
  const ChatController = {
    index(req, res) {
      const { usuario } = req.session;
      res.render('chat/index', { usuario });
    }
  };
  return ChatController;
};
```

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

```
const autenticar = require('../middlewares/autenticador');

module.exports = (app) => {
  const { chat } = app.controllers;
  app.get('/chat', autenticar, chat.index);
};
```

Por último, crie sua view em `views/chat/index.ejs`. Usaremos JavaScript puro para a manipulação de elementos do

HTML, mas sinta-se à vontade para utilizar qualquer framework do gênero, como *jQuery* (<https://jquery.com>) ou *ZeptoJS* (<https://zeptojs.com>).

O importante é carregar o script `/socket.io/socket.io.js` para que a brincadeira comece. Aliás, não se preocupe de onde vem 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 pelo tráfego de objetos JSON. Vamos incluir as funções de enviar e receber mensagens na view `views/chat/index.ejs`:

```
<% include ../header %>
<script src="/socket.io/socket.io.js"></script>
<script>
  const socket = io();
  socket.on('send-client', (msg) => {
    document.getElementById('chat').innerHTML += msg;
  });
  const enviar = () => {
    const nome = document.getElementById('nome').value;
    const msg = document.getElementById('msg').value;
    socket.emit('send-server', { nome, 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 o 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 o nome e e-mail do usuário da outra janela em contatos (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.

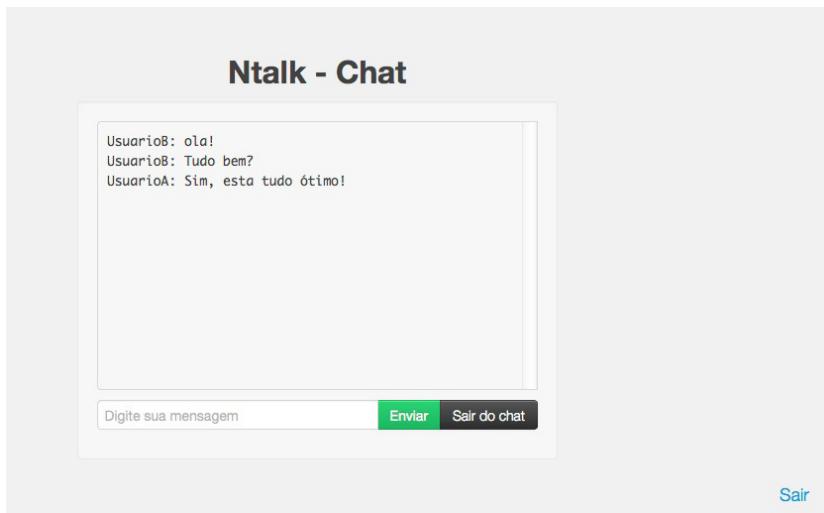


Figura 6.5: Usuário A conversando com Usuário B

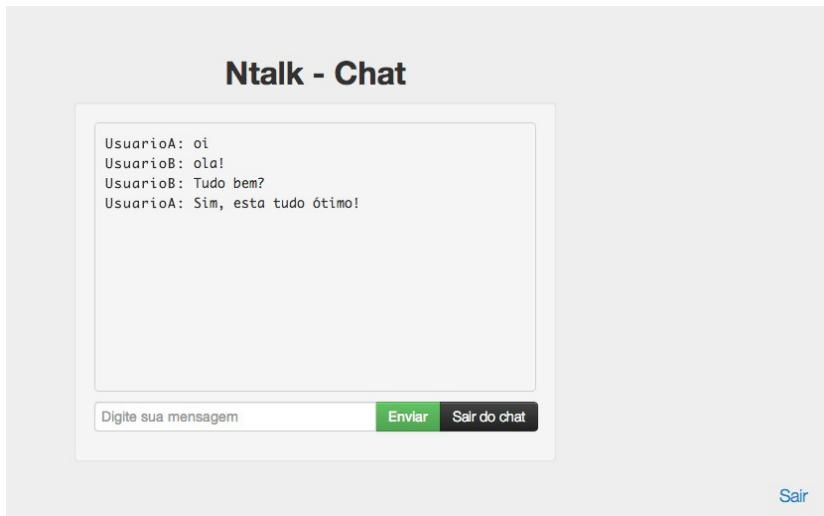


Figura 6.6: Usuário B respondendo mensagens do Usuário A

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 à qual fizemos com as *views*, *models* e *controllers*.

Com base nesse problema, crie o diretório `events` para manter futuros módulos dos eventos do Socket.IO e, dentro desse diretório, adicione o arquivo `chat.js`. Para esse arquivo, migre o trecho de código Socket.IO do `app.js`. Veja o exemplo que segue:

```
module.exports = (app, io) => {
  io.on('connection', (client) => {
    client.on('send-server', (data) => {
      const resposta = `<b>${data.nome}</b> ${data.msg}<br>`;
      client.emit('send-client', resposta);
      client.broadcast.emit('send-client', resposta);
    });
  });
};
```

Dessa forma, deixamos o app.js com menos código. Um detalhe importante é que o chat será carregado também pelo consign e, como segundo parâmetro, vamos passar a variável io :

```
// carregamento dos módulos..
// carregamento dos middlewares...
consign({})
  .include('models')
  .then('controllers')
  .then('routes')
  .then('events')
  .into(app, io)
;
// server.listen()...
```

Mais uma vez, usamos 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 COMPARTILHANDO SESSÃO ENTRE SOCKET.IO E EXPRESS

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 ela 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 o Socket.IO. Para começar, vamos criar o arquivo com configurações. Por enquanto, ele vai conter apenas os dados de `sessionKey` e `sessionSecret`, que serão utilizados para buscar o decodificar e o ID de uma sessão de usuário logado. Para fazer isso, crie o arquivo `config.js`, na raiz do projeto.

```
module.exports = {
  sessionKey: 'ntalk.id',
  sessionSecret: 'ntalk_secret'
};
```

ATENÇÃO

Por fins didáticos, o `sessionSecret` terá um valor simples de decorar. Porém, em um ambiente de produção, a boa prática é gerar um hash complexo para usá-lo como chave secreta na sessão. Isso impossibilitará o *hijacking* da sessão de um usuário no sistema. Um bom exemplo para esse caso seria:

```
module.exports = {
  sessionKey: 'ntalk.id',
  sessionSecret: 'up8F4PRpx9iR7HPJBgaJVZdp23WI3qGdk8UvC4Q'
};
```

Agora, temos de fazer algumas modificações no `app.js`, seguindo esse trecho de código:

```
const express = require('express');
const path = require('path');
const http = require('http');
const socketIO = require('socket.io');
const consign = require('consign');
const bodyParser = require('body-parser');
const cookie = require('cookie');
const expressSession = require('express-session');
const methodOverride = require('method-override');
const config = require('./config');
const error = require('./middlewares/error');

const app = express();
const server = http.Server(app);
const io = socketIO(server);
const store = new expressSession.MemoryStore();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
app.use(expressSession({
    store,
    name: config.sessionKey,
    secret: config.sessionSecret
}));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(methodOverride('_method'));
app.use(express.static(path.join(__dirname, 'public')));
// continuação do app.js ...
```

Com esses recursos habilitados, será possível compartilhar os dados de sessão do Express no Socket.IO. Mas, para isso, vamos implementar um controle de checagem e autorização via função `io.use(callback)`. Assim, a cada conexão, o Socket.IO fará uma leitura do cookie do usuário para checar se o `sessionID` é válido para autorização e recuperação dos dados do usuário presente na sessão.

A seguir, implementaremos um simples verificador para essa regra dentro do `app.js`:

```
// carregamento dos módulos..
// carregamento dos middlewares...
io.use((socket, next) => {
  const cookieData = socket.request.headers.cookie;
  const cookieObj = cookie.parse(cookieData);
  const sessionHash = cookieObj[config.sessionKey] || '';
  const sessionID = sessionHash.split('.')[0].slice(2);
  store.all((err, sessions) => {
    const currentSession = sessions[sessionID];
    if (err || !currentSession) {
      return next(new Error('Acesso negado!'));
    }
    socket.handshake.session = currentSession;
    return next();
  });
});
// consign()...
// server.listen()...
```

A função `next()` é responsável pela autorização ou não da conexão, e o objeto `socket.request.headers.cookie` contém as informações do cookie da sessão do cliente, ou seja, é nele que existirão os dados salvos no cookie, incluindo o hash da sessão do usuário. No nosso caso, precisamos capturar o `sessionID`.

Para isso, basta executar a função `cookie.parse(cookieData)` para gerar um objeto com os dados do cookie e, em seguida, acessar esse hash da sessão via `cookieObj[config.sessionKey]`. Este vai retornar uma string padronizada no formato: '`s:`' + 32 chars + '.' + 43 chars, por exemplo,

`'s:B1wCrzrCZEis69iR7HPJBgaJVZdp23WI.up8F4PRYqbgdVtu5sw3qpxeXv5d9eQgEP8Gdk8Uvc4Q'`.

Nessa string, o `sessionID` é apenas o conjunto dos 32 chars,

após 's:' . Para capturar corretamente esses 32 chars, foi usada a combinação da função `split('.')` e da `slice(2)` , que se encontram no trecho: `sessionHash.split('.')[0].slice(2);` . Após reconstruir o `sessionID` , utilizamos a função `store.all()` e, em seu callback, é retornado um objeto com todas as sessions desta aplicação. Assim, recuperamos a sessão atual pelo trecho: `const currentSession = sessions[sessionID] .`

Incluímos a sessão no objeto `socket.handshake.session` e, para finalizar, a conexão do Socket.IO é autorizada pela função `return next()` . Caso contrário, para bloquear uma conexão em caso de erro ou sessão inválida, é executada a função `next(new Error('Acesso negado!'))` , passando em seu primeiro parâmetro um objeto `Error('Acesso negado!')` .

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 %>">` . Em seguida, modificaremos a função `enviar()` , para que ela envie somente 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>
  const socket = io();
  socket.on('send-client', (msg) => {
    document.getElementById('chat').innerHTML += msg;
```

```

    });
    const enviar = () => {
        const msg = document.getElementById('msg').value;
        socket.emit('send-server', msg);
    };
</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 %>

```

Também removeremos do controllers/chat.js toda chamada referente aos dados da sessão do usuário. Neste caso, vamos remover: `const { usuario } = req.session;`. Também modificaremos `res.render('chat/index', { usuario })` para apenas `res.render('chat/index')`. Veja como ficará:

```

module.exports = (app) => {
    const ChatController = {
        index(req, res) {
            res.render('chat/index');
        }
    };
    return ChatController;
};

```

Com a *view* e o *controller* atualizados, vamos adaptar no events/chat.js dentro do evento `io.on('connection')` para que as informações de sessão do usuário sejam tratadas e emitidas pelo servidor, concatenando o nome e o conteúdo da mensagem do usuário em uma simples tag HMTL. Dessa vez, os dados do usuário serão carregados corretamente pelo objeto:

`client.handshake.session.usuario`. Este é o responsável por manter os dados de sessão do usuário.

