© 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 auto rização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

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

Casa do Código

**Agradecimentos**

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

Um agradecimento especial a minha namorada Natália Santos, afinal começa mos a namorar na mesma época que comecei este livro, e sua companhia, compre ensão e incentivo foram essenciais para persistir neste projeto.

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

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

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

Obrigado pessoal da editora Casa do Código, em especial ao Paulo Silveira e Adriano Almeida. Muito obrigado pelo suporte e pela oportunidade! Obrigado galera da comunidade NodeBR. Seus *feedbacks* ajudaram a melhorar este livro e também agradeço a todos os leitores do blog Underground WebDev. Afi nal a essência deste livro foi baseado nos posts sobre Node.js publicados lá. Por último, obrigado você, prezado leitor, por adquirir este livro. Espero que este livro seja uma ótima referência para ti.

i

Casa do Código

**Comentários**

Veja abaixo alguns comentários no blog Underground WebDev a respeito do con teúdo que você esta prestes a ler.

*Parabéns pelo Post! Adorei, muito explicativo. A comunidade brasileira agradece.* – Rafael Henrique Moreira - virtualjoker@gmail.com - “http://nodebr.com”

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

– Magno Ozzyr - magno\_ozzyr@hotmail.com

*Digno de reconhecimento o empenho do Caio no projeto de contribuir com o desenvolvimento e propagação dessa tecnologia. Isso combina com o estilo ambicioso e persistente que sempre demonstrou no processo de formação. Sucesso! Continue compartilhando os frutos do seu trabalho para assim deixar sua marca na história da computação.*

– Fernando Macedo - fernando@fmacedo.com.br - “http://fmacedo.com.br”

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

– Thiago Ferauche - thiago.ferauche@gmail.com

*Wow, muito bacana Caio! Eu mesmo estou ensaiando para aprender Javascript e cia. Hoje trabalho mais com HTML/CSS, e essa ideia de “para Leigos” me interessa muito! Fico no aguardo dos próximos posts!! =)*

– Marcio Toledo - mntoledo@gmail.com - “http://marciotoledo.com” iii

Casa do Código

*Caião, parabéns pela iniciativa, pelo trabalho e pela contribuição para a comunidade. Trabalhamos juntos e sei que você é uma pessoa extremamente dedicada e ansioso por novos conhecimentos. Continue assim e sucesso!*

– Leonardo Pinto - leonardo.pinto@gmail.com

*Caio, parabéns pelo curso e pelo conteúdo. É sempre bom contar com material de qualidade produzido no Brasil, pois precisamos difundir o uso de novas tecnologias e encorajar seu uso.*

– Evaldo Junior - evaldojuniorbento@gmail.com - “http://evaldojunior.com.br”

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

iv

Casa do Código

**Sobre o autor**

****

Figura 1: Caio Ribeiro Pereira

Sou *Web Developer* na *startup* BankFacil, minha experiência baseia-se no domínio dessa sopa de letrinhas: Node.js, Modular Javascript, Modular CSS, Ruby, Java, Mon goDB, Redis, Agile, Filosofia Lean, Scrum, XP, Kanban e TDD.

Bacharel em Sistemas de Informação pela Universidade Católica de Santos, blo gueiro nos tempos livres, apaixonado por programação, web, tecnologias, filmes e seriados.

Participante das comunidades:

• NodeBR: Comunidade Brasileira de Node.js

• MeteorBrasil: Comunidade Brasileira de Meteor

• DevInSantos: Grupo de Desenvolvedores de Software em Santos

Iniciei em 2011 como palestrante nos eventos DevInSantos e Exatec, abordando temas atuais sobre Node.js e Javascript.

Autor dos Blogs: Underground WebDev e Underground Linux. v

Casa do Código

**Prefácio**

**As mudanças do mundo web**

Tudo na web se trata de consumismo e produção de conteúdo. Ler ou escrever blogs, assistir ou enviar vídeos, ver ou publicar fotos, ouvir músicas e assim por diante. Isso fazemos naturalmente todos os dias na internet. E cada vez mais aumenta a neces sidade dessa interação entre os usuários com os diversos serviços da web. De fato, o mundo inteiro quer interagir mais e mais na internet, seja através de conversas com amigos em chats, jogando games online, atualizando constantemente suas redes so ciais ou participando de sistemas colaborativos. Esses tipos de aplicações requerem um poder de processamento extremamente veloz, para que seja eficaz a interação em tempo real entre cliente e servidor. E mais, isto precisa acontecer em uma escala massiva, suportando de centenas a milhões de usuários.

Então o que nós desenvolvedores precisamos fazer? Nós precisamos criar uma comunicação em tempo real entre cliente e servidor — que seja rápido, atenda muitos usuários ao mesmo tempo e utilize recursos de I/O (dispositivos de entrada ou saída) de forma eficiente. Qualquer pessoa com experiência desenvolvimento web sabe que o HTTP não foi projetado para suportar estes requisitos. E pior, infelizmente exis tem sistemas que os adotam de forma ineficiente e incorreta, implementando solu ções *workaround* (“Gambiarras”) que executam constantemente requisições assín cronas no servidor, mais conhecidas como *long-polling*. Para sistemas trabalharem em tempo real, servidores precisam enviar e receber dados utilizando comunica ção bidirecional, ao invés de utilizar intensamente requisição e resposta do modelo HTTP através do*Ajax*. E também temos que manter esse tipo comunicação de forma leve e rápida para manter escalável, reutilizável e fácil de manter o desenvolvimento a longo prazo.

**A quem se destina esse livro?**

Esse livro é destinado aos desenvolvedores web, que tenham pelo menos conhe cimentos básicos de Javascript e arquitetura cliente-servidor. Ter domínio desses

vii

Casa do Código

conceitos, mesmo que seja um conhecimento básico deles, será essencial para que a leitura desse 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 4 até o capítulo final, iremos desenvolver na prática um projeto web, utilizando os principais frameworks e aplicando as boas práticas de desenvolvimento Javascript para Node.js.

viii

Casa do Código Sumário

**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 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3 1.5 Instalação e configuração . . . . . . . . . . . . . . . . . . . . . . . . . . 4 1.6 Gerenciando módulos com NPM . . . . . . . . . . . . . . . . . . . . . 7 1.7 Entendendo o package.json . . . . . . . . . . . . . . . . . . . . . . . . . 8 1.8 Escopos de variáveis globais . . . . . . . . . . . . . . . . . . . . . . . . 10 1.9 CommonJS, Como ele funciona? . . . . . . . . . . . . . . . . . . . . . 10

**2 Desenvolvendo aplicações web 13** 2.1 Criando nossa primeira aplicação web . . . . . . . . . . . . . . . . . . 13 2.2 Como funciona um servidor http? . . . . . . . . . . . . . . . . . . . . . 14 2.3 Trabalhando com diversas rotas . . . . . . . . . . . . . . . . . . . . . . 15 2.4 Separando o HTML do Javascript . . . . . . . . . . . . . . . . . . . . . 18 2.5 Desafio: Implementar um roteador de url . . . . . . . . . . . . . . . . 19

**3 Por que o assíncrono? 21** 3.1 Desenvolvendo de forma assíncrona . . . . . . . . . . . . . . . . . . . 21 3.2 Assincronismo versus Sincronismo . . . . . . . . . . . . . . . . . . . . 24 3.3 Entendendo o Event-Loop . . . . . . . . . . . . . . . . . . . . . . . . . 26 3.4 Evitando Callbacks Hell . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

ix

Sumário Casa do Código

**4 Iniciando com o Express 31** 4.1 Por que utilizá-lo? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31 4.2 Instalação e configuração . . . . . . . . . . . . . . . . . . . . . . . . . . 32 4.3 Criando um projeto de verdade . . . . . . . . . . . . . . . . . . . . . . 33 4.4 Gerando scaffold do projeto . . . . . . . . . . . . . . . . . . . . . . . . 35 4.5 Organizando os diretórios do projeto . . . . . . . . . . . . . . . . . . . 38

**5 Dominando o Express 43** 5.1 Estruturando views . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43 5.2 Controlando as sessões de usuários . . . . . . . . . . . . . . . . . . . . 44 5.3 Criando rotas no padrão REST . . . . . . . . . . . . . . . . . . . . . . 49 5.4 Aplicando filtros antes de acessar as rotas . . . . . . . . . . . . . . . . 54 5.5 Indo além: criando páginas de erros amigáveis . . . . . . . . . . . . . 56

**6 Programando sistemas real-time 61** 6.1 Como funciona uma conexão bidirecional? . . . . . . . . . . . . . . . 61 6.2 Conhecendo o framework Socket.IO . . . . . . . . . . . . . . . . . . . 62 6.3 Implementando um chat real-time . . . . . . . . . . . . . . . . . . . . 63 6.4 Organizando o carregamento de Sockets . . . . . . . . . . . . . . . . . 69 6.5 Socket.IO e Express em uma mesma sessão . . . . . . . . . . . . . . . 70 6.6 Gerenciando salas do chat . . . . . . . . . . . . . . . . . . . . . . . . . 73 6.7 Notificadores na agenda de contatos . . . . . . . . . . . . . . . . . . . 77 6.8 Principais eventos do Socket.IO . . . . . . . . . . . . . . . . . . . . . . 80