```
module.exports = (app, io) => {
  io.on('connection', (client) => {
    const { session } = client.handshake;
    const { usuario } = session;

    client.on('send-server', (msg) => {
      const resposta = `<b>${usuario.nome}</b> ${msg}<br>`;
      client.emit('send-client', resposta);
      client.broadcast.emit('send-client', resposta);
    });
  });
};
```

Que tal testar essas modificações? Faça o seguinte: reinicie o servidor e acesse o browser em `http://localhost:3000`. **Antes de logar no sistema, apague os cookies do browser**, pois foi configurada uma nova chave do cookie nesta aplicação. Após esse procedimento, teste o seu chat iniciando uma nova conversa entre dois usuários, e veja se as mensagens estão incluindo seus nomes 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. Assim, vamos prevenir que outros entrem no meio de uma conversa a dois.

Desenvolver essa funcionalidade é muito simples: apenas precisamos criar uma string, que será o nome da sala, e utilizá-la pela função `io.to('nome_da_sala').emit()`. Outro detalhe dessa função é que ela emite um evento para todos os usuários da

sala, incluindo o próprio 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 criptografá-los. Ele será um *Hash MD5* de um *timestamp*, criado pelo primeiro usuário que tomar a iniciativa de iniciar uma conversa com outra pessoa.

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, vamos carregar o módulo `crypto` e seu objeto responsável por gerar o valor `hash md5` da sala. Para isso, edite o `controllers/chat.js` da seguinte maneira:

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

module.exports = (app) => {
```

```

const ChatController = {
  index(req, res) {
    const { sala } = req.query;
    let hashDaSala = sala;
    if (!hashDaSala) {
      const timestamp = Date.now().toString();
      const md5 = crypto.createHash('md5');
      hashDaSala = md5.update(timestamp).digest('hex');
    }
    res.render('chat/index', { sala: hashDaSala });
  }
};

return ChatController;
};

```

Dessa forma, vamos garantir que será gerado um hash da sala quando não for enviado um hash de sala existente via querystring. Assim, garantiremos a geração de uma nova sala quando o primeiro usuário iniciar o chat.

Após criar o gerador de hash no controller, vamos editar a página `views/chat/index.ejs` para que, quando um usuário entrar nessa tela, automaticamente dispare o evento `socket.emit('create-room', sala)` para incluir o usuário lá corretamente. A sala e a mensagem do usuário também serão enviadas pelo evento `socket.emit('send-server', sala, msg.value);`:

```

<% include ../header %>
<script src="/socket.io/socket.io.js"></script>
<script>
  const socket = io();
  const sala = '<%- sala %>';
  socket.emit('create-room', sala);
  socket.on('send-client', (msg) => {
    document.getElementById('chat').innerHTML += msg;
  });
  const enviar = () => {
    const msg = document.getElementById('msg');
    socket.emit('send-server', sala, msg.value);
  };
</script>

```

```

    };
</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 criar alguns eventos importantes no events/chat.js . Dentro do evento `io.on('connection')` , criaremos um novo, o `client.on('create-room')` , que será emitido quando um usuário enviar o hash de uma sala, para que entre em um chat. Em seguida, salvaremos o seu hash na sessão do usuário, via `session.sala = sala` .

Por fim, para confirmar a inclusão de um usuário em uma sala na sessão, vamos utilizar a função `client.join(sala)` . Esta é responsável por incluir conexões do Socket.IO (conexões dos usuários) em uma determinada sala.

```

module.exports = (app, io) => {
  io.on('connection', (client) => {
    const { session } = client.handshake;
    const { usuario } = session;

    client.on('send-server', (msg) => {
      const msg = `<b>${usuario.nome}</b> ${msg}<br>`;
      client.emit('send-client', msg);
      client.broadcast.emit('send-client', msg);
    });

    client.on('create-room', (hashDaSala) => {
      session.sala = hashDaSala;
      client.join(hashDaSala);
    });
}

```

```
        client.on('disconnect', () => {
            const { sala } = session;
            session.sasl = null;
            client.leave(sala);
        });
    });
}
```

Para controlar a saída de usuários de uma sala, também foi usado o evento chamado de `client.on('disconnect')`, um evento padrão do Socket.IO. Ele será usado para remover um usuário de uma sala quando este se encontrar desconectado da página. Para descobrirmos em qual sala o usuário está conectado, usaremos a string do objeto `session.sala` para recuperar o hash da sala e, em seu callback, o usuário será removido de uma sala pela função `client.leave(session.sala)`.

Para finalizar, vamos atualizar o evento `client.on('send-server')`, para que ele também receba o hash da sala e atualize o `session.sala = hashDaSala;` de um usuário que iniciar o chat, no caso da primeira mensagem. Também vamos modificar a maneira de reenviar as mensagens para os usuários e, nesse caso, somente mensagens para uma determinada sala serão enviadas pela função `io.to(hashDaSala).emit('send-client', resposta)`.

Para finalizar, criaremos um novo evento chamado `client.broadcast.emit('new-message', novaMensagem)`, que será usado para avisar aos usuários online que uma nova mensagem foi enviada a eles. O objeto `novaMensagem` terá como atributos o e-mail e o hash da sala do usuário, e será injetado na URL do botão **Conversar** de cada contato da tela de lista de contatos.

```
client.on('send-server', (hashDaSala, msg) => {
  const novaMensagem = { email: usuario.email, sala: hashDaSala };
  const resposta = `<b>${usuario.nome}</b> ${msg}<br>`;
  session.sala = hashDaSala;
  client.broadcast.emit('new-message', novaMensagem);
  io.to(hashDaSala).emit('send-client', resposta);
});
```

Agora vamos incluir no botão **Conversar** um meio de obter o *hash* em 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 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>
  const socket = io();
  socket.on('new-message', (data) => {
    const chat = document.getElementById(`chat_${data.email}`);
    if (chat.href.includes('?sala=')) {
      chat.href = chat.href.replace(/\?sala=[\w]+/, `?sala=${data.sala}`);
    } else {
      chat.href += `?sala=${data.sala}`;
    }
  });
</script>
```

Agora em `views/contatos/index.ejs`, inclua o *partial* anterior pela tag `<% include notify_script %>`. Também inclua no botão **Conversar** um *id* para o script anterior identificar onde será atualizada a URL com o *hash* da 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>
                    <a href="/chat" id="chat_<%= contato.email %>">"
                        Conversar
                    </a>
                </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. Ele 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 `events/chat.js` e implemente no início do evento `io.on('connection')` uma nova regra que seguirá a seguinte lógica: armazena o e-mail dos usuários online e, depois, roda um loop desses e-mails para que as funções `client.emit('notify-onlines', email)` e `client.broadcast.emit('notify-onlines', email)` sejam executadas em cada iteração. Isso notificará quem estiver online para os outros usuários da lista de contatos.

Para que ele informe quando um usuário estiver **offline**, ou quando estiver enviando uma **mensagem** em sua agenda, 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')`. Isso garantirá que um usuário saiu do chat.

Por último, removemos o e-mail do usuário da variável `onlines` (pelo código `delete onlines[usuario.email]`) para garantir que, na próxima iteração, ele não esteja mais online. Veja como faremos tudo isso no código a seguir:

```
module.exports = (app, io) => {
  const onlines = {};
  io.on('connection', (client) => {
    const { session } = client.handshake;
    const { usuario } = session;

    onlines[usuario.email] = usuario.email;
    for (let email in onlines) {
      client.emit('notify-onlines', email);
```

```

        client.broadcast.emit('notify-onlines', email);
    }

    client.on('send-server', (hashDaSala, msg) => {
        const novaMensagem = { email: usuario.email, sala: hashDa
Sala };
        const resposta = `<b>${usuario.nome}:</b> ${msg}<br>`;
        session.sala = hashDaSala;
        client.broadcast.emit('new-message', novaMensagem);
        io.to(hashDaSala).emit('send-client', resposta);
    });

    client.on('create-room', (sala) => {
        session.sala = sala;
        client.join(sala);
    });

    client.on('disconnect', () => {
        const { sala } = session;
        const resposta = `<b>${usuario.nome}:</b> saiu.<br>`;
        delete onlines[usuario.email];
        session.sala = null;
        client.leave(sala);
        client.broadcast.emit('notify-offlines', usuario.email);
        io.to(sala).emit('send-client', resposta);
    });
});
}

```

Depois de implementar as regras do chat no lado servidor, 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>
    const socket = io();
    const notify = (data) => {
        const notify = document.getElementById(`notify_${data.email
}`);
        if (notify) {
            notify.textContent = data.msg;
        }

```

```

};

socket.on('notify-onlines', (email) => {
    notify({ email, msg: 'Online' });
});

socket.on('notify-offlines', (email) => {
    notify({ email, msg: 'Offline' });
});

socket.on('new-message', (data) => {
    const chat = document.getElementById(`chat_${data.email}`);
    if (chat.href.includes('?sala=')) {
        chat.href = chat.href.replace(/\?sala=[\w]+/, `?sala=${data.sala}`);
    } else {
        chat.href += `?sala=${data.sala}`;
    }
    notify({ email: data.email, msg: 'Mensagem!' });
});
</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 a 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</span>
                </td>
            </tr>
        <% }) %>
    </tbody>
</table>

```

```

<td>
    <a href="/contato/<%- index %>">Detalhes</a>
    <a href="/chat" id="chat_<%- contato.email %>">
        Conversar
    </a>
</td>
</tr>
<% }) %>
</tbody>
</table>

```

Agora temos o nosso notificador pronto! Para testá-lo, reinicie o servidor e 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. Então, 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.

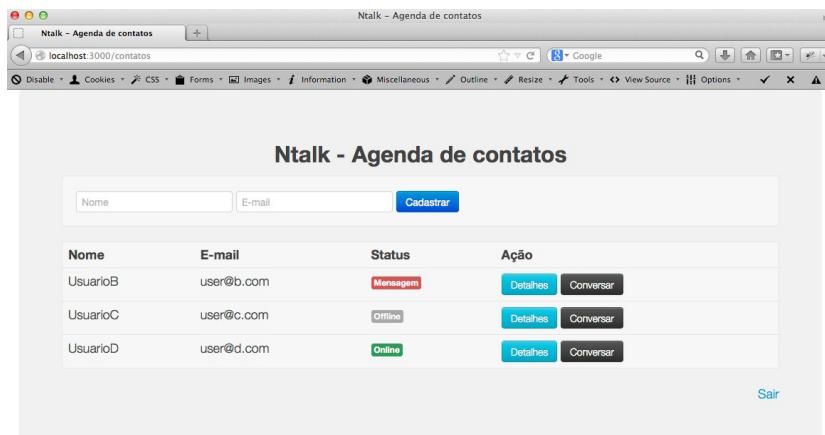


Figura 6.7: Notificações da agenda de contatos do Usuário A

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, temos:

- `io.on('connection', function(client))` – Evento que acontece quando um novo cliente se conecta ao servidor.
- `client.on('message', function(mensagem, callback))` – Ocorre quando um cliente se comunica pela função `send()`. O callback desse evento responde automaticamente o cliente ao 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 por meio 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, temos:

- `client.on('connect', callback)` – Ocorre quando o cliente se conecta ao servidor.
- `client.on('connecting', callback)` – Ocorre quando

o cliente está se conectando ao 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 ao servidor devido a falhas de comunicação entre eles.
- `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 ao servidor.
- `client.on('reconnect', callback)` – Ocorre quando o cliente se reconecta ao servidor.
- `client.on('reconnecting', callback)` – Ocorre quando o cliente está se reconnectando ao 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, que não é recomendado para uso em ambiente de produção, e é mais conhecido como sessão em *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 bancos de dados NoSQL. É claro que existem módulos de banco de dados SQL, mas módulos NoSQL são mais populares nesta plataforma, principalmente porque existem muitos bancos NoSQL que

trabalham diretamente com objetos JSON. Isso facilita a manipulação de dados e integração fácil com uma aplicação Node.js que suporta nativamente JSON.

A grande vantagem de trabalhar com esse modelo de banco de dados é a grande compatibilidade e o suporte mantido pela comunidade própria do Node.js. Os NoSQL populares são:

- **MongoDB** – <https://www.mongodb.org>
- **Redis** – <https://redis.io/>
- **CouchDB** – <https://couchdb.apache.org>



Figura 7.1: NoSQL MongoDB

Neste livro, usaremos o MongoDB, que é um banco de dados NoSQL, escrito em linguagem C/C++. Ele utiliza JavaScript como interface para manipulação de dados, e a persistência dos dados é feita por meio de objetos JSON, que adota conceitualmente o paradigma orientado a documentos.

Nele, trabalhamos com o conceito *schema-less*. Ou seja, não existem relacionamentos de tabelas, nem chaves primárias ou estrangeiras, mas sim documentos que possuem (ou não) documentos embutidos. Estes são subdocumentos dentro de um mesmo documento, e tudo isso é 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 ocorre um **insert** de dados que não possuem todos os atributos de um documento que foi inserido. Isso faz com que o MongoDB seja flexível a grandes mudanças de base de dados.

Como disse antes, com o MongoDB, podemos persistir subdocumentos dentro de um documento, que seria o mesmo que criar um 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. Esses subdocumentos podem ser objetos, como também arrays.

7.2 INSTALANDO O MONGODB

Não há segredos para instalar o MongoDB, tanto é que usaremos suas configurações padrão. Primeiro, acesse <https://www.mongodb.com/download-center>, e faça o download do MongoDB compatível com o seu sistema operacional.

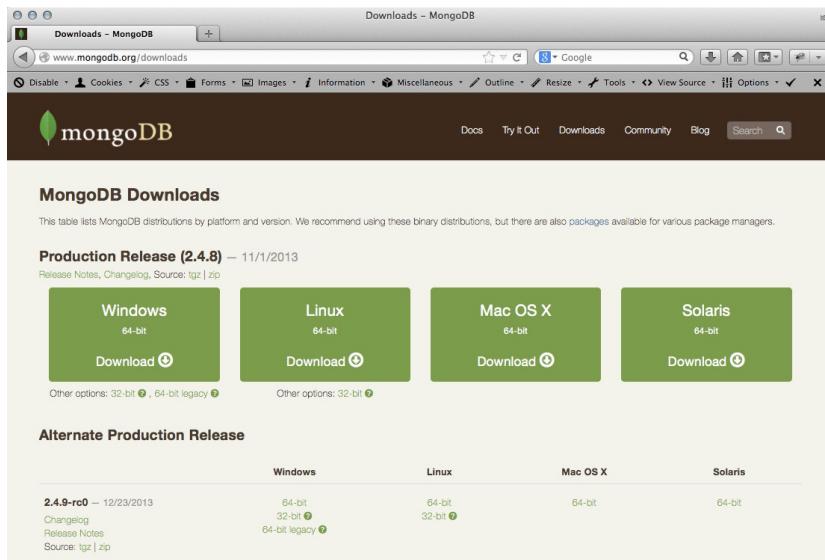


Figura 7.2: Página de download do MongoDB

Neste livro, usaremos sua última versão estável, a **3.4.9**. 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`. Para mais detalhes, basta seguir as dicas deste link:

<https://docs.mongodb.com/master/tutorial/install-mongodb-on-ubuntu>.

Já no Windows, o processo é tão mais trabalhoso que nem apresentarei neste livro... Brincadeira! Instalar no Windows

também é fácil, apenas baixe o instalador do MongoDB de acordo com a versão do seu Windows e certifique-se de que o serviço `mongodb` seja iniciado.

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`, que será o framework responsável por mapear objetos do Node.js para MongoDB; e o `bluebird` para habilitar o uso de **Promises** nas funções do `mongoose`. Para isso, execute o comando:

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

Para a aplicação se conectar com o banco de dados, no `app.js`, usaremos a variável `db` em escopo global (temporariamente apenas para exemplificar o funcionamento da conexão de banco na aplicação). Então, vamos manter uma conexão com o banco de dados compartilhando seus recursos em todo o projeto:

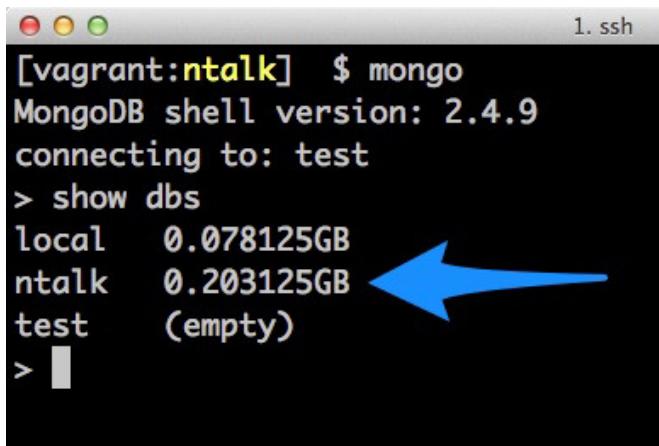
```
const express = require('express');
const path = require('path');
const http = require('http');
const socketIO = require('socket.io');
const consign = require('consign');
const bodyParser = require('body-parser');
const cookie = require('cookie');
const expressSession = require('express-session');
const methodOverride = require('method-override');
const mongoose = require('mongoose');
```

```
const bluebird = require('bluebird');
const config = require('./config');
const error = require('./middlewares/error');
mongoose.Promise = bluebird;
global.db = mongoose.connect('mongodb://localhost:27017/ntalk')
;
```

Quando a função `mongoose.connect` é executada, 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 pela 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 ao modelar seus atributos, por meio do objeto `mongoose.Schema`. Após a implementação de um modelo, temos de registrá-lo no banco de dados usando 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 a aplicação para substituir o modelo de persistência de sessão para o modelo do Mongoose, que armazena os dados no MongoDB.

Para começar, vamos criar o modelo `models/usuario.js`. Ele será responsável por persistir dados de usuários e seus contatos, e terá os seguintes atributos: `nome`, `email` e `contatos`. A modelagem dos atributos acontece pelo objeto `require('mongoose').Schema`. Veja a seguir como esta ficará:

```
/* global db */
const Schema = require('mongoose').Schema;

module.exports = () => {
  const contato = Schema({
    nome: String,
    email: String
  });
  const usuario = Schema({
    nome: {
      type: String,
      required: true
    },
    email: {
```

```
        type: String,
        required: true,
        index: { unique: true }
    },
    contatos: [contato]
});
return db.model('usuarios', usuario);
};
```

Repare 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 é alterar os controllers para usarem suas funções. Começando do mais fácil, vamos modificar o `controllers/home.js`. Nele, vamos executar a função `findOneAndUpdate`, que retorna apenas um objeto, e a função `select('nome email')` para filtrar esse objeto

retornando um novo, que conterá apenas os atributos nome e email do usuário.

Com isso, evitamos que o subdocumento contatos seja carregado ao efetuar o login no sistema. Na prática, essa query seria algo que, em banco de dados SQL, faríamos com o seguinte comando:

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

Nosso tratamento de login será algo bem simplificado, por questões didáticas. Basicamente, vamos adotar o fluxo simples de, ao realizar um login com dados não existentes, automaticamente será cadastrado um novo usuário com eles, e tudo isso será feito por meio da função `Usuario.findOneAndUpdate`. Veja o código-fonte:

```
module.exports = (app) => {
  const Usuario = app.models.usuario;

  const HomeController = {
    index(req, res) {
      res.render('home/index');
    },
    login(req, res) {
      const { usuario } = req.body;
      const { email, nome } = usuario;
      const where = { email, nome };
      const set = {
        $setOnInsert: { email, nome, contatos: [] }
      };
      const options = {
        upsert: true, runValidators: true, new: true
      };
      Usuario.findOneAndUpdate(where, set, options)
        .select('email nome')
        .then((usuario) => {
          req.session.usuario = usuario;
          res.redirect('/contatos');
        });
    }
  };
};

export default HomeController;
```

```

        })
        .catch(() => res.redirect('/'))
    ;
},
logout(req, res) {
    req.session.destroy();
    res.redirect('/');
}
};

return HomeController;
};

```

Repare que, com o *Mongoose*, é possível criar queries complexas chamando funções em que seus retornos ocorrem por meio de **Promises**, pelas funções `then()`. Este é responsável por retornar um resultado de sucesso) e `catch()` (responsável por retornar um erro quando ocorre algum problema.

No refactoring que fizemos, estamos utilizando a função `Usuario.findOneAndUpdate()`, que basicamente vai procurar um registro de um usuário que tenha nome e e-mail válidos. Caso isso exista, será feito um update no registro usando os mesmos dados, apenas para retornar os dados do usuário; caso contrário, usaremos o segundo fluxo, que será o cadastro desse novo usuário.

Isso ocorre através da função `update`, com o `upsert` ativado. Ele é um recurso bem bacana do MongoDB, responsável por atualizar um registro existente. Caso não exista tal registro, ele cadastra um novo.

Para garantir a validação de que será cadastrado um registro válido, também ativamos o atributo `{ runValidators: true }`, responsável por rodar todas as validações existentes no model antes de cadastrar um novo registro. Também ativaremos os atributos `{ upsert: true, new: true }`, que serão

responsáveis por criar e retornar um novo usuário, caso seja feita uma tentativa de login com um e-mail e um nome não existentes na base de dados.

Com isso, eles vão utilizar os atributos do objeto `const set = { $setOnInsert: { email, nome, contatos: [] } }` para criar um novo usuário, com uma lista de contatos vazia.

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 pelo modelo `Usuario`. Veja a seguir as mudanças para cada `action`:

```
const { Types: { ObjectId } } = require('mongoose');

module.exports = (app) => {
  const Usuario = app.models.usuario;

  const ContatosController = {
    index(req, res) {
      const { _id } = req.session.usuario;
      Usuario.findById(_id)
        .then((usuario) => {
          const { contatos } = usuario;
          res.render('contatos/index', { contatos });
        })
        .catch(() => res.redirect('/'));
    },
  };
}
```

Na `action index`, usamos 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.

Também foi incluído no topo o submódulo `const { Types:`

{ ObjectId } } = require('mongoose') . Este será utilizado para fazer um *cast* de um *id string* para o *ObjectId*, nativo do MongoDB. Assim, as funções a seguir vão trabalhar com o tipo de dado do *id* corretamente.