**7 Integração com banco de dados 83** 7.1 Bancos de dados mais adaptados para Node.js . . . . . . . . . . . . . . 83 7.2 MongoDB no Node.js utilizando Mongoose . . . . . . . . . . . . . . . 85 7.3 Modelando com Mongoose . . . . . . . . . . . . . . . . . . . . . . . . . 86 7.4 Implementando um CRUD na agenda de contatos . . . . . . . . . . . 87 7.5 Persistindo estruturas de dados com NoSQL Redis . . . . . . . . . . . 91 7.6 Mantendo um histórico de conversas do chat . . . . . . . . . . . . . . 92

x

Casa do Código Sumário

**8 Preparando um ambiente de testes 97** 8.1 Mocha, o framework de testes para Node.js . . . . . . . . . . . . . . . 97 8.2 Criando um Environment para testes . . . . . . . . . . . . . . . . . . . 98 8.3 Instalando e configurando o Mocha . . . . . . . . . . . . . . . . . . . . 100 8.4 Rodando o Mocha no ambiente de testes . . . . . . . . . . . . . . . . . 101 8.5 Testando as rotas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102 8.6 Deixando seus testes mais limpos . . . . . . . . . . . . . . . . . . . . . 109

**9 Aplicação Node em produção 111** 9.1 O que vamos fazer? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111 9.2 Configurando Clusters . . . . . . . . . . . . . . . . . . . . . . . . . . . 111 9.3 Redis controlando as sessões da aplicação . . . . . . . . . . . . . . . . 114 9.4 Monitorando aplicação através de logs . . . . . . . . . . . . . . . . . . 116 9.5 Otimizações no Express . . . . . . . . . . . . . . . . . . . . . . . . . . . 118 9.6 Otimizando requisições do Socket.IO . . . . . . . . . . . . . . . . . . . 119 9.7 Aplicando Singleton nas conexões do Mongoose . . . . . . . . . . . . 122 9.8 Mantendo o sistema no ar com Forever . . . . . . . . . . . . . . . . . . 122 9.9 Integrando Nginx no Node.js . . . . . . . . . . . . . . . . . . . . . . . 124

**10 Continuando os estudos 129 Índice Remissivo 131**

xi

Capítulo 1

Bem-vindo ao mundo Node.js

**1.1 O problema das arquiteturas bloqueantes**

Os sistemas para web desenvolvidos sobre plataforma **.NET, Java, PHP, Ruby ou Python** 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 requisição feita pelo usuá rio. Com o decorrer da aplicação, novos usuários vão acessando-a, gerando uma requisição no servidor. Um sistema bloqueante enfileira cada requisição e depois as processa, uma a uma, não permitindo múltiplos processamentos delas. Enquanto uma requisição é processada as demais ficam em espera, mantendo por um período de tempo uma fila de requisições ociosas.

Esta é uma arquitetura clássica, existente em diversos sistemas pelo qual possui um design ineficiente. É gasto grande parte do tempo mantendo uma fila ociosa enquanto é executado um I/O. Tarefas como enviar e-mail, consultar o banco de dados, leitura em disco, são exemplos de tarefas que gastam uma grande fatia desse

1.2. E assim nasceu o Node.js Casa do Código

tempo, bloqueando o sistema inteiro enquanto não são finalizadas. Com o aumento de acessos no sistema, a frequência de gargalos serão mais frequentes, aumentando a necessidade de fazer um *upgrade* nos *hardwares* dos servidores. Mas *upgrade* das máquinas é algo muito custoso, o ideal seria buscar novas tecnologias que façam bom uso do *hardware* existente, que utilizem ao máximo o poder do processador atual, não o mantendo ocioso quando o mesmo realizar tarefas do tipo bloqueante.

**1.2 E assim nasceu o Node.js**

****Figura 1.1: Logotipo do Node.js.

Foi baseado neste problema que, no final de 2009, Ryan Dahl com a ajuda inicial de 14 colaboradores criou o Node.js. Esta tecnologia possui um modelo inovador, sua arquitetura é totalmente *non-blocking thread* (não-bloqueante), apresentando uma boa performance com consumo de memória e utilizando ao máximo e de forma eficiente o poder de processamento dos servidores, principalmente em sistemas que produzem uma alta carga de processamento. Usuários de sistemas Node estão livres de aguardarem por muito tempo o resultado de seus processos, e principalmente não sofrerão de *dead-locks* no sistema, porque nada bloqueia em sua plataforma e desenvolver sistemas nesse paradigma é simples e prático.

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

2

Casa do Código Capítulo 1. Bem-vindo ao mundo Node.js

**1.3 Single-thread**

Suas aplicações serão *single-thread*, ou seja, cada aplicação terá instância de um único processo. Se você esta 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, como por exemplo, utilizando *clusters* (assunto a ser explicado no capítulo 9.2), que é um módulo nativo do Node.js e é super fácil de implementá-lo. Outra maneira é utilizar ao máximo a programação assíncrona. Esse será o assunto mais abordado durante o decorrer deste livro, pelo qual explicarei diversos cenários e exemplos práticos em que são executados em paralelo funções em *background* que aguardam o seu retorno através de funções de *callback* e tudo isso é trabalhado forma não-bloqueante.

**1.4 Event-Loop**

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

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

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

3

1.5. Instalação e configuração Casa do Código

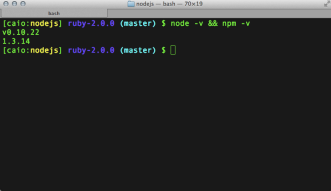
**1.5 Instalação e configuração**

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

Figura 1.2: Página de Download do Node.js.

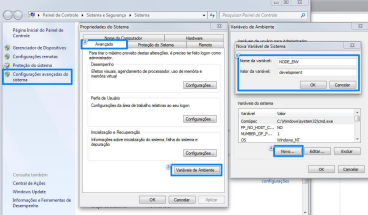
**Instalando Node.js:** Primeiro passo, acesse o site oficial: (http://nodejs.org) e clique em **Download**, para usuários do *Windows e MacOSX*, basta baixar os seus instaladores e executá-los normalmente. Para quem já utiliza *Linux* com *Package Manager* instalado, acesse esse link (https://github.com/joyent/node/wiki/ Installing-Node.js-via-package-manager) que é referente as instruções sobre como instalá-lo em diferentes sistemas. Instale o Node.js de acordo com seu sistema, caso não ocorra problemas, basta abrir o seu terminal console ou prompt de comando e digitar o comando: node -v && npm -v para ver as respectivas versões do Node.js e NPM *(Node Package Manager)* que foram instaladas.

4

Casa do Código Capítulo 1. Bem-vindo ao mundo Node.js Figura 1.3: Versão do Node.js e NPM utilizada neste livro.

A última versão estável utilizada neste livro é **Node 0.10.22** e **NPM 1.3.14**. Alias é altamente recomendável utilizar esta versão ou superior, pois recen temente foi identificado uma vulnerabilidade de ataque *DoS* em versões an teriores, veja mais detalhes no blog oficial: (http://blog.nodejs.org/2013/10/22/ cve-2013-4450-http-server-pipeline-flood-dos) Dica: Todo conteúdo deste livro será compatível com versões do Node.js **0.8.0** ou superiores. **Configurando ambi ente de desenvolvimento:** Para configurá-lo basta adicionar uma variável de am biente **NODE\_ENV** no sistema operacional. Em sistemas *Linux ou OSX*, basta acessar com um editor de texto qualquer e em modo super user **(sudo)** o ar quivo .bash\_profile ou .bashrc e adicionar o seguinte comando: export NODE\_ENV=’development’. No Windows 7, o processo é um pouco diferente.

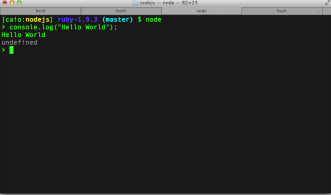
5

1.5. Instalação e configuração Casa do Código Figura 1.4: Configurando a variável NODE\_ENV no Windows 7.

Clique com botão direito no ícone **Meu Computador** e selecione a opção **Pro priedades**, no lado esquerdo da janela, clique no link **Configurações avançadas do sistema**. Na janela seguinte, acesse a aba **Avançado** e clique no botão **Variáveis de Ambiente...**, agora no campo **Variáveis do sistema** clique no botão **Novo...**, em nome da variável digite NODE\_ENV e em valor da variável digite: development. Após finalizar essa tarefa reinicie seu computador para carregar essa variável no sis tema 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.

6

Casa do Código Capítulo 1. Bem-vindo ao mundo Node.js Figura 1.5: Hello World via REPL do Node.js

**1.6 Gerenciando módulos com NPM**

Assim como o *Gems* do *Ruby* ou o *Maven* do *Java*, o Node.js também possui o seu próprio gerenciador de pacotes, ele se chama NPM *(Node Package Manager)*. Ele se tornou tão popular pela comunidade, que foi a partir da versão **0.6.0** do Node.js que ele se integrou no instalador do Node.js, tornando-se o gerenciador *default*. Isto simplificou a vida dos desenvolvedores na época, pois fez com que diversos projetos se convergissem para esta plataforma. Não listarei todos, mas apenas os comandos principais para que você tenha noções de como gerenciar módulos nele:

• npm install nome\_do\_módulo: instala um módulo no projeto. • npm install -g nome\_do\_módulo: instala um módulo global.

• npm install nome\_do\_módulo --save: instala o módulo no projeto, atualizando o package.json na lista de dependências.

• npm list: lista todos os módulos do projeto.

• npm list -g: lista todos os módulos globais.

• npm remove nome\_do\_módulo: desinstala um módulo do projeto. 7

1.7. Entendendo o package.json Casa do Código

• npm remove -g nome\_do\_módulo: desinstala um módulo global. • npm update nome\_do\_módulo: atualiza a versão do módulo. • npm update -g nome\_do\_módulo: atualiza a versão do módulo global. • npm -v: exibe a versão atual do npm.

• npm adduser nome\_do\_usuário: cria uma conta no npm, através do site (https://npmjs.org) .

• npm whoami: exibe detalhes do seu perfil público npm (é necessário criar uma conta antes).

• npm publish: publica um módulo no site do npm, é necessário ter uma conta antes.