```
create(req, res) {
  const { contato } = req.body;
  const { _id } = req.session.usuario;
  const set = { $push: { contatos: contato } };
  Usuario.findByIdAndUpdate(_id, set)
    .then(() => res.redirect('/contatos'))
    .catch(() => res.redirect('/'))
  ;
},
```

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(req, res) {
  const { _id } = req.session.usuario;
  const contatoId = req.params.id;
  Usuario.findById(_id)
    .then((usuario) => {
      const { contatos } = usuario;
      const contato = contatos.find((ct) => {
        return ct._id.toString() === contatoId;
      });
      res.render('contatos/show', { contato });
    })
    .catch(() => res.redirect('/'))
  ;
},
```

```
edit(req, res) {
  const { _id } = req.session.usuario;
  const contatoId = req.params.id;
  Usuario.findById(_id)
```

```

        .then((usuario) => {
            const { contatos } = usuario;
            const contato = contatos.find((ct) => {
                return ct._id.toString() === contatoId;
            });
            res.render('contatos/edit', { contato, usuario });
        })
        .catch(() => res.redirect('/'))
    ;
},

```

As *actions* show e edit possuem o mesmo comportamento. Elas retornam os dados de um específico contato do usuário pela `contatos.find()`, uma função nativa de arrays do JavaScript, que basicamente vai percorrer e buscar, neste caso, um contato com base no `ObjectId(contatoId)`.

```

update(req, res) {
    const contatoId = req.params.id;
    const { contato } = req.body;
    const { usuario } = req.session;
    const where = { _id: usuario._id, 'contatos._id': contatoId }

    const set = { $set: { 'contatos.$': contato } };
    Usuario.update(where, set)
        .then(() => res.redirect('/contatos'))
        .catch(() => res.redirect('/'))
    ;
},

```

Nessa *action*, a implementação do update não tem segredos. Estamos usando a `Usuario.update()`, função que permite atualizar dados de um documento. Neste caso, queremos apenas adicionar um novo item no array de contatos de um usuário.

Para isso, primeiro precisamos buscar o registro corretamente, por meio do trecho: `const where = { _id: usuario._id, 'contatos._id': contatoId }`, e isso é feito pela declaração `const set = { $set: { 'contatos.$': contato } }`.

Ambos os objetos serão enviados para a função `Usuario.update(where, set)`, responsável pela atualização da lista de contatos de um usuário na base de dados. Neste caso, será uma remoção de um contato dessa lista.

```
destroy(req, res) {
  const contatoId = req.params.id;
  const { _id } = req.session.usuario;
  const where = { _id };
  const set = {
    $pull: {
      contatos: { _id: ObjectId(contatoId) }
    }
  };
  Usuario.update(where, set)
    .then(() => res.redirect('/contatos'))
    .catch(() => res.redirect('/'))
;
}
```

Em `destroy`, também será utilizada a função `Usuario.update()`, só que, dessa vez, o objeto `set` usará o atributo `$pull`, responsável por remover um elemento de um array de um documento. No nosso caso, a linha `const set = { $pull: { contatos: { _id: ObjectId(contatoId) } } };` será responsável por essa ação de excluir um contato do array de contatos.

Para terminar, vamos atualizar as views da tela de contatos. 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`, e nele mude apenas a URL da `action` de seus respectivos `form`, para que o atributo `contato._id` seja renderizado:

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

Agora em `views/contatos/index.ejs`, alteraremos as URLs dos links 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 %>">offline</span>
    </td>
    <td>
      <a href="/contato/<%- contato._id %>">Detalhes</a>
      <a href="/chat" id="chat_<%- contato.email %>">Conversar<, a>
      </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 escrita e leitura de forma rápida. 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 elementos chave-valor em sua base de dados, de maneira extremamente rápida, pois mantém os dados na memória em grande parte do tempo, fazendo a sincronização dos

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

As estruturas de dados com as quais ele trabalha são: *Strings*, *Hashes*, *Lists*, *Sets* e *Sorted Sets*. Ele possui um CLI (pelo comando `redis-cli`) que permite brincar em *runtime* com seus inúmeros comandos. Aliás, são vários comandos que realizam operações com os dados, então, 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 sua sala. 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 sua configuração padrão. 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, vamos instalar o seu próprio módulo para o Node.js. No terminal, execute o comando:

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

Com o Redis e seu respectivo driver instalado corretamente, vamos ao que interessa: implementar o histórico do chat. Abra o código `events/chat.js`, pois será nele que vamos conectar o driver ao 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 usamos as configurações padrão do Redis, então não há necessidade de enviar parâmetros extras para a função `createClient()`.

Caso você precise se conectar de um local diferente, apenas inclua o seguinte parâmetro: `createClient(porta, ip)`. Veja como será o nosso código:

```
const redis = require('redis').createClient();  
  
module.exports = (app, io) => {  
  const onlines = {};  
  io.on('connection', (client) => {  
    const { session } = client.handshake;  
    const { usuario } = session;  
    // 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 armazená-las. Cada lista terá uma sala como chave para pesquisa. Primeiro, vamos implementá-la no evento `client.on('create-room')`:

```
client.on('create-room', (hashDaSala) => {  
  session.sala = hashDaSala;
```

```
client.join(hashDaSala);
const resposta = `<b>${usuario.nome}</b> entrou.<br>`;
redis.lpush(hashDaSala, resposta, () => {
    redis.lrange(hashDaSala, 0, -1, (err, msgs) => {
        msgs.forEach((msg) => {
            io.to(hashDaSala).emit('send-client', msg);
        });
    });
});
```

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

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, pelo trecho `io.to(hashDaSala).emit('send-client', msg)`.

Agora, para finalizar o nosso histórico do chat, vamos implementar a função `redis.lpush` nos eventos `send-server` e `disconnect`. Como eles enviam uma única mensagem, simplificaremos o código incluindo a função `redis.lpush(hashDaSala, resposta)`, sem utilizar callbacks. Primeiro, vamos editar o evento `send-server`:

```
client.on('send-server', (hashDaSala, msg) => {
    const resposta = `<b>${usuario.nome}</b> ${msg}<br>`;
    const novaMensagem = { email: usuario.email, sala: hashDaSala
```

```
};

redis.lpush(hashDaSala, resposta);
client.broadcast.emit('new-message', novaMensagem);
io.to(hashDaSala).emit('send-client', resposta);
});
```

Em seguida, faremos o mesmo para o evento `disconnect`:

```
client.on('disconnect', () => {
  const { sala } = session;
  const resposta = `<b>${usuario.nome}</b> saiu.<br>`;
  delete onlines[usuario.email];
  redis.lpush(sala, resposta, () => {
    session.sala = null;
    client.leave(sala);
    client.broadcast.emit('notify-offlines', usuario.email);
    io.to(sala).emit('send-client', resposta);
  });
});
```

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 guardar os e-mails dos usuários online em memória na aplicação, vamos persisti-los também no Redis.

O motivo de fazermos isso é que deixaremos a memória do servidor menos sobrecarregada, delegando toda essa tarefa para o banco de dados, que persiste informação tanto em disco quanto em memória, para um rápido acesso. Entretanto, teremos a vantagem de que, ao isolarmos isso para um banco de dados a parte, (neste caso, o Redis), podemos possibilitar a delegação de todas essas tarefas para um outro servidor dedicado a trabalhar com Redis.

Para realizar essa alteração, vamos usar uma estrutura de dados que só mantém dados distintos, conhecida pelo nome de `Sets`. Para trabalhar com ela, usaremos 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 `events/chat.js`. Nele remova todas as referências à variável `onlines` e altere para as chamadas das funções do Redis. Dessa forma, o loop de e-mails `onlines` vai ficar assim:

```
module.exports = (app, io) => {
  io.on('connection', (client) => {
    const { session } = client.handshake;
    const { usuario } = session;

    redis.sadd('onlines', usuario.email, () => {
      redis.smembers('onlines', (err, emails) => {
        emails.forEach((email) => {
          client.emit('notify-onlines', email);
          client.broadcast.emit('notify-onlines', email);
        });
      });
    });

    client.on('send-server', (hashDaSala, msg) => {
      const resposta = `<b>${usuario.nome}</b> ${msg}<br>`;
      const novaMensagem = { email: usuario.email, sala: hashDaSala };
      redis.lpush(hashDaSala, resposta, () => {
        client.broadcast.emit('new-message', novaMensagem);
        io.to(hashDaSala).emit('send-client', resposta);
      });
    });
    // continuação dos eventos do socket.io...
  });
}
```

Agora, dentro do evento `client.on('disconnect')` , substitua o trecho `delete onlines[usuario.email]` pela função `redis.srem('onlines', usuario.email)` . Assim, delegamos a tarefa de remover informações de um usuário desconectado da sala para o Redis:

```
client.on('disconnect', () => {
  const { sala } = session;
  const resposta = `<b>${usuario.nome}</b> saiu.<br>`;
  redis.lpush(sala, resposta, () => {
    session.sala = null;
    redis.srem('onlines', usuario.email);
    client.leave(sala);
    client.broadcast.emit('notify-offlines', usuario.email);
    io.to(sala).emit('send-client', resposta);
  });
});
```

Mais uma vez, terminamos um excelente capítulo! Agora temos uma aplicação 100% funcional que utiliza dois bancos de dados NoSQL. Acredite, o que 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, pois ele possui uma suíte completa de funcionalidade para a criação de diversos testes, reports customizados, integração com ferramentas para cobertura de testes e muito mais. Seu site é <https://mochajs.org>.



Figura 8.1: Mocha – Framework para testes

Ele foi feito pelo TJ Holowaychuk, o mesmo criador do framework Express e, hoje, após ele sair da liderança desses projetos, uma grande comunidade os mantém ativos e atualizados. Ele 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`.

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

8.2 CRIANDO UM AMBIENTE 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 este de testes ou de desenvolvimento.

Para identificar em qual ambiente está o projeto, usamos a variável `process.env.NODE_ENV`. Ela vai procurar se existe em seu sistema operacional.

Vamos criar um novo diretório, responsável por manter módulos genéricos, que se chamará de `libs`. Nele, vamos incluir códigos utilitários. Nesta pasta, vamos criar o arquivo `libs/db.js`, responsável pela conexão com o banco de dados, de

acordo com o ambiente de desenvolvimento:

```
const mongoose = require('mongoose');
const bluebird = require('bluebird');
const currentEnv = process.env.NODE_ENV || 'development';
const envURL = {
  test: 'mongodb://localhost:27017/ntalk_test',
  development: 'mongodb://localhost:27017/ntalk'
};
mongoose.Promise = bluebird;
module.exports = mongoose.connect(envURL[currentEnv]);
```

LENDÔ VARIÁVEIS DE AMBIENTE NO Node

Quando se trabalha com variáveis de ambiente, é muito comum persistir dados de configurações ou dados sensíveis no sistema operacional. A URL de acesso ao 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, faremos o mesmo procedimento. No Node.js, podemos lê-las por meio do `process.env['VARIABEL']` , 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` , como também toda a referência de uso desse módulo. 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.js` será carregado diretamente no modelo `models/usuario.js`:

```
const db = require('../libs/db.js');
const Schema = require('mongoose').Schema;

module.exports = () => {
  const contato = Schema({
    nome: String,
    email: String
  });
  const usuario = Schema({
    nome: {
      type: String,
      required: true
    },
    email: {
      type: String,
      required: true,
      index: { unique: true }
    },
    contatos: [contato]
  });
  return db.model('usuarios', usuario);
};
```

Dessa forma, não manteremos em nossa aplicação nenhuma variável global. Já a variável `db`, que mantém uma conexão com MongoDB, será gerenciada pelo `libs/db.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 um banco de dados, de acordo com sua variável de ambiente. Porém, em aplicações mais complexas, usa-se muito esse 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 usarmos seu CLI. Em seu console, execute o comando:

```
npm install -g mocha
```

O Mocha é um módulo focado em testes e vamos adicioná-lo ao `package.json`, porém, ele não será incluído dentro de `dependencies`, e sim como `devDependencies`, para que seja possível adotar a boa prática de separar módulos importantes do projeto de módulos que só auxiliam nos testes do projeto.

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 a interface BDD (*Behavior Driven-Development*) para usar funções como `describe`, `it`, `beforeEach`, `afterEach`, entre outras que auxiliam na criação dos testes. A função `should` é responsável por fazer verificações assertivas nos resultados de cada teste, entretanto, ela não vem nativamente no framework Mocha. Com isso, teremos de habilitá-la.

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

Para explorarmos o Mocha e por conta de fins didáticos,

simplificaremos essa criação dos testes, criando apenas testes em cima das rotas de nossa 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 isso permite verificar como será o comportamento de uma requisição feita por um usuário. Para realizar esses testes, usaremos o módulo `supertest`, que também nasceu pelos 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 que codificaremos os testes da aplicação.

8.4 RODANDO O MOCHA NO AMBIENTE DE TESTES

Assim como criamos o `libs/db.js` – uma simples biblioteca que retorna uma conexão MongoDB, baseada no valor da variável `NODE_ENV` –, temos de executar os testes utilizando essa variável com valor '`test`' para que eles rodem no devido ambiente. Para isso, um simples comando no terminal, `mocha test`, já será o suficiente, mas neste caso não será executado no ambiente de testes.

O comando correto é `NODE_ENV=test mocha test --exit`, pois ele define o valor de `NODE_ENV` para `test`, e a flag `--exit` vai garantir que o processo do teste seja finalizado após toda a sua execução. Esse parâmetro torna-se necessário desde a versão `4.0.0`, pois, por default, ele mantém o processo do Mocha em

execução após realizar todos os testes.

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

Uma boa prática para simplificar este comando é usar 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 mocha test/**/*.js --exit"  
}
```

Caso esteja no Windows, faça o seguinte:

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

Foram adicionados dois comandos dentro do atributo scripts: o start e o test. Estes são os 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 serão executados via npm run [nome-do-comando], inclusive o npm run test e o 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 usamos 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 (<https://travis-ci.org>), um serviço de *deploy contínuo* compatível com diversas linguagens, inclusive Node.js. Ele roda os testes do seu projeto pelo comando `npm test`, e não instala módulos globais, apenas utiliza os 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 o seguinte trecho na última linha do `app.js`:

```
// 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.

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

describe('No controller home', () => {
  it('GET "/" retorna status 200', (done) => {
    request.get('/').end((err, res) => {
      res.status.should.eql(200);
      done();
    });
  });
  // continuação dos testes...
```

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

```
it('GET "/sair" redireciona para GET "/"', (done) => {
  request.get('/sair').end((err, res) => {
    res.headers.location.should.eql('/');
    done();
  });
});
// continuação dos testes...
```

Agora vamos trabalhar com testes em cima de rotas mais complexas. Simularemos um `POST` enviando parâmetros válidos (`nome` e `email`) , que faz um login e é redirecionado para a rota `"/contatos"` .

```
it('POST "/entrar" válido redireciona para GET "/contatos"', (done) => {
  const usuario = { nome: 'Teste', email: 'teste@teste' };
  request.post('/entrar')
    .send({ usuario })
```

```
.end((err, res) => {
  res.headers.location.should.eql('/contatos');
  done();
});
});
// continuaçao dos testes...
```

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

```
it('POST "/entrar" inválido redireciona para GET "/"', (done)
=> {
  const usuario = { nome: '', email: '' };
  request.post('/entrar')
    .send({ usuario })
    .end((err, res) => {
      res.headers.location.should.eql('/');
      done();
    });
});
}); // fim da função describe()
```

Esse foi o nosso teste com a rota `routes/home.js`. Repare que cada teste ficou bem descritivo e atômico, realizando apenas uma única verificação por meio da função `should`. Afinal, essa é uma das boas práticas de criação de testes, pois, dessa forma, eles vão ajudar o desenvolvedor no entendimento melhor do que cada código faz.

Vamos rodar os testes para ver se deu tudo certo? Para executá-los, é simples! Como já configuramos o script no `package.json` para a execução de testes, execute no terminal o comando `npm test`. Veja o resultado dos seus testes semelhante ao da figura:

```
[caio:ntalk] (v2017) $ npm test
> ntalk@0.0.1 test /Users/caio/Workspace/nodejs/livro-nodejs/capitulo-8/ntalk
> NODE_ENV=test mocha test/**/*.js --exit

Ntalk no ar.
Controller home
  ✓ GET "/" retorna status 200 (49ms)
  ✓ GET "/sair" redireciona para GET "/"
  ✓ POST "/entrar" válido redireciona para GET "/contatos" (61ms)
  ✓ POST "/entrar" inválido redireciona para GET "/"

  4 passing (161ms)

[caio:ntalk] (v2017) $
```

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, apareceu o resultado dos testes. Isso acontece porque, em `app.js`, exportamos a variável que contém uma instância de um servidor da aplicação, a `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 nele, assim permitindo 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 para um usuário logado, e um para um usuário não logado.

Para isso, crie o arquivo `test/requests/contatos.js`. Nele vamos criar um `describe` com duas funções `describe` interna,

que serão usadas para descrever esses testes.

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

describe('No controller contatos', () => {
  describe('usuario nao logado', () => {
    // testes aqui...
  });

  describe('usuario logado', () => {
    // testes aqui...
  });
});
```

No caso do 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, e este comportamento seja testado.

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

```
it('GET "/contatos" redireciona para GET "/"', (done)=> {
  request.get('/contatos').end((err, res) => {
    res.headers.location.should.eql('/');
    done();
  });
});
it('GET "/contato/1" redireciona para GET "/"', (done)=> {
  request.get('/contato/1').end((err, res) => {
    res.headers.location.should.eql('/');
    done();
  });
});
it('GET "/contato/1/editar" redireciona para GET "/"', (done)=>
{
  request.get('/contato/1/editar').end((err, res) => {
```

```
    res.headers.location.should.eql('/');
    done();
  });
});
it('POST "/ contato" redireciona para GET "/"', (done)=> {
  request.post('/ contato').end((err, res) => {
    res.headers.location.should.eql('/');
    done();
  });
});
it('DELETE "/ contato/1" redireciona para GET "/"', (done)=> {
  request.del('/ contato/1').end((err, res) => {
    res.headers.location.should.eql('/');
    done();
  });
});
it('PUT "/ contato/1" redireciona para GET "/"', (done)=> {
  request.put('/ contato/1').end((err, res) => {
    res.headers.location.should.eql('/');
    done();
  });
});
```

Basicamente, todos tiveram um resultado parecido, pois, nesse caso, estamos simulando o acesso de um usuário não logado em rotas que exigem dados de usuário logado. Com isso, o comportamento padrão de nossa aplicação será de redirecionar cada acesso não autorizado para a rota principal, a / .

Mais uma vez, vamos rodar os testes para ter a certeza de que tudo ocorreu bem na sua implementação. Se tudo estiver ok, um resultado semelhante à figura a seguir será exibido:

```
1. bash
[caio:ntalk] (v2017) $ npm test

> ntalk@0.0.1 test /Users/caio/Workspace/nodejs/livro-nodejs/capitulo-8/ntalk
> NODE_ENV=test mocha test/**/*.js --exit

Ntalk no ar.
No controller contatos
  usuario nao logado
    ✓ GET "/contatos" redireciona para GET "/"
    ✓ GET "/contato/1" redireciona para GET "/"
    ✓ GET "/contato/1/editar" redireciona para GET "/"
    ✓ POST "/contato" redireciona para GET "/"
    ✓ DELETE "/contato/1" redireciona para GET "/"
    ✓ PUT "/contato/1" redireciona para GET "/"

No controller home
  ✓ GET "/" retorna status 200
  ✓ GET "/sair" redireciona para GET "/"
  ✓ POST "/entrar" válido redireciona para GET "/contatos" (47ms)
  ✓ POST "/entrar" inválido redireciona para GET "/"

  10 passing (174ms)

[caio:ntalk] (v2017) $
```

Figura 8.3: Resultado dos testes

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

1. **Antes**, cada teste deve fazer uma requisição `POST` na rota de login;
2. O mesmo teste deve manter um **cookie** válido, gerado após o login de sucesso, para pular o filtro.

Para implementar essa rotina, vamos usar a função `beforeEach()`, executada antes de cada teste. Dentro dela, faremos um login no sistema para capturar o seu `cookie`, que é encontrado dentro de `res.headers['set-cookie']`.

Com isso, armazenaremos seu resultado em uma variável que estará no mesmo escopo da função `describe('usuario logado')`, para que ela seja reutilizada em cada um de seus testes. Para entender melhor, veja o seguinte código:

```
describe('usuario logado', function() {
  const usuario = { nome: 'Teste', email: 'teste@teste' };
  const contato = usuario;
  let cookie = null;

  beforeEach((done) => {
    request.post('/entrar')
      .send({ usuario })
      .end((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 faremos isso via `req.cookies = cookie`. Vamos implementá-los?

A seguir, vamos usar 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('GET "/contatos" retorna status 200', (done) => {
  const req = request.get('/contatos');
  req.cookies = cookie;
  req.end((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('POST "/contato" redireciona para GET "/contatos"', (done) => {
  const contato = usuario;
  const req = request.post('/contato');
  req.cookies = cookie;
  req.send({ contato }).end((err, res) => {
    res.headers.location.should.eql('/contatos');
    done();
  });
});
}); // fim da função describe('usuario logado')
}); // fim da função describe principal
```

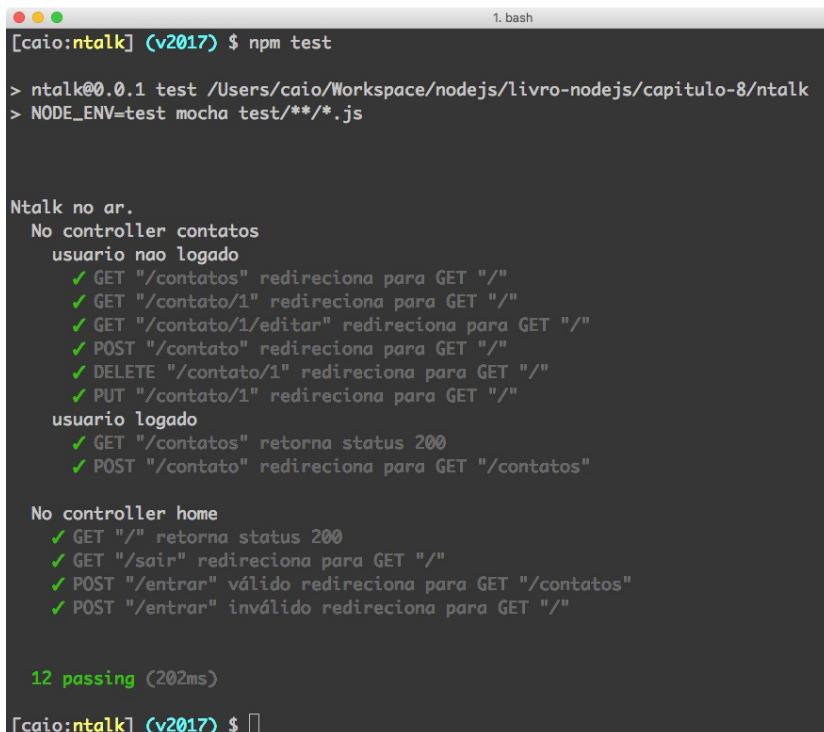
Então, vem a pergunta: *por que não foram testadas todas as rotas para um usuário logado?*

Testes de aceitação verificam o comportamento do sistema ao simular uma requisição real, pela 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. Porém, caso queira testar essas rotas, você pode criar objetos *fakes* no banco de dados de ambiente de testes, para utilizarem um id fixo que possibilitaria a elaboração desses testes de forma mais precisa.

Para finalizarmos a implementação dos testes, execute novamente o npm test , e veja se o resultado será semelhante ao

da figura a seguir.



```
1. bash
[caio:ntalk] (v2017) $ npm test
> ntalk@0.0.1 test /Users/caio/Workspace/nodejs/livro-nodejs/capitulo-8/ntalk
> NODE_ENV=test mocha test/**/*.js

Ntalk no ar.

No controller contatos
  usuario nao logado
    ✓ GET "/contatos" redireciona para GET "/"
    ✓ GET "/contato/1" redireciona para GET "/"
    ✓ GET "/contato/1/editar" redireciona para GET "/"
    ✓ POST "/contato" redireciona para GET "/"
    ✓ DELETE "/contato/1" redireciona para GET "/"
    ✓ PUT "/contato/1" redireciona para GET "/"

  usuario logado
    ✓ GET "/contatos" retorna status 200
    ✓ POST "/contato" redireciona para GET "/contatos"

No controller home
  ✓ GET "/" retorna status 200
  ✓ GET "/sair" redireciona para GET "/"
  ✓ POST "/entrar" valido redireciona para GET "/contatos"
  ✓ POST "/entrar" invalido redireciona para GET "/"

12 passing (202ms)

[caio:ntalk] (v2017) $
```

Figura 8.4: Resultado final dos 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, é necessário declará-las. Porém, é possível criar um arquivo de configuração do próprio Mocha para que elas sejam centralizadas 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 usar 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  
--exit
```

Com isso, podemos simplificar os atuais alias no `package.json`:

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

Caso esteja no Windows, faça o seguinte:

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

Além de carregarmos o `should`, também foi adicionado um novo parâmetro, o `--exit`, que finaliza o processo dele após executar todos os testes. Caso contrário, o processo do mocha continuará aberto depois de ele executar todos os testes.

Caso queria incluir outros parâmetros em seu `mocha.opts`, execute o comando `./node_modules/.bin/mocha -h` no terminal, para visualizar todas as opções possíveis de configuração.

```
1.bash
[caio:ntalk] [v2017] $ ./node_modules/.bin/mocha -h
Usage: mocha [debug] [options] [files]

Options:

-V, --version          output the version number
-A, --async-only       force all tests to take a callback (async) or return a promise
-c, --colors            force enabling of colors
-C, --no-colors        force disabling of colors
-G, --growl             enable growl notification support
-O, --reporter-options <k=v,k2=v2,...> reporter-specific options
-R, --reporter <name>  specify the reporter to use
-S, --sort               sort test files
-b, --bail              bail after first test failure
-d, --debug             enable node's debugger, synonym for node --debug
-g, --grep <pattern>   only run tests matching <pattern>
-f, --fgrep <string>   only run tests containing <string>
-gc, --expose-gc        expose gc extension
-i, --invert            inverts --grep and --fgrep matches
-r, --require <name>    require the given module
-s, --slow <n>           "slow" test threshold in milliseconds [75]
-t, --timeout <n>        set test-case timeout in milliseconds [2000]
-u, --ui <name>         specify user-interface (bdd|tdd|qunit|exports)
-w, --watch              watch files for changes
--check-leaks           check for global variable leaks
--full-trace            display the full stack trace
--compilers <ext>:<module>,... use the given module(s) to compile files
--debug-brk              enable node's debugger breaking on the first line
--globals <names>        allow the given comma-delimited global [names]
--es_staging             enable all staged features
--harmony<_classes,_generators,...> all node --harmony* flags are available
--preserve-symlinks      Instructs the module loader to preserve symbolic links when resolving and

caching modules
--icu-data-dir          include ICU data
--inline-diffs          display actual/expected differences inline within each string
--no-diff                do not show a diff on failure
--inspect               activate devtools in chrome
--inspect-brk           activate devtools in chrome and break on the first line
--interfaces            display available interfaces
```

Figura 8.5: Parâmetros opcionais do Mocha – Parte 1

```
1.bash
--no-deprecation           silence deprecation warnings
--exit                      force shutdown of the event loop after test run: mocha will call process.
exit                         disables timeouts, given implicitly with --debug
                             silence all node process warnings
--no-timeouts               specify opts path
--no-warnings                enable perf linux profiler (basic support)
--opts <path>                 enable experimental NAPI modules
--perf-basic-prof            log statistical profiling information
--napi-modules                Time events including external callbacks
--prof                         include sub directories
--log-timer-events            display available reporters
--recursive                   set numbers of time to retry a failed test case
--reporters                  throw an exception anytime a deprecated function is used
--retries <times>             trace function calls
--throw-deprecation           show stack traces on deprecations
--trace                        show stack traces on node process warnings
--trace-deprecation           enforce strict mode
--trace-warnings              additional extensions to monitor with --watch
--use_strict                  wait for async suite definition
--watch-extensions <ext>,...  enable uncaught errors to propagate
--delay                         causes test marked with only to fail the suite
--allow-uncaught               causes pending tests and test marked with skip to fail the suite
--forbid-only                 output usage information

Commands:
  init <path>  initialize a client-side mocha setup at <path>
[caio:ntalk] (v2017) $
```

Figura 8.6: Parâmetros opcionais do Mocha – Parte 2

Agora, só para finalizar, remova a função `const should = require('should')` de todos os testes criados, pois agora eles serão automaticamente carregados como escopo global via `mocha.opts`. Assim, deixaremos os testes *mais clean*, sem a necessidade de carregar esse módulo repetidamente em todos eles.

CAPÍTULO 9

APLICAÇÃO NODE EM PRODUÇÃO – PARTE 1

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 *logs*.

9.1 CONFIGURANDO CLUSTERS

Infelizmente, o Node.js não trabalha com *threads*. Isso é algo que, na opinião de alguns desenvolvedores, é considerado 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 encarrega-se 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 o ideal é 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 abstraída para o desenvolvedor.

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

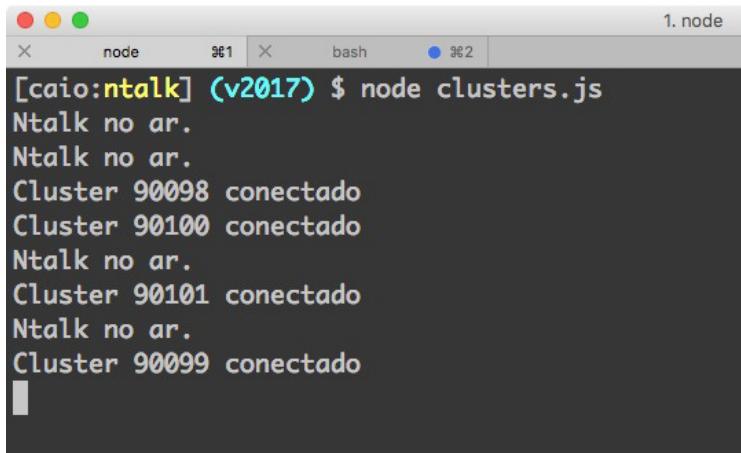
Com base nesses conceitos, vamos aplicar na prática a implementação de *clusters*. Crie no diretório raiz o arquivo `clusters.js`, para que, por meio dele, sejam carregados os clusters da nossa aplicação. Veja o código a seguir:

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

```
const cpus = require('os').cpus();

if (cluster.isMaster) {
  cpus.forEach(() => cluster.fork());
  cluster.on('listening', (worker) => {
    console.log(`Cluster ${worker.process.pid} conectado`);
  });
  cluster.on('disconnect', (worker) => {
    console.log(`Cluster ${worker.process.pid} desconectado`);
  });
  cluster.on('exit', (worker) => {
    console.log(`Cluster ${worker.process.pid} finalizado`);
  });
} 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 a mensagem "Ntalk no ar" se repetiu.