**1.7 Entendendo o package.json**

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

Este arquivo é essencial para um projeto Node.js. Um package.json mal es crito 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 abaixo apresentarei um package.json que contém os principais atributos para descrever um módulo:

{

"name": "meu-primero-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"

8

Casa do Código Capítulo 1. Bem-vindo ao mundo Node.js

},

"devDependencies": {

"modulo-4": "\*"

}

}

Com esses atributos, você já descreve o mínimo possível o que será sua aplicação. O atributo name é o principal, com ele você descreve o nome do projeto, nome pelo qual seu módulo será chamado via função require('meu-primeiro-node-app'). Em description descrevemos o que será este módulo. Ele deve ser escrito de forma curta e clara explicando um resumo do módulo. O author é um atributo para informar o nome e email do autor, utilize o formato: Nome <email> para que sites como (https://npmjs.org) reconheça corretamente esses dados. Outro atributo principal é o version, é com ele que definimos a versão atual do módulo, é extremamente recomendado que te nha este atributo, senão 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 exem plo, a versão 1.2.3 esta dividida nos níveis: *Major* (1), *Minor* (2) e *Patch* (3). Repare que no campo dependencies foram incluídos 4 módulos, cada módulo utilizou uma forma diferente de definir a versão que será adicionada no projeto. O primeiro, o modulo-1 somente será incluído sua versão fixa, a 1.0.0. Utilize este tipo ver são para instalar dependências cuja suas atualizações possam quebrar o projeto pelo simples fato de que certas funcionalidades foram removidas e ainda as utilizamos na aplicação. O segundo módulo já possui uma certa flexibilidade de update. Ele utiliza o caractere **~** que faz atualizações a nível de *patch* (1.0.x), geralmente essas atua lizações são seguras, trazendo apenas melhorias ou correções de bugs. O modulo-3 atualiza versões que seja maior ou igual a 1.0.0 em todos os níveis de versão. Em muitos casos utilizar **”>=”** pode ser perigoso, por que a dependência pode ser atua lizada a nível *major* ou *minor*, contendo grandes modificações que podem quebrar um sistema em produção, comprometendo seu funcionamento e exigindo que você atualize todo código até voltar ao normal. O último, o modulo-4, utiliza o caractere **"\*”**, este sempre pegará a última versão do módulo em qualquer nível. Ele também pode causar problemas nas atualizações e tem o mesmo comportamento do versio namento do modulo-3. Geralmente ele é utilizado em devDependencies, que são dependências focadas para testes automatizados, e as atualizações dos módulos

9

1.8. Escopos de variáveis globais Casa do Código 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 utiliza **escopos locais e globais** de variáveis. A única diferença é como são imple mentados 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 utilizamos uma outra *keyword* para aplicar essa mesma técnica:

global.hoje = new Date();

console.log(global.hoje);

Ao utilizar global mantemos uma variável global, acessível em qualquer parte do projeto sem a necessidade de chamá-la via require ou passá-la por parâmetro em uma função. Esse conceito de variável global é existente na maioria das lingua gens de programação, assim como sua utilização, pelo qual 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 será utilizada com frequência em um projeto Node.js. A função require('nome-do-modulo') é um exem plo disso, ela carrega um módulo. E para criar um código Javascript que seja mo dular e carregável pelo require, utilizam-se as variáveis globais: exports ou module.exports. Abaixo apresento-lhe dois exemplos de códigos que utilizam esse padrão do *CommonJS*, primeiro crie o código hello.js:

module.exports = function(msg) {

console.log(msg);

};

10

Casa do Código Capítulo 1. Bem-vindo ao mundo Node.js

E também crie o código human.js com o seguinte código:

exports.hello = function(msg) {

console.log(msg);

};

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

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

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

hello('Olá pessoal!');

human.hello('Olá galera!');

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

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

Percebam o quão 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*.

11

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 protoco los: *HTTP, HTTPS, FTP, SSH, DNS, TCP, UDP, WebSockets* e também existem outros protocolos, que são disponíveis através de módulos não-oficiais criados pela comu nidade. Um dos mais utilizados para desenvolver sistemas web é o protocolo *HTTP*. De fato, é o protocolo com a maior quantidade de módulos disponíveis para trabalhar no Node.js. Na prática desenvolveremos um sistema web utilizando o módulo nativo HTTP, mostrando suas vantagens e desvantagens. Também apresentarei soluções de módulos estruturados para desenvolver aplicações complexas de forma modular e escalável.

Toda aplicação web necessita de um servidor para disponibilizar todos os seus recursos. Na prática, com o Node.js você desenvolve uma *"aplicação middleware”*, ou seja, além de programar as funcionalidades da sua aplicação, você também programa códigos de configuração de infraestrutura da sua aplicação. Inicialmente isso parece ser muito trabalhoso, pois o Node.js utiliza o mínimo de configurações para servir

2.2. Como funciona um servidor http? Casa do Código

uma aplicação, mas esse trabalho permite que você customize ao máximo o seu ser vidor. Uma vantagem disso é poder configurar em detalhes o sistema, permitindo desenvolver algo performático e controlado pelo programador.

Caso performance não seja prioridade no desenvolvimento do seu sistema, reco mendo que utilize alguns módulos adicionais que já vêm com o mínimo necessário de configurações prontas para você não perder tempo trabalhando com isso. Alguns módulos conhecidos são: **Connect (https://github.com/senchalabs/connect) , Ex press (http://expressjs.com) , Geddy (http://geddyjs.org) , CompoundJS (http: //compoundjs.com) , Sails (http://balderdashy.github.io/sails)** . Esses módulos já são preparados para trabalhar desde uma infraestrutura mínima até uma mais en xuta, permitindo trabalhar desde arquiteturas *RESTFul*, padrão MVC *(Model-View Controller)* e também com conexões real-time utilizando *WebSockets*.

Primeiro usaremos apenas o módulo nativo *HTTP*, pois precisamos entender todo o conceito desse módulo, visto que todos os frameworks citados acima o uti lizam como estrutura inicial em seus projetos. Abaixo mostro a vocês uma clássica aplicação *Hello World*. Crie o arquivo hello\_server.js com o seguinte con teúdo:

var http = require('http');

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

response.end();

});

server.listen(3000);

Esse é um exemplo clássico e simples de um servidor node.js. Ele está sendo exe cutado na **porta 3000**, por padrão ele responde através da **rota raiz “/”** um resultado em formato html com a mensagem: Hello World!.

Vá para a linha de comando e rode node hello\_server.js. Faça o teste acessando, no seu navegador, o endereço http://localhost:3000 .

**2.2 Como funciona um servidor http?**

Um servidor node.js utiliza o mecanismo *Event loop*, sendo responsável por lidar com a emissão de eventos. Na prática, a função http.createServer() é respon sável por levantar um servidor e o seu callback function(request, response)

14

Casa do Código Capítulo 2. Desenvolvendo aplicações web

apenas é executado quando o servidor recebe uma requisição. Para isso, o *Event loop* constantemente verifica se o servidor foi requisitado e, quando ele recebe uma re quisição, ele emite um evento para que seja executado o seu callback.

O Node.js trabalha muito com chamadas assíncronas que respondem através callbacks do javascript. Por exemplo, se quisermos notificar que o servidor está de pé, mudamos a linha server.listen para receber em parâmetro uma função que faz esse aviso:

server.listen(3000, function(){

console.log('Servidor Hello World rodando!');

});

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

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

var http = require('http');

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

response.end();

}

var server = http.createServer(atendeRequisicao);

var servidorLigou = function() {

console.log('Servidor Hello World rodando!');

}

server.listen(3000, servidorLigou);

**2.3 Trabalhando com diversas rotas**

Até agora respondemos apenas o endereço /, mas queremos possibilitar que nosso servidor também responda a outros endereços. Utilizando um palavreado comum entre desenvolvedores rails, queremos adicionar novas **rotas**.

15

2.3. Trabalhando com diversas rotas Casa do Código

Vamos adicionar duas novas rotas, uma rota /bemvindo para página de “Bem vindo ao Node.js!” e uma rota genérica, que leva para uma página de erro. Fare mos isso através de um simples encadeamento de condições, em um novo arquivo: hello\_server3.js:

var http = require('http');

var server = http.createServer(function(request, response){ response.writeHead(200, {"Content-Type": "text/html"}); if(request.url == "/"){

response.write("<h1>Página principal</h1>");

}else if(request.url == "/bemvindo"){

response.write("<h1>Bem-vindo :)</h1>");

}else{

response.write("<h1>Página não encontrada :(</h1>");

}

response.end();

});

server.listen(3000, function(){

console.log('Servidor rodando!');

});

Rode novamente e faça o teste acessando a url http://localhost:3000/bemvindo , e também acessando uma outra, diferente desta. Viu o resultado? Reparem na complexidade do nosso código: o roteamento foi tratado através dos comandos**if e else**, e a leitura de url é obtida através da função request.url() que retorna uma string sobre o que foi digitado na barra de endereço do browser. Esses endereços utilizam padrões para capturar valores na url. Esses padrões são: *query strings* ( ?nome=joao) e *path* ( /admin). Em um projeto maior, tratar todas as urls dessa maneira seria trabalhoso e confuso demais. No Node.js, existe o módulo nativo chamado url, que é responsável por fazer *parser* e formatação de urls. Acompanhe como capturamos valores de uma query string no exemplo abaixo. Aproveite e crie o novo arquivo url\_server.js:

var http = require('http');

var url = require('url');

var server = http.createServer(function(request, response){ response.writeHead(200, {"Content-Type": "text/html"}); response.write("<h1>Dados da query string</h1>");

var result = url.parse(request.url, true);

16

Casa do Código Capítulo 2. Desenvolvendo aplicações web

for(var key in result.query){

response.write("<h2>"+key+" : "+result.query[key]+"</h2>"); }

response.end();

});

server.listen(3000, function(){

console.log('Servidor http.');

});

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

Esse módulo identifica através do retorno da função **url.parser()** os seguintes atributos:

• **href:** Retorna a url completa: ‘http://user:pass@host.com:8080/p/a/t/h? query=string#hash’

• **protocol:** Retorna o protocolo: ‘http’

• **host:** Retorna o domínio com a porta: ‘host.com:8080’

• **auth:** Retorna dados de autenticação: ‘user:pass’

• **hostname:** Retorna o domínio: ‘host.com’

• **port:** Retorna a porta: ‘8080’

• **pathname:** Retorna os pathnames da url: ‘/p/a/t/h’

• **search:** Retorna uma query string: ‘?query=string’

• **path:** Retorna a concatenação de pathname com query string: ‘/p/a/t/h?query=string’

• **query:** Retorna uma query string em JSON: {‘query’:’string’}

• **hash:** Retorna ancora da url: ‘#hash’

Resumindo, o módulo url permite organizar todas as urls da aplicação. 17

2.4. Separando o HTML do Javascript Casa do Código

**2.4 Separando o HTML do Javascript**

Agora precisamos organizar os códigos HTML, e uma boa prática é separá-los do Javascript, fazendo com que a aplicação renderize código HTML quando o usuário solicitar uma determinada rota. Para isso, utilizaremos outro módulo nativo **FS** *(File System)*. Ele é responsável por manipular arquivos e diretórios do sistema operaci onal. O mais interessante desse módulo é que ele possui diversas funções de mani pulação tanto de forma assíncrona como de forma síncrona. Por padrão, as funções nomeadas com o final Sync() são para tratamento síncrono. No exemplo abaixo, apresento as duas maneiras de ler um arquivo utilizando *File System*:

var fs = require('fs');

fs.readFile('/index.html', function(erro, arquivo){

if (erro) throw erro;

console.log(arquivo);

});

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

console.log(arquivo);

Diversos módulos do Node.js possuem funções com versões assíncronas e sín cronas. O fs.readFile() faz uma leitura assíncrona do arquivo index.html. Depois que o arquivo foi carregado, é invocado uma função callback para fazer os tra tamentos 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 re tornar o arquivo.

| **Limitações do File System nos sistemas operacionais**  Um detalhe importante sobre o módulo *File System* é que ele não é 100% consistente entre os sistemas operacionais. Algumas funções são específicas para sistemas *Linux, OS X, Unix* e outras são apenas para *Win dows*.  Para melhores informações leia sua documentação: http://nodejs. org/api/fs.html |
| --- |

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

18

Casa do Código Capítulo 2. Desenvolvendo aplicações web

var http = require('http');

var fs = require('fs');

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

response.writeHeader(200, {'Content-Type': 'text/html'}); response.write(html);

response.end();

});

});

server.listen(3000, function(){

console.log('Executando Site Pessoal');

});

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

*<!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: Implementar um roteador de url**

Antes de finalizar esse capítulo, quero propor um desafio. Já que aprendemos a utili zar 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;

19

2.5. Desafio: Implementar um roteador de url Casa do Código

• Ao digitar no browser o path: **/artigos** deve renderizar **artigos.html**; • A regra anterior também se aplica para o arquivo **contato.html**;

• Ao digitar qualquer path diferente de **/artigos** e **/contato** deve renderizar **erro.html**;

• A leitura dos arquivos html deve ser assíncrona;

• A rota principal **"/”** deve renderizar **artigos.html**;

Algumas dicas importantes:

1) Utilize o retorno da função: url.parse() para capturar o **pathname** digitado e renderizar o html correspondente. Se o **pathname** estiver vazio significa que deve renderizar a página de artigos, e se estiver com um valor diferente do nome dos arquivos html, renderize a página de erros.

2) Você também pode inserir conteúdo html na função: response.end(html), economizando linha de código ao não utilizar a função: response.write(html).

3) Utilize a função: fs.exists(html) para verificar se existe o html com o mesmo nome do pathname digitado.

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

20

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 quando elas são invocadas. O código abaixo exem plifica as diferenças entre uma função síncrona e assíncrona em relação a linha do tempo que ela são executadas. Basicamente criaremos um loop de 5 iterações, a cada iteração será criado um arquivo texto com o mesmo conteúdo Hello Node.js!. Primeiro vamos começar com o código síncrono. Crie o arquivo text\_sync.js com o código abaixo:

var fs = require('fs');

for(var i = 1; i <= 5; i++) {

var file = "sync-txt" + i + ".txt";

var out = fs.writeFileSync(file, "Hello Node.js!");

console.log(out);

}

3.1. Desenvolvendo de forma assíncrona Casa do Código

Agora vamos criar o arquivo text\_async.js, com seu respectivo código, di ferente apenas na forma de chamar a função writeFileSync, que será a versão assíncrona writeFile, recebendo uma função como argumento:

var fs = require('fs');

for(var i = 1; i <= 5; i++) {

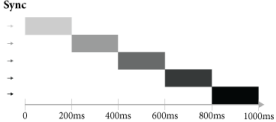
var file = "async-txt" + i + ".txt";

fs.writeFile(file, "Hello Node.js!", function(err, out) { console.log(out);

});

}

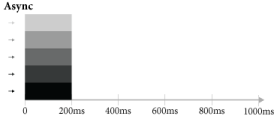
Vamos rodar? Execute os comandos: node text\_sync e depois node text\_async. Se for gerado 10 arquivos no mesmo diretório do código-fonte, en tão deu tudo certo. Mas a execução de ambos foi tão rápida que não foi perceptível ver as diferenças entre o text\_async e o text\_sync. Para entender melhor as diferenças, veja as *timelines* que foram geradas. O text\_sync por ser um código síncrono, invocou chamadas de I/O bloqueante, gerando o gráfico abaixo:

Figura 3.1: Timeline síncrona bloqueante.

Repare no tempo de execução, o text\_sync demorou 1000 milissegundos, 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:

22

Casa do Código Capítulo 3. Por que o assíncrono? Figura 3.2: Timeline assíncrona não-bloqueante.

Isto fez com que o tempo de execução levasse 200 milissegundos, afinal foram invocados 5 vezes em paralelo a função fs.writeFile(), maximizando proces samento e minimizando o tempo de execução.

| **Threads vs Assincronismos**  Por mais que as funções assíncronas possam executar em paralelo vá rias tarefas, elas jamais serão consideradas umaThread (por exemploTh reads do Java). A diferença é que Threads são manipuláveis pelo desen volvedor, ou seja, você pode pausar a execução de uma Thread ou fazê-la esperar o término de uma outra. Chamadas assíncronas apenas invocam suas funções numa ordem de que você não tem controle, e você só sabe quando uma chamada terminou quando seu callback é executado.  Pode parecer vantajoso ter o controle sobre as Threads a favor de um sistema que executa tarefas em paralelo, mas pouco domínio sobre eles pode transformar seu sistema em um caos de travamentos *dead-locks*, afi nalThreads são executadas de forma bloqueante. Este é o grande diferen cial das chamadas assíncronas, elas executam em paralelo suas funções sem travar processamento das outras e principalmente sem bloquear o sistema principal. |
| --- |

É fundamental que o seu código Node.js invoque o mínimo possível de fun ções bloqueantes. Toda função síncrona impedirá, naquele instante, que o Node.js continue executando os demais códigos até que aquela função seja finalizada. Por

23

3.2. Assincronismo versus Sincronismo Casa do Código

exemplo, se essa função fizer um *I/O* em disco, ele vai bloquear o sistema inteiro, deixando o processador ocioso enquanto ele utiliza outros recursos de hardware, como por exemplo leitura em disco, utilização da rede etc.

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

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

**3.2 Assincronismo versus Sincronismo**

Para exemplificar melhor, os códigos abaixo representam um benchmark compa rando o tempo de bloqueio de execução **assíncrona vs síncrona**. Para isso crie 3 ar quivos: processamento.js, leitura\_async.js e leitura\_sync.js. Cri aremos isoladamente o código leitura\_async.js que faz leitura assíncrona:

var fs = require('fs');

var leituraAsync = function(arquivo){

console.log("Fazendo leitura assíncrona");

var inicio = new Date().getTime();

fs.readFile(arquivo)

var fim = new Date().getTime();

console.log("Bloqueio assíncrono: "+(fim - inicio)+ "ms"); };

module.exports = leituraAsync;

Em seguida criaremos o código leitura\_sync.js que faz leitura síncrona:

var fs = require('fs');

var leituraSync = function(arquivo){

console.log("Fazendo leitura síncrona");

var inicio = new Date().getTime();

fs.readFileSync(arquivo);

var fim = new Date().getTime();

console.log("Bloqueio síncrono: "+(fim - inicio)+ "ms"); };

module.exports = leituraSync;

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

24

Casa do Código Capítulo 3. Por que o assíncrono?

var http = require('http');

var fs = require('fs');

var leituraAsync = require('./leitura\_async');

var leituraSync = require('./leitura\_sync');

var arquivo = "./node.exe";

var stream = fs.createWriteStream(arquivo);

var download = "http://nodejs.org/dist/latest/node.exe"; http.get(download, function(res) {

console.log("Fazendo download do Node.js");

res.on('data', function(data){

stream.write(data);

});

res.on('end', function(){

stream.end();

console.log("Download finalizado!");

leituraAsync(arquivo);

leituraSync(arquivo);

});

});

Rode o comando node processamento.js para executar o benchmark. E agora, ficou clara a diferença entre o modelo bloqueante e o não-bloqueante? Parece que o método readFile executou muito rápido, mas não quer dizer que o arquivo foi lido. Ele recebe um último parâmetro, que é um callback indicando quando o arquivo foi lido, que não passamos na invocação que fizemos. Ao usar o fs.readFileSync(), bastaria fazer var conteudo = fs.readFileSync(). Mas qual é o problema dessa abordagem? **Ela segura todo o mecanismo do Node.JS**!