```
[caio:ntalk] (v2017) $ node clusters.js
Ntalk no ar.
Ntalk no ar.
Cluster 90098 conectado
Cluster 90100 conectado
Ntalk no ar.
Cluster 90101 conectado
Ntalk no ar.
Cluster 90099 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 no trecho `cpus.forEach()` – e esse trecho 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) e, consequentemente, ele não cai na condicional `if(cluster.isMaster)`, 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 finalizado 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 (<https://nodejs.org/api/cluster.html>) para ficar por dentro de todos os eventos e funções deste módulo.

Para finalizar e deixar automatizado o comando `start` do servidor em modo *cluster* (via comando `npm`), atualize em seu `package.json` o atributo `scripts` , de acordo com o código a seguir:

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

Caso esteja no Windows, faça o seguinte:

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

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

9.2 REDIS CONTROLANDO AS SESSÕES DA APLICAÇÃO

No Node.js, quando desenvolvemos uma aplicação orientada a *clusters*, aliada 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, está tudo bem, correto? Tente cadastrar um novo contato ou ver os detalhes de um existente. Repare que automaticamente você foi redirecionado para a tela de login. Mas que estranho! Por que aconteceu isso?

No momento, usamos um controle de sessão em memória (pelo objeto `expressSession.MemoryStore`). A natureza desse tipo de controle não consegue compartilhar dados entre os *clusters*, já que 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 armazenamento de sessão, e o Redis é uma ótima alternativa. Caso já tivéssemos usado-o dentro do chat da aplicação, teríamos 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. Logo, vamos implantar 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 o seguinte comando no terminal:

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

Agora, vamos modificar o `app.js` para usar esses módulos. Assim, toda sessão do Express e do Socket.IO será gerenciada pelo Redis com os módulos `connect-redis` e `socket.io-redis`:

```
const express = require('express');
const path = require('path');
const http = require('http');
const socketIO = require('socket.io');
const consign = require('consign');
const bodyParser = require('body-parser');
const cookie = require('cookie');
const expressSession = require('express-session');
const methodOverride = require('method-override');
const config = require('./config');
const error = require('./middlewares/error');
const redisAdapter = require('socket.io-redis');
const RedisStore = require('connect-redis')(expressSession)

const app = express();
const server = http.Server(app);
const io = socketIO(server);
const store = new RedisStore({ prefix: config.sessionKey });

// carregamento dos middlewares ...
io.adapter(redisAdapter());
io.use((socket, next) => {
    const cookieData = socket.request.headers.cookie;
    const cookieObj = cookie.parse(cookieData);
    const sessionHash = cookieObj[config.sessionKey] || '';
    const sessionId = sessionHash.split('.')[0].slice(2);
    store.get(sessionId, (err, currentSession) => {
        if (err) {
            return next(new Error('Acesso negado!'));
        }
        socket.handshake.session = currentSession;
        return next();
    });
    return true;
});
```

```
// continuação do app.js ...
```

Com esse upgrade implementado, agora o sistema vai rodar perfeitamente em *clusters*, sem causar bugs no controle de sessão, sendo este compartilhado entre os *clusters* existentes. Também foi mudado o `store.all` para `store.get`, para que sejam consultados dados de apenas um único `sessionID`. Afinal, é mais otimizado e recomendado o uso da função que vem da implementação do `RedisStore` do que o `MemoryStore`, já que se torna viável delegar toda sessão para um servidor Redis dedicado em vez de deixar na memória do servidor de nossa aplicação.

9.3 MONITORANDO APLICAÇÃO POR MEIO DE LOGS

Quando colocamos um sistema em produção, aumentamos o risco de acontecer bugs que não foram identificados durante os testes de desenvolvimento. Isso é normal, e toda aplicação já passou ou vai passar por essa situação. Neste caso, o importante é ter um meio de monitorar todo o comportamento da aplicação, por meio de arquivos de *logs*.

Esses arquivos de logs vão mostrar todo output da aplicação, que é mostrado no terminal, só que em arquivos de texto. Com isso, será viável acessar históricos desses logs, uma vez que não é nada legal ficar com o terminal aberto 24 horas para olhar os outputs da aplicação, não é mesmo?

Tanto o Express quanto o Socket.IO possuem *middlewares* para gerar *logs*. Para configurar um modo debug em nossa aplicação, para que seus *logs* sejam gerados, é muito simples! Você

precisa apenas iniciar o servidor usando a variável `DEBUG` , ou seja, é só rodar a aplicação com este comando:

```
DEBUG=* node clusters.js
```

No Windows, faremos desta forma:

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

Agora o seu sistema está gerando logs com maiores detalhes de todo o 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 o seu console aberto para vê-los, afinal, você também tem uma vida para viver no mundo real!

Se acontecer um problema no sistema *do nada*, e você não estiver presente para ver o erro gerado, você perderá informações úteis de debug da aplicação. Para resolver esse problema, é necessário rodar um 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 a escrevê-los dentro do arquivo `app.log` .

Para simplificar ainda mais, vamos manter esse comando dentro do *alias* `npm start` . Assim, ao iniciar o servidor de nossa aplicação, automaticamente todo output será enviado para um arquivo de log. Para isso, edite o `package.json` na seguinte linha:

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

No Windows, faremos desta forma:

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

Agora sua aplicação está preparada para gerar arquivos 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:

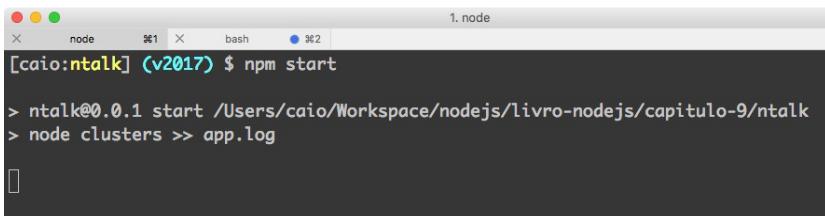
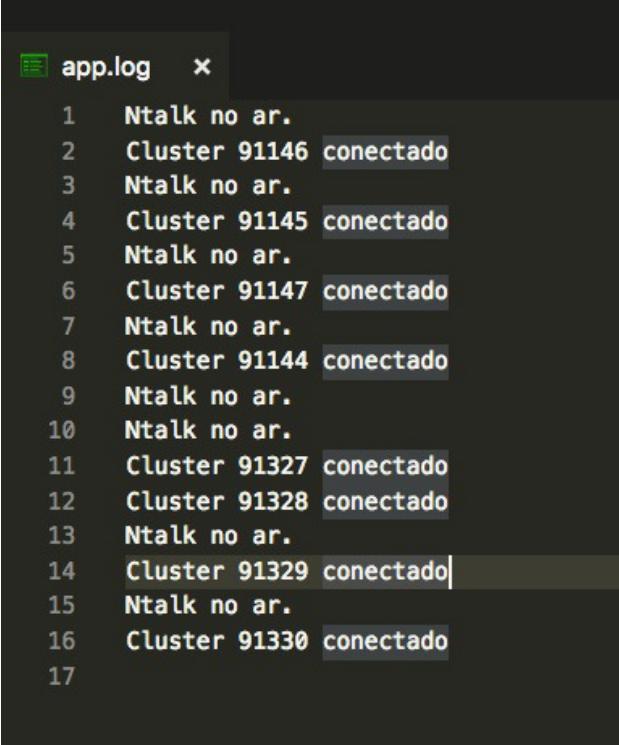


Figura 9.2: Tela do terminal não emitindo logs

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



The screenshot shows a terminal window with the title bar "app.log". The log file contains 17 numbered entries. Most entries consist of the text "Ntalk no ar." followed by "Cluster" and a port number (e.g., 91146, 91145, 91147, 91327, 91328, 91329, 91330) with the word "conectado" in white text on a black background. Entry 14 is partially highlighted with a dark grey background, showing "Cluster 91329 conectado|".

```
1 Ntalk no ar.  
2 Cluster 91146 conectado  
3 Ntalk no ar.  
4 Cluster 91145 conectado  
5 Ntalk no ar.  
6 Cluster 91147 conectado  
7 Ntalk no ar.  
8 Cluster 91144 conectado  
9 Ntalk no ar.  
10 Ntalk no ar.  
11 Cluster 91327 conectado  
12 Cluster 91328 conectado  
13 Ntalk no ar.  
14 Cluster 91329 conectado|  
15 Ntalk no ar.  
16 Cluster 91330 conectado  
17
```

Figura 9.3: Logs da aplicação no arquivo app.log

9.4 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 para o Mongoose, com o objetivo de otimizar tanto o *server-side* quanto 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 carrega e executa todos os seus submódulos. Vamos começar otimizando o Express.

Faremos nele duas otimizações: habilitar compactação **Gzip** e adicionar um simples *cache* de arquivos estáticos – incluindo o atributo `maxAge` dentro de `express.static`.

Dessa forma, otimizaremos o carregamento de arquivos estáticos no servidor, compactando-os com `gzip` e adotando um cache neles para sobrecarregar menos o nosso servidor. Primeiro, instale o módulo `compression` para trabalhar com o `gzip`:

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

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

```
const express = require('express');
const path = require('path');
const http = require('http');
const socketIO = require('socket.io');
const consign = require('consign');
const bodyParser = require('body-parser');
const cookie = require('cookie');
const compression = require('compression');
const expressSession = require('express-session');
const methodOverride = require('method-override');
const config = require('./config');
const error = require('./middlewares/error');
const redisAdapter = require('socket.io-redis');
const RedisStore = require('connect-redis')(expressSession)

const app = express();
const server = http.Server(app);
const io = socketIO(server);
const store = new RedisStore({ prefix: config.sessionKey });

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
// Carregando módulo de compactação gzip
app.use(compression());
app.use(expressSession({
  store,
  name: config.sessionKey,
```

```
    secret: config.sessionSecret
});
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(methodOverride('_method'));
// Incluindo maxAge para manter em cache arquivos estáticos
app.use(
  express.static(
    path.join(__dirname, 'public'),
    { maxAge: 3600000 } // milisegundos
  )
);
// continuação do app.js ...
```

Calma! Falta pouco para terminar este livro! Nas próximas páginas, continuaremos a segunda parte dele, apresentando mais dicas importantes para você preparar sua aplicação e rodá-la em um ambiente de produção.

CAPÍTULO 10

APLICAÇÃO NODE EM PRODUÇÃO – PARTE 2

10.1 MANTENDO A APLICAÇÃO PROTEGIDA

Para evitar alguns problemas de *hacking* ou adulteração de dados em nossa aplicação, vamos incluir alguns módulos que visam torná-la mais segura para os usuários.