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

Abaixo apresento o resultado do benchmark realizado em minha máquina:

• Modelo: MacBook Air 2011

• Processador: Core i5

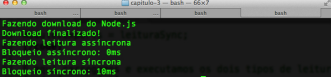
• Memória: 4GB RAM

25

3.3. Entendendo o Event-Loop Casa do Código

• Disco: 128GB SSD

Veja a pequena, porém significante diferença de tempo entre as duas funções de leitura.

Figura 3.3: Benchmark de leitura Async vs Sync.

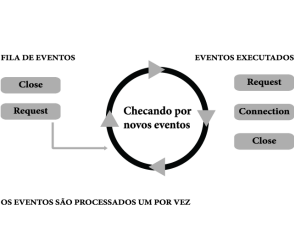
Se esse teste foi com um arquivo de mais ou menos 50 MB, imagine esse teste em larga escala, lendo múltiplos arquivos de 1 GB ao mesmo tempo ou realizando múltiplos uploads em seu servidor? Esse é um dos pontos fortes do Node.js!

**3.3 Entendendo o Event-Loop**

Realmente trabalhar de forma assíncrona tem ótimos benefícios em relação a proces samento I/O. Isso acontece devido ao fato, de que uma chamada de I/O é considerada um tarefa muito custosa para um computador realizar. Tão custosa que chega a ser perceptível para um usuário, por exemplo, quando ele tenta abrir um arquivo de 1 GB e o sistema operacional trava alguns segundos para abri-lo. Vendo o contexto de um servidor, por mais potente que seja seu *hardware*, eles terão os mesmos bloqueios perceptíveis pelo usuário, a diferença é que um servidor estará lidando com milhares usuários requisitando I/O, com a grande probabilidade de ser ao mesmo tempo.

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

26

Casa do Código Capítulo 3. Por que o assíncrono? Figura 3.4: Event-Loop do Node.js.

Basicamente ele é um loop infinito, que em cada iteração verifica se existem no vos eventos em sua **fila de eventos**. Tais eventos somente aparecem nesta fila quando são emitidos durante suas interações na aplicação. O EventEmitter, é o módulo responsável por emitir eventos e a maioria das bibliotecas do Node.js herdam deste módulo suas funcionalidades de eventos. Quando um determinado código emite um evento, o mesmo 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 através de uma função de escuta, semanticamente conhecida pelo nome: on().

Programar orientado a eventos vai manter sua aplicação mais robusta e es truturada para lidar com eventos que são executados de forma assíncrona não bloqueantes. Para conhecer mais sobre as funcionalidades do EventEmitter acesse sua documentação:

http://nodejs.org/api/events.html

27

3.4. Evitando Callbacks Hell Casa do Código

**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 as síncronas, que serão encadeadas uma na outra através das suas funções callback. No código a seguir apresentarei um exemplo desse caso. Crie um arquivo chamado callback\_hell.js, implemente e execute o código abaixo:

var fs = require('fs');

fs.readdir(\_\_dirname, function(erro, contents) {

if (erro) { throw erro; }

contents.forEach(function(content) {

var path = './' + content;

fs.stat(path, function(erro, stat) {

if (erro) { throw erro; }

if (stat.isFile()) {

console.log('%s %d bytes', content, stat.size);

}

});

});

});

Reparem na quantidade de callbacks encadeados que existem em nosso código. Detalhe: ele apenas faz uma simples leitura dos arquivos de seu diretório e imprime na tela seu nome e tamanho em bytes. Um pequena tarefa como essa deveria ter menos encadeamentos, concorda? Agora, imagine como seria a organização disso para realizar tarefas mas complexas? Praticamente o seu código seria um caos e total mente difícil de fazer manutenções. Por ser assíncrono, você perde o controle do que está executando em troca de ganhos com performance, porém, um detalhe impor tante sobre assincronismo é que na maioria dos casos os callbacks bem elaborados possuem como parâmetro uma variável de erro. Verifique nas documentações sobre sua existência e sempre faça o tratamento deles na execução do seu callback: if (erro) { throw erro; }, isso vai impedir a continuação da execução aleatória quando for identificado um erro.

Uma boa prática de código Javascript é criar funções que expressem seu ob jetivo e de forma isoladas, salvando em variável e passando-as como callback. Ao invés de criar funções anônimas, por exemplo, crie um arquivo chamado callback\_heaven.js com o código abaixo:

28

Casa do Código Capítulo 3. Por que o assíncrono?

var fs = require('fs');

var lerDiretorio = function() {

fs.readdir(\_\_dirname, function(erro, diretorio) {

if (erro) return erro;

diretorio.forEach(function(arquivo) {

ler(arquivo);

});

});

};

var ler = function(arquivo) {

var path = './' + arquivo;

fs.stat(path, function(erro, stat) {

if (erro) return erro;

if (stat.isFile()) {

console.log('%s %d bytes', arquivo, stat.size);

}

});

};

lerDiretorio();

Veja o quanto melhorou a legibilidade do seu código. Dessa forma deixamos mais semântico e legível o nome das funções e diminuímos o número de encade amentos 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.

29

Capítulo 4

Iniciando com o Express

**4.1 Por que utilizá-lo?**

Programar utilizando apenas a API HTTP nativa é muito trabalhoso! Conforme surgem necessidades de implementar novas funcionalidades, códigos gigantescos se riam acrescentados, aumentando a complexidade do projeto e dificultando futuras manutenções.

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

4.2. Instalação e configuração Casa do Código Figura 4.1: Framework Express.

Ele possui as seguintes características:

• MVR *(Model-View-Routes)*;

• MVC *(Model-View-Controller)*;

• Roteamento de urls via callbacks;

• Middleware;

• Interface *RESTFul*;

• Suporte a *File Uploads*;

• Configuração baseado em variáveis de ambiente;

• Suporte a *helpers* dinâmicos;

• Integração com *Template Engines*;

• Integração com SQL e NoSQL;

**4.2 Instalação e configuração**

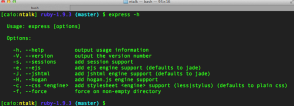
Sua instalação é muito simples e há algumas opções de configurações para começar um projeto. Para aproveitar todos os seus recursos, recomendo que instale-o em modo global:

npm install -g express

32

Casa do Código Capítulo 4. Iniciando com o Express

Feito isso, será necessário fechar e abrir seu terminal para habilitar o comando express, que é um *CLI (Command Line Interface)* do framework. Ele permite gerar um projeto inicial com suporte a sessões, Template engine (por padrão ele inclui o framework Jade) e um CSS engine (por padrão ele utiliza CSS puro). Para visualizar todas as opções execute o comando: express -h

Figura 4.2: Opções do Express em modo CLI.

**4.3 Criando um projeto de verdade**

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

Os requisitos do projeto são:

• O usuário deve criar, editar ou excluir um contato;

• O usuário deve se logar informando seu nome e e-mail;

• O usuário deve conectar ou desconectar no chat;

• O usuário deve enviar e receber mensagens no chat somente entre os contatos online;

O nome do projeto será Ntalk *(Node talk)* e utilizaremos as seguintes tecnologias: • **Node.js**: Backend do projeto;

• **MongoDB**: Banco de dados NoSQL orientado a documentos; 33

4.3. Criando um projeto de verdade Casa do Código

• **Redis**: Banco de dados NoSQL para estruturas de chave-valor; • **Express**: Framework para aplicações web;

• **Socket.IO**: Módulo para comunicação real-time;

• **MongooseJS**: ODM *(Object Data Mapper)* MongoDB para Node.js; • **Node Redis**: Cliente Redis para Node.js;

• **EJS**: *Template engine* para implementação de html dinâmico; • **Mocha**: Framework para testes automatizados;

• **SuperTest**: Módulo para emular requisições que será utilizado no teste de in tegração;

• **Nginx**: Servidor Web de alta performance para arquivos estáticos;

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

Caso você esteja com pressa de ver este projeto rodando, você pode cloná lo através do meu repositório público: (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

E depois acesse no seu navegador favorito o endereço:

http://localhost:3000

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

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

express ntalk --ejs

cd ntalk

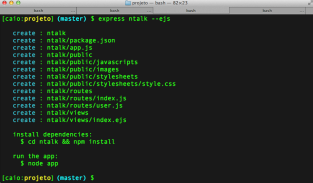
npm install

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

34

Casa do Código Capítulo 4. Iniciando com o Express

**4.4 Gerando scaffold do projeto**

Ao acessar o diretório do projeto, veja como foi gerado o seu *scaffold*: Figura 4.3: Estrutura do Express.

• package.json: contém as principais informações sobre a aplicação como: nome, autor, versão, colaboradores, url, dependências e muito mais.

• public: pasta pública que armazena conteúdo estático, por exemplo: ima gens, css, javascript etc.

• app.js: arquivo que inicializa o servidor do projeto, através do comando: node app.js.

• routes: diretório que mantém todas as rotas da aplicação.

• views: diretório que contém todas as *views* que são renderizadas pelas rotas.

Ao rodarmos o comando npm install, por padrão ele instalou as dependên cias existentes no package.json. Neste caso, ele apenas instalou o Express e o EJS (Embedded Javascript).

Vamos fazer algumas alterações nos códigos gerados pelo *scaffold* do Express. O primeiro passo será criar uma descrição sobre o nosso projeto e definir false o atributo private. Isso tudo será modificado no arquivo package.json, veja o código abaixo como ficou:

35

4.4. Gerando scaffold do projeto Casa do Código

{

"name": "ntalk",

"description": "Node talk - Agenda de contatos",

"private": false,

"version": "0.0.1",

"scripts": {

"start": "node app.js"

},

"dependencies": {

"express": "3.4.7",

"ejs": "0.8.5"

}

}

Também modificaremos o app.js, deixando-o com o mínimo de código pos sível para explicarmos em *baby-steps* o que realmente será necessário no desenvolvi mento deste projeto. Recomendo que apague o código gerado pelo *scaffold* e coloque o código abaixo:

var express = require('express')

, routes = require('./routes');

var app = express();

app.set('views', \_\_dirname + '/views');

app.set('view engine', 'ejs');

app.use(express.static(\_\_dirname + '/public'));

app.get('/', routes.index);

app.get('/usuarios', routes.user.index);

app.listen(3000, function(){

console.log("Ntalk no ar.");

});

Essa versão inicialmente atende os requisitos mínimos de uma aplicação Express. A brincadeira começa quando executamos a função express(), pois o seu retorno habilita todas as suas funcionalidades de seu framework, pelo qual armazenamos na variável app.

Com app.listen() fazemos algo parecido com o http.listen(), ou seja, ele é um *alias* responsável por colocar a aplicação no ar.

36

Casa do Código Capítulo 4. Iniciando com o Express

Os métodos app.get(), app.post(), app.put() e app.del() são funções de roteamento, cada uma delas associa seus respectivos métodos do pro tocolo HTTP *(GET, POST, PUT e DELETE)*. Seu primeiro parâmetro é uma string referente a uma rota da aplicação e o segundo é uma função callback que con tém uma requisição e uma resposta. Exemplo: app.get('/contatos', function(request, response));

O app.set(chave, valor) é uma estrutura de chave e valor man tida dentro da variável app. Seria o mesmo que criar no javascript o có digo: app["chave"] = "valor";. Um exemplo prático são as configura ções de views que foram definidos no código anterior: (app.set('views', '/views')) e (app.set('view engine', 'ejs')).

A maioria das funções chamadas diretamente pela variável express são her dadas de seus submódulos Connect e HTTP.

| **Detalhes sobre Connect**  O Connect (https://github.com/senchalabs/connect) é um mid dleware para servidores HTTP. Com ele é possível configurar aspectos do servidor através do conceito de pilha, ou seja, os primeiros itens in seridos são os primeiros a serem executados antes de chegar nas funções callbacks das rotas. O Express herdou todas as funcionalidades do Con nect, por isso é recomendável compreender a ordem referente aos itens a serem incluídos no *stack* de configurações. Caso você não respeite a ordem de inserção de cada elemento, sua aplicação não irá se comportar bem, gerando erros inesperados ou não realizando tais rotinas que fo ram estabelecidas. Para isso, é necessário sempre dar uma olhada na do cumentação (http://www.senchalabs.org/connect) para saber mais sobre esses itens e também sobre a ordem de cada um. Um detalhe importante é que na própria documentação esses itens já estão listados na ordem em que cada um deve ser incluído na *stack* de sua aplicação. |
| --- |

Em nossa aplicação apenas inserimos dois itens na *stack* de configurações: o *tem plate engine* EJS *(Embedded JavaScript)* e o diretório de arquivos estáticos. No decor rer deste livro serão incluídos e explicados novos itens no nosso projeto.

Também deixei apenas duas rotas: / e /usuarios, repare em como foram executados os seus callbacks; eles vieram da variável var routes =

37

4.5. Organizando os diretórios do projeto Casa do Código

require('./routes') pelo qual foi chamado um diretório e não um código ja vascript.

Por *default*, a chamada routes.index busca o arquivo index.js que con tenha a função: exports.index, esta é uma convenção do Express para incluir a rota raiz da aplicação. Já o routes.users.index seguiu a regra normal de carre gamento do arquivo users.js e sua respectiva função exports.index.

**4.5 Organizando os diretórios do projeto**

Quando o assunto é organização de códigos, o Express se comporta de forma bem flexível e liberal. Apesar de utilizar o seu *scaffold* de geração inicial, temos a total liberdade de modificar sua estrutura de diretórios e arquivos. Tudo vai depender da complexidade do projeto. Por exemplo, se o projeto for um sistema *single-page*, você pode desenvolver todo backend dentro código app.js, ou se o projeto possuir di versas rotas, *views*, *models* e *controllers*, o ideal seria montar uma estrutura modular utilizando o *pattern* MVC *(Model-View-Controller)*.

Em nosso projeto utilizaremos o padrão MVC, para isso falta criar os seguinte diretórios: models e controllers, deixando sua estrutura dessa forma:

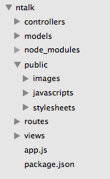


Figura 4.4: Estrutura de diretórios do ntalk.

Cada *model* que for utilizado em um *controller* realizará uma chamada a fun ção require('/models/nome-do-model');. Em *controllers*, ou qualquer ou

38

Casa do Código Capítulo 4. Iniciando com o Express

tro código, diversas chamadas a função require serão realizadas e isso vai gerar uma poluição nos códigos. O ideal é utilizarmos apenas para chamadas de módulos externos ou utilizar com frequência a função require dentro de app.js. Com base nesse problema, surgiu um *plugin* que visa minimizar essas chamadas, ele se chama **express-load**, sendo responsável por mapear diretórios para carregar e inje tar módulos dentro de uma variável que definirmos na função into. Para entender melhor o que será feito, adicione-o como dependência no package.json:

"dependencies": {

"express": "3.4.7",

"express-load": "1.1.8",

"ejs": "0.8.5"

}

Faça um update dos módulos para instalar o express-load, executando o comando: npm install.

Agora faremos um *refactoring* no app.js para implementarmos este módulo e utilizarmos sua função load():

var express = require('express')

, load = require('express-load')

, app = express();

// ...stack de configurações do servidor...

load('models')

.then('controllers')

.then('routes')

.into(app);

// ...app.listen(3000)...

É importante colocar em ordem os recursos a serem carregados pela função load(). Neste caso os *models* são carregados primeiro, em seguida vêm os *con trollers*, e por último os *routes*.

Continuando o *refactoring*, exclua o arquivo routes/user.js que foi ge rado pelo Express. E o código routes/index.js, vamos renomeá-lo para routes/home.js. Nele vamos incluir o seguinte código, que será uma adaptação para que ele utilize a variável app vinda do express-load:

39

4.5. Organizando os diretórios do projeto Casa do Código

module.exports = function(app) {

var home = app.controllers.home;

app.get('/', home.index);

};

O express-load criou um objeto chamado controllers dentro de app. Ele cria uma estrutura de objetos de acordo com o diretório e ar quivo. Neste caso o app.controllers.home esta se referenciando ao arquivo controllers/home.js. Agora para esta rota funcionar corretamente, vamos criar o controller home.js e incluir sua primeira *action* chamada de index, se guindo o exemplo abaixo:

module.exports = function(app) {

var HomeController = {

index: function(req, res) {

res.render('home/index');

}

};

return HomeController;

};

Pronto! Para finalizar o fluxo entre routes e controllers, exclua o arquivo views/index.ejs e crie o diretório views/home. A nossa homepage será uma simples tela de login para acessar o sistema. Para fins ilustrativos a futura lógica desse aplicativo será de implementar um login que autocadastra um novo usuário, quando for informado um login novo no sistema. Em views/home crie o arquivo index.ejs, pelo qual implementaremos um formulário que conterá os campos nome e email. Veja como será esta *view* com base na implementação abaixo:

*<!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>

40

Casa do Código Capítulo 4. Iniciando com o Express

<form action="/entrar" method="post">

<input type="text" name="nome" placeholder="Seu nome"> <br>

<input type="text" name="email" placeholder="Seu e-mail"> <br>

<button type="submit">Entrar</button>

</form>

</section>

<footer>

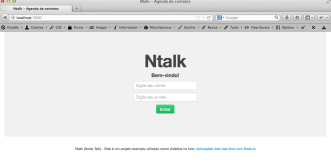
<small>Ntalk - Agenda de contatos</small>

</footer>

</body>

</html>

Vamos rodar o projeto? Execute no terminal o comando: node app.js, em seguida acesse em seu *browser* o endereço: http://localhost:3000 Se tudo deu certo você verá uma tela com formulário semelhante a imagem abaixo:

Figura 4.5: Tela de login do Ntalk.

Um detalhe a informar, nos exemplos deste livro não será implementado código css ou boas práticas de html, pois vamos focar apenas em código Javascript. Algumas imagens do projeto serão apresentados com um layout customizado. A versão com pleta deste projeto que inclui o código CSS e HTML de acordo com os *screenshots* pode ser acessada através do link do meu github:

http://github.com/caio-ribeiro-pereira/livro-nodejs/tree/master/projeto/ntalk 41

4.5. Organizando os diretórios do projeto Casa do Código Parabéns! Acabamos de implementar o fluxo da tela inicial do projeto.

42

Capítulo 5

Dominando o Express

**5.1 Estruturando views**

O módulo EJS possui diversas funcionalidades que permitem programar conteúdo dinâmico em código html. Não entraremos a fundo neste framework, apenas uti lizaremos seus principais recursos para renderizar conteúdo dinâmico e minimizar repetições de código. Com isso, isolaremos em outras *views*, conhecidas como *par tials*, possíveis códigos que serão reutilizados com maior frequência. Dentro do di retório views, vamos criar dois arquivos que serão reaproveitados na homepage. 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>

5.2. Controlando as sessões de usuários Casa do Código

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 chamem esses *partials* através da função include:

<% include ../header %>

<header>

<h1>Ntalk</h1>

<h4>Bem-vindo!</h4>

</header>

<section>

<form action="/entrar" method="post">

<input type="text" name="usuario[nome]"

placeholder="Digite o nome">

<br>

<input type="text" name="usuario[email]"

placeholder="Digite o e-mail">

<br>

<button type="submit">Entrar</button>

</form>

</section>

<% include ../footer %>

Agora a sua homepage ficou mais enxuta, fácil de ler e estruturada para reutili zação de *partials*.

**5.2 Controlando as sessões de usuários**

Para o sistema fazer *login* e *logout* é necessário ter um controle de sessão. Esse con trole permitirá que o usuário mantenha seus principais dados de acesso em memória no servidor, pois esses dados serão utilizados com maior frequência por grande parte do sistema. Trabalhar com sessão é muito simples, e os dados são manipulados atra vés de objeto JSON dentro da variável: req.session.