Prevenindo ataques XSS

Aplicaremos 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, para que seja ativado o módulo `csurf` em nossa aplicação. Para isso, edite o `app.js` e adicione esse

middleware no último lugar, com o seguinte código:

```
const express = require('express');
const path = require('path');
const http = require('http');
const socketIO = require('socket.io');
const consign = require('consign');
const bodyParser = require('body-parser');
const csurf = require('csurf');
const cookie = require('cookie');
const compression = require('compression')
const expressSession = require('express-session');
const methodOverride = require('method-override');
const config = require('./config');
const error = require('./middlewares/error');
const redisAdapter = require('socket.io-redis');
const RedisStore = require('connect-redis')(expressSession)

const app = express();
const server = http.Server(app);
const io = socketIO(server);
const store = new RedisStore({ prefix: config.sessionKey });

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
app.use(compression());
app.use(expressSession({
    store,
    name: config.sessionKey,
    secret: config.sessionSecret
}));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(methodOverride('_method'));
app.use(
    express.static(
        path.join(__dirname, 'public'),
        { maxAge: 3600000 }
    )
);
app.use(csurf());
app.use((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 por meio do código `res.locals._csrf`, visível pelas views pela 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 os ataques XSS.

Para aplicar esse token nos formulários, adicionaremos a tag `<input type="hidden" name="_csrf" value="<%- _csrf %>">`. Assim, cada submissão de formulário será validada pelo servidor com um token que é autogerado a cada requisição.

Então, vamos lá! Editaremos todos os formulários para incluir essa tag pelo módulo `csurf`, para garantir que toda requisição seja segura contra esse tipo de ataque. Para isso, vamos editar alguns arquivos. Primeiro, configure o arquivo `views/home/index.ejs` da seguinte maneira:

```
<% 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">  
    <br>  
    <input type="text" name="usuario[email]" placeholder="Email">  
  >  
  <br>  
  <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</a>
    </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 `<input type="hidden" name="_csrf" value="<%- _csrf %>">` gera 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[nome]" placeholder="Nome">
        <br>
        <input type="text" name="usuario[email]" placeholder="E-mail">
        <br>
        <button type="submit">Entrar</button>
      </form>
    </section>
  ▶<footer>...</footer>
</body>
</html>
```

Token csrf



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 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 o protocolo HTTP, por exemplo.

Para trabalhar com HTTP, temos de 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.

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, 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 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 é de forma programável em código Node.js, utilizando o módulo `forever-monitor`. Esta última é a mais recomendada 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 a configuração feita via JavaScript, e também por 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 um diretório vazio na raiz chamado de `logs`. Em seguida, abra o `server.js` e implemente o código que será responsável por carregar o `forever-monitor`, que vai gerenciar o processo de nossa aplicação:

```
const { Monitor } = require('forever-monitor');
const 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', () => console.log('Servidor foi finalizado.'))
;

child.start();
```

Tudo começa quando instanciamos o objeto `const { Monitor } = require('forever-monitor')`. Nele, passamos dois parâmetros em seu construtor: o primeiro é o código da

aplicação que desejamos executar (no nosso caso, o `cluster.js`). Já no segundo, passamos um objeto com os atributos de configuração do `forever-monitor`.

Em `max`, definimos o total de vezes que poderemos reiniciar o servidor quando ele cair. Vale lembrar de que, ao passar este total, sua aplicação será totalmente finalizada. Infelizmente, essa é uma limitação do `forever-monitor`, pois, por ser um CLI, ele permite reiniciar infinitamente a 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 mocha test/**/*.js"  
}
```

Ou se você estiver no Windows, faça o seguinte:

```
"scripts": {  
  "start": "SET DEBUG=* && node server",  
  "test": "SET NODE_ENV=test && 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 sua aplicação Node.js será hospedada. Vale lembrar de que, em ambientes limitados que não permitem instalar CLIs, a melhor alternativa é usar o `forever-monitor`.

10.3 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.

Essas informações serão repassadas para o atual `config.js`. Para o nosso caso, vamos manter informações de conexão do MongoDB, Redis e cache estático. Dessa forma, será adotada a boa prática de externalizar as variáveis de configurações de serviços, usadas em nossa aplicação, em um único arquivo. Então, edite o arquivo `config.js` e inclua os seguintes atributos:

```
const sessionKey = 'ntalk.id';
const sessionSecret = 'ntalk_secret';

module.exports = {
  sessionKey,
  sessionSecret,
```

```

env: process.env.NODE_ENV || 'development',
mongodb: {
  test: 'mongodb://localhost:27017/ntalk_test',
  development: 'mongodb://localhost:27017/ntalk'
},
mongoose: {
  useMongoClient: true
},
forever: {
  max: 10,
  silent: true,
  killTree: true,
  logFile: 'logs/forever.log',
  outFile: 'logs/app.log',
  errFile: 'logs/error.log'
},
redis: {
  host: 'localhost',
  port: 6379
},
redisStore: {
  prefix: sessionKey
},
cache: {
  maxAge: 3600000
}
};

```

Agora, voltando ao `app.js`, vamos utilizar esses atributos por meio do carregamento e uso do `config.js`:

```

const express = require('express');
const path = require('path');
const http = require('http');
const socketIO = require('socket.io');
const consign = require('consign');
const bodyParser = require('body-parser');
const cookie = require('cookie');
const compression = require('compression')
const expressSession = require('express-session');
const methodOverride = require('method-override');
const config = require('./config');
const error = require('./middlewares/error');
const redisAdapter = require('socket.io-redis');

```

```

const RedisStore = require('connect-redis')(expressSession)

const app = express();
const server = http.Server(app);
const io = socketIO(server);
const store = new RedisStore(config.redisStore);

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
app.use(compression());
app.use(expressSession({
  store,
  name: config.sessionKey,
  secret: config.sessionSecret
}));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(methodOverride('_method'));
app.use(
  express.static(
    path.join(__dirname, 'public'),
    config.cache
  )
);

```

Também usaremos essas configurações no trecho de código responsável por configurar o Socket.IO no `app.js`:

```

io.adapter(redisAdapter(config.redis));
io.use((socket, next) => {
  const cookieData = socket.request.headers.cookie;
  const cookieObj = cookie.parse(cookieData);
  const sessionHash = cookieObj[config.sessionKey] || '';
  const sessionID = sessionHash.split('.')[0].slice(2);
  store.get(sessionID, (err, currentSession) => {
    if (err) {
      return next(new Error('Acesso negado!'));
    }
    socket.handshake.session = currentSession;
    return next();
  });
  return true;
});

```

Em seguida, edite o arquivo `server.js` para carregar as configurações do módulo `forever` via `config.js`:

```
const { Monitor } = require('forever-monitor');
const config = require('./config.js');
const child = new Monitor('clusters.js', config.forever);

child.on('exit', () => console.log('Servidor foi finalizado.'))
;
child.start();
```

Também será necessário atualizar a conexão do Redis em `events/chat.js`, para que ele inicie o cliente `redis` de acordo com o `host` e o `port`, informados pelo `config.redis`. Veja como ficará:

```
const config = require('../config.js');
const redis = require('redis').createClient(config.redis);

// continuação do events/chat.js
```

Para finalizar, adotaremos essa mesma prática no `libs/db.js`, fazendo as seguintes modificações:

```
const mongoose = require('mongoose');
const bluebird = require('bluebird');
const config = require('../config.js');
const host = config.mongodb[config.env];
mongoose.Promise = bluebird;

module.exports = mongoose.connect(host, config.mongoose);
```

Com isso, todas as configurações de nossa aplicação estão centralizadas em um único arquivo, e isso nos permitirá criar arquivos de configurações diferentes para cada ambiente, por exemplo, sejam eles de desenvolvimento, de testes ou de ambiente de produção.

Agora, nossa aplicação está otimizada e corretamente

configurada 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, e um meio de persistência de dados usando o MongoDB. Também implantamos o Redis, e criamos um meio da nossa aplicação interagir em tempo real por uma comunicação bidirecional com o Socket.IO.

Configuramos clusters, logs e 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, pois, 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. Dessa forma, delegaremos todo o trabalho de servir arquivos estáticos para um servidor mais robusto para fazer isso, deixando nossa aplicação Node.js dedicada apenas para o processamento de dados.



Figura 11.1: Servidor Nginx

O objetivo dessa integração é aumentar a performance da aplicação, por isso criaremos um *proxy* do Nginx com Node.js. Isso vai diminuir o número de requisições diretas em nossa aplicação Node.js, já que, em grande parte dos casos, serão realizadas várias requisições aos arquivos estáticos. Ao delegarmos isso para um servidor que possui características dedicadas a servir arquivos estáticos, vamos reduzir a sobrecarga dessa tarefa, logo, nossa aplicação Node.js focará apenas no processamento de dados dos usuários e do chat.

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. Com isso, 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 a compactação `gzip` nativa, com o trecho `gzip on;`, e criamos dois `locations` dentro de `server`. O primeiro `location` é o responsável por servir um 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, por meio 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 – algo semelhante a `root C:/www/ntalk/public` , ou para qualquer outro endereço em que você deseje manter os arquivos estáticos deste projeto.

Também aplicamos dentro desse `location` um *cache* dos arquivos por meio do item `expires max;` . No último `location` , aplicamos um simples controle de *proxy*. O item `proxy_pass` praticamente redireciona as demais rotas para a nossa aplicação, que estará ativa em `http://localhost:3000` .

```
location / {  
    proxy_pass http://localhost:3000;  
}
```

Com o Nginx já configurado e rodando, que tal testar essa integração? Reinicie tanto Nginx, por meio do comando `nginx -s reload` , como também a nossa aplicação, via `npm start` . Se até agora nenhum problema aconteceu, basta acessar a aplicação em seu 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, como ler e acompanhar novas referências sobre a 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 — <https://nodejs.org>
- GitHub do Node.js — <https://github.com/nodejs/node>
- Documentação do Node.js — <https://nodejs.org/api>
- Blog do Node.js — <https://blog.nodejs.org>
- NPM Registry — <https://npmjs.org>
- Fórum do NodeBR —
<https://groups.google.com/forum/#!forum/nodebr>
- Underground WebDev (autor deste livro) —
<https://udgwebdev.com>
- Scoop.it do Node.js — <https://scoop.it/t/nodejs-code>
- EchoJS - <https://www.echojs.com>

- TJ Holowaychuk (autor do Express e Mocha) — <https://medium.com/@tjholowaychuk>
- NodeCloud — <https://nodecloud.org>
- NodeSecurity — <https://nodesecurity.io>
- Blog RisingStack - <https://blog.risingstack.com>
- Blog Caelum — <https://blog.caelum.com.br>
- NodeSchool — <https://nodeschool.io>

Eventos, screencasts, podcasts e cursos

- NodeSchool — <https://nodeschool.io>
 - EggHead.IO — <https://egghead.io/technologies/node>
 - Node Knockout (Hackaton Node.js inspirado em Rails Rumble) — <https://nodeknockout.com>
 - Codeschool: real time with Node.js — <https://codeschool.com/courses/real-time-web-with-nodejs>
 - Tuts+ Courses: building web apps with Node.js and Express — <https://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/. 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.