44

Casa do Código Capítulo 5. Dominando o Express

Vamos aplicar um controle de sessão?. Primeiro, temos que criar duas novas ro tas dentro de routes/home.js, sendo que uma será o app.post('/entrar', home.login) e a outra app.get('/sair', home.logout):

module.exports = function(app) {

var home = app.controllers.home;

app.get('/', home.index);

app.post('/entrar', home.login);

app.get('/sair', home.logout);

};

Depois implementaremos suas respectivas *actions* no controller/home.js, seguindo a convenção de nomes login e logout, que foram utilizados no routes/home.js. Na *action login* será implementada uma simples regra de vali dação dos campos nome e email vindos do formulário. Se os campos passarem na validação, esses dados serão armazenados na sessão em req.session.usuario, assim como criarmos um array vazio ( usuario['contatos'] = [];) de con tatos para no futuro utilizá-lo na gestão de contatos. Em seguida será feito um redirecionamento para rota **/contatos**. Na *action logout* é chamada a função: req.session.destroy(), que irá limpar os dados da sessão e gerar uma nova.

module.exports = function(app) {

var Usuario = app.models.usuario;

var HomeController = {

index: function(req, res) {

res.render('home/index');

},

login: function(req, res) {

var email = req.body.usuario.email

, nome = req.body.usuario.nome;

if(email && nome) {

var usuario = req.body.usuario;

usuario['contatos'] = [];

req.session.usuario = usuario;

res.redirect('/contatos');

} else {

res.redirect('/');

}

},

logout: function(req, res) {

45

5.2. Controlando as sessões de usuários Casa do Código

req.session.destroy();

res.redirect('/');

}

};

return HomeController;

};

Vamos testar essa implementação? Dê um restart no servidor. Para isso fina lize o servidor atual teclando no terminal CTRL+C (no Windows ou Linux) ou Command+C (no MacOSX), em seguida execute node app.js e depois em seu browser tente fazer um login no sistema.

Figura 5.1: Infelizmente deu mensagem de erro.

O que houve? Eu implementei tudo certo! Então, meu amigo, esse erro acon teceu porque faltou habilitar um novo item na *stack* de configurações. Esse item é responsável por receber os dados do formulário html e fazer um *parser* para objeto JSON, afinal ele não reconheceu o objeto req.body.usuario. Já adiantando, tam bém vamos incluir na *stack* um controle de sessão e cookies para que na próxima vez tudo funcione corretamente. Abaixo segue em ordem correta os novos itens da *stack* de configurações do nosso servidor:

app.set('views', \_\_dirname + '/views');

app.set('view engine', 'ejs');

app.use(express.cookieParser('ntalk'));

app.use(express.session());

app.use(express.json());

app.use(express.urlencoded());

app.use(express.static(\_\_dirname + '/public'));

46

Casa do Código Capítulo 5. Dominando o Express

É necessário incluir o express.cookieParser(), pois o express.session() utiliza-o para codificar e ou decodificar o SessionID que foi persistido no cookie. Outra configurações habilitadas foram o express.json() e express.urlencoded(), que são responsáveis por criar objetos JSON vindos de um formulário HTML. Ele

cria um objeto através dos atributos name e value , existentes nas tags <input> , <select> e <textarea> . Ao submeter um formulário com a tag: <input name="usuario[idade] value="23”> será criado um objeto dentro de req.body . Neste caso, será criado req.body.usuario.idade

| **Alguns cuidados ao trabalhar com Sessions**  Qualquer nome informado para o req.session será armazenado como um atributo deste objeto, por exemplo: req.session.mensagem = "Olá". Cuidado para não so brescrever os nomes de suas funções nativas, como por exemplo: req.session.destroy ou req.session.regenerate. Ao fazer isso, você sobrescreve essas funções desabilitando suas respectivas funcionalidades. Com isso, no decorrer de sua aplicação possíveis inesperados bugs acontecerão no sistema. Para entender melhor as funções da session, veja a documentação do Connect:  http://www.senchalabs.org/connect/session.html |
| --- |

Para finalizar, antes de testarmos as modificações, precisamos criar um controller e routes para contatos. Afinal temos a função res.redirect('/contatos') e até agora não foi implementada nenhuma lógica para realizar este redirecionamento.

A rota /contatos permite entrar em uma nova área do sistema, que será uma das principais áreas que iremos explorar no decorrer deste livro. Por enquanto vamos simplificar a implementação criando um *controller*, um rota e uma *view* para ele.

Crie o diretório contatos dentro de *views* e codifique o arquivo index.ejs com o seguinte conteúdo:

<% include ../header %>

<header>

47

5.2. Controlando as sessões de usuários Casa do Código

<h2>Ntalk - Agenda de contatos</h2>

</header>

<section>

<p>Bem-vindo <%- usuario.nome %></p>

</section>

<% include ../exit %>

<% include ../footer %>

Nesta *view* vamos renderizar o nome do usuário logado, para isso utilizamos a função: <%- usuario.nome %>. Este objeto será enviado pelo controller que implementaremos a seguir. Repare que desta vez reaproveitamos o header.ejs e footer.ejs e isso tornou muito mais limpo esta *view*. Também incluímos um novo *partial* - <% include ../exit %>. Basicamente, ele possui o link **Sair**, que simplesmente faz *logout* no sistema. Ele será reaproveitado por grande parte do sistema, então iremos criá-lo dentro de views/exit.ejs:

<section>

<a href='/sair'>Sair</a>

</section>

Agora vamos criar o seu controller chamado de contatos.js, contendo ape nas a *action* index. Ele basicamente vai pegar os dados de um usuário logado atra vés da variável req.session.usuario e vai enviá-los para *view*através da função: res.render().

module.exports = function(app) {

var ContatoController = {

index: function(req, res) {

var usuario = req.session.usuario

, params = {usuario: usuario};

res.render('contatos/index', params);

}

}

return ContatoController;

};

Para finalizar, criaremos um novo *routes*, por convenção, utilizamos o mesmo nome do seu *controller* e por enquanto, vamos incluir a rota GET com o path: /contatos.

48

Casa do Código Capítulo 5. Dominando o Express

module.exports = function(app) {

var contatos = app.controllers.contatos;

app.get('/contatos', contatos.index);

};

Vamos testar novamente? Reinicie o servidor teclando no terminal CTRL+C (no Windows ou Linux) ou Command+C (no MacOSX), em seguida execute node app.js, por último acesse em seu browser: http://localhost:3000 . Faça novamente um login no sistema, dessa vez temos uma nova tela no sistema, a agenda de contatos.

Acabamos de expandir nosso projeto, incluímos a área principal. Aprendemos como habilitar session para implementar um controle de *login* e *logout*. Deixamos todas as views enxutas, isolando possíveis trechos de repetição de código html, além explorarmos novos itens da *stack* de configuração do servidor Node.

**5.3 Criando rotas no padrão REST**

A nossa agenda de contatos precisa ter como requisito mínimo um meio de permi tir o usuário criar, listar, atualizar e excluir contatos. Esse é o conjunto clássico de funcionalidades, mais conhecido como CRUD *(Create, Receive, Update e Delete)*.

As rotas que utilizaremos para implementar o CRUD da agenda de contatos será aplicando o padrão de rotas REST. Esse padrão consiste em criar rotas utilizando os principais métodos do HTTP (GET, POST, PUT e DELETE). Para isso teremos que habilitar um novo item na *stack* de configurações no app.js:

app.set('views', \_\_dirname + '/views');

app.set('view engine', 'ejs');

app.use(express.cookieParser('ntalk'));

app.use(express.session());

app.use(express.json());

app.use(express.urlencoded());

app.use(express.methodOverride());

app.use(app.router);

app.use(express.static(\_\_dirname + '/public'));

Incluímos o express.methodOverride() que permite utilizar um mesmo path entre os métodos do HTTP, fazendo uma sobrescrita de métodos. Também foi adicionado o *middleware* app.router, que gerencia as rotas da aplicação, permi tindo a implementação de rotas para páginas de erros (muita calma! Isso ainda será

49

5.3. Criando rotas no padrão REST Casa do Código

explicado neste capítulo.) e rotas para arquivos estáticos, sem conflitar com as rotas da aplicação.

Abra o arquivo routes/contatos.js, nele vamos implementar as futuras rotas para implementar o CRUD da agenda de contatos:

module.exports = function(app) {

var contatos = app.controllers.contatos;

app.get('/contatos', contatos.index);

app.get('/contato/:id', contatos.show);

app.post('/contato', contatos.create);

app.get('/contato/:id/editar', contatos.edit);

app.put('/contato/:id', contatos.update);

app.del('/contato/:id', contatos.destroy);

};

Com as rotas criadas, precisamos agora implementar as regras de negócio den tro do controller contatos.js. Como ainda não utilizaremos um banco de da dos, todos os dados serão persistidos na própria sessão do usuário, ou seja, todos os contatos serão gravados em memória e não em um banco de dados dedicado. Em controller/contatos.js implemente a seguinte lógica na *action* index:

module.exports = function(app) {

var ContatoController = {

index: function(req, res) {

var usuario = req.session.usuario

, contatos = usuario.contatos

, params = {usuario: usuario

, contatos: contatos};

res.render('contatos/index', params);

},

// continuação do controller...

}

Já na *action* create utilizaremos um simples *array* para persistir os contatos do usuário — usuario.contatos.push(contato); — para em seguida redireci onar o usuário para rota **/contatos**:

create: function(req, res) {

var contato = req.body.contato

, usuario = req.session.usuario;

usuario.contatos.push(contato);

50

Casa do Código Capítulo 5. Dominando o Express

res.redirect('/contatos');

},

// continuação do controller...

Em show e edit enviamos via parâmetro no path, o ID do usuário. Neste caso, passamos apenas o índice do contato referente a sua posição no ar ray e em seguida enviamos o contato para a renderização de sua respectiva view res.render('contatos/show') e res.render('contatos/edit');:

show: function(req, res) {

var id = req.params.id

, contato = req.session.usuario.contatos[id]

, params = {contato: contato, id: id};

res.render('contatos/show', params);

},

edit: function(req, res) {

var id = req.params.id

, usuario = req.session.usuario

, contato = usuario.contatos[id]

, params = {usuario: usuario

, contato: contato

, id: id};

res.render('contatos/edit', params);

},

// continuação do controller...

Agora temos a *actions* update. Ela recebe os dados de um contato atualizado que são submetidos pelo formulário da view contatos/edit. Em seguida utiliza mos seu índice via req.params.id para atualizar no array o contato.

update: function(req, res) {

var contato = req.body.contato

, usuario = req.session.usuario;

usuario.contatos[req.params.id] = contato;

res.redirect('/contatos');

},

// continuação do controller...

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

51

5.3. Criando rotas no padrão REST Casa do Código

destroy: function(req, res) {

var usuario = req.session.usuario

, id = req.params.id;

usuario.contatos.splice(id, 1);

res.redirect('/contatos');

}

// fim do controller...

return ContatoController;

};

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

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

<% include ../header %>

<header>

<h2>Ntalk - Agenda de contatos</h2>

</header>

<section>

<form action="/contato" method="post">

<input type="text" name="contato[nome]" placeholder="Nome"> <input type="text" name="contato[email]" placeholder="E-mail"> <button type="submit">Cadastrar</button>

</form>

<table>

<thead>

<tr>

<th>Nome</th>

<th>E-mail</th>

<th>Ação</th>

</tr>

</thead>

<tbody>

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

<tr>

<td><%- contato.nome %></td>

<td><%- contato.email %></td>

<td>

52

Casa do Código Capítulo 5. Dominando o Express

<a href="/contato/<%- index %>">Detalhes</a>

</td>

</tr>

<% }) %>

</tbody>

</table>

</section>

<% include ../exit %>

<% include ../footer %>

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

<% include ../header %>

<header>

<h2>Ntalk - Editar contato</h2>

</header>

<section>

<form action="/contato/<%- id %>" method="post">

<input type="hidden" name="\_method" value="put">

<label>Nome:</label>

<input type="text" name="contato[nome]"

value="<%- contato.nome %>">

<label>E-mail:</label>

<input type="text" name="contato[email]"

value="<%- contato.email %>">

<button type="submit">Atualizar</button>

</form>

</section>

<% include ../exit %>

<% include ../footer %>

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

<% include ../header %>

<header>

<h2>Ntalk - Dados do contato</h2>

</header>

<section>

<form action="/contato/<%- id %>" method="post">

53

5.4. Aplicando filtros antes de acessar as rotas Casa do Código

<input type="hidden" name="\_method" value="delete">

<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 utilizar 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">  A solução paliativa é criar a tag <input> com o nome \_method e o valor put ou delete, para sobrescrever o método POST ou GET da tag <form>. Veja o exemplo abaixo:  <input type="hidden" name="\_method" value="put"> <input type="hidden" name="\_method" value="delete"> |
| --- |

**5.4 Aplicando filtros antes de acessar as rotas**

Já percebeu que quando acessamos a rota /contatos sem logar no sistema acon tece um bug? Não? Tente agora mesmo! Mas o que realmente provocou este erro? Bem, quando fazemos um login com sucesso, armazenamos os principais dados do usuário na sessão para utilizá-los no decorrer da aplicação. Quando acessa mos /contatos sem antes ter feito um login, o controller tenta acessar a variá vel req.session.usuario que não existe ainda. Isso causa o seguinte bug na aplicação:

54

Casa do Código Capítulo 5. Dominando o Express Figura 5.2: Bug: Cannot read property ‘usuario’ of undefined.

Diferente do Rails, Sinatra ou Django, o Express não possui filtros de forma ex plícita e em código legível. Não existe uma função before ou after pela qual se possa processar algo antes ou depois de entrar em uma rota. Mas calma! Nem tudo esta 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 callback encadeados em uma rota. Resumindo, quando criamos uma rota, após informar no primeiro parâmetro a sua string, no segundo parâmetro em diante é possível incluir callbacks que são executados de forma ordenada, por exem plo: app.get('/', callback1, callback2, callback3).

Para implementarmos isso, primeiro vamos criar o filtro de autenticação. Ele será a criação do nosso primeiro *middleware* e será utilizado na maioria das rotas. Na raiz do projeto crie a pasta *middleware* incluindo o arquivo autenticador.js, nele implemente a lógica abaixo:

module.exports = function(req, res, next) {

if(!req.session.usuario) {

return res.redirect('/');

}

return next();

};

Esse filtro faz uma simples verificação se existe um usuário dentro da ses são. Se o usuário estiver autenticado, ou seja, estiver na session, será execu tado o callback return next() responsável por pular este filtro e indo para fun ção ao lado. Caso autenticação não aconteça, executamos um simples return

55

5.5. Indo além: criando páginas de erros amigáveis Casa do Código

res.redirect('/'), que faz o usuário voltar para página inicial e impedindo que ocorra o bug.

Com esse filtro implementado, agora temos que injetá-lo nos call backs das rotas que precisam desse tratamento, faremos essas alterações no routes/contatos.js, de acordo com o código a seguir:

module.exports = function(app) {

var autenticar = require('./../middleware/autenticador') , contatos = app.controllers.contatos;

app.get('/contatos', autenticar, contatos.index);

app.get('/contato/:id', autenticar, contatos.show);

app.post('/contato', autenticar, contatos.create);

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

Basicamente inserimos o callback autenticar antes da função principal da rota. Isso nos permitiu emular a execução de um filtro before. Caso queira criar um filtro after, não há segredos: apenas coloque o callback do filtro por último. O importante é colocar as funções na ordem lógica de suas execuções.

**5.5 Indo além: criando páginas de erros amigáveis**

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

| **Sobre o código de erros do HTTP**  O protocolo HTTP tem diversos tipos de erros. O órgão W3C pos sui uma documentação explicando em detalhes o comportamento e có digo de cada erro. Para ficar por dentro desse assunto veja nesse link (http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html) a especifica ção dos status de erro gerados por este protocolo. |
| --- |

56

Casa do Código Capítulo 5. Dominando o Express

Que tal implementarmos um controle de erros para renderizar aos usuários pá ginas customizadas e mais amigáveis? Primeiro, crie duas novas views, uma para apresentar a tela do erro 404 conhecida na web como **"Página não encontrada”**, seu nome será 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 %>

...e a outra será focada em mostrar erros gerados pelo protocolo HTTP com o nome de 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 %>

Feito isso, agora só precisamos adicionar dois novos itens na *stack* de configura ções do app.js para renderizar as respectivas views de erros:

app.set('views', \_\_dirname + '/views');

app.set('view engine', 'ejs');

app.use(express.cookieParser('ntalk'));

app.use(express.session());

app.use(express.json());

app.use(express.urlencoded());

app.use(app.router);

app.use(express.static(\_\_dirname + '/public'));

57

5.5. Indo além: criando páginas de erros amigáveis Casa do Código

app.use(function(req, res, next) {

res.status(404);

res.render('not-found');

});

app.use(function(error, req, res, next) {

res.status(500);

res.render('server-error', error);

});

Para minimizar essa poluição de código na *stack* do servidor, exporte as funções de erros para um novo arquivo chamado error.js dentro do diretório *middleware*, deixando-as da seguinte forma:

exports.notFound = function(req, res, next) {

res.status(404);

res.render('not-found');

};

exports.serverError = function(error, req, res, next) { res.status(500);

res.render('server-error', {error: error});

};

Em seguida modifique o *stack* do app.js para ficar mais limpo:

var express = require('express')

, app = express()

, load = require('express-load')

, error = require('./middleware/error');

app.set('views', \_\_dirname + '/views');

app.set('view engine', 'ejs');

app.use(express.cookieParser('ntalk'));

app.use(express.session());

app.use(express.json());

app.use(express.urlencoded());

app.use(app.router);

app.use(express.static(\_\_dirname + '/public'));

app.use(error.notFound);

app.use(error.serverError);

Vamos testar o nosso código? Reinicie o servidor e, no browser, digite pro positalmente uma nome de uma rota que não exista na aplicação, por exemplo:

58

Casa do Código Capítulo 5. Dominando o Express

http://localhost:3000/url-errada . Esta foi a renderização da tela de erro para página não encontrada (erro 404).

Figura 5.3: Tela de erro 404.

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

Figura 5.4: Tela de erro 500.

Parabéns! Agora temos uma aplicação que cadastra, edita, exclui e lista conta tos. Utilizamos o padrão de rotas REST, criamos um simples controle de login que mantém o usuário na sessão, implementamos filtros para barrar acesso de usuários não autenticados e pra finalizar, implementamos a renderização de páginas de er

59

5.5. Indo além: criando páginas de erros amigáveis Casa do Código

ros amigáveis — afinal erros inesperados podem ocorrer em nossa aplicação e seria desagradável o usuário ver informações complexas na tela.

60

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.

6.2. Conhecendo o framework Socket.IO Casa do Código 

Figura 6.1: Imagem explicando sobre conexão bidirecional.

O Node.js se tornou popular por oferecer bibliotecas de baixo nível que supor tam 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 ser vidor.

Infelizmente 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*.

**6.2 Conhecendo o framework Socket.IO**

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

Figura 6.2: Framework Socket.IO.

62

Casa do Código Capítulo 6. Programando sistemas real-time

O Socket.IO funciona da seguinte maneira: é incluído um script no cliente que detecta informações sobre o browser do usuário para definir qual será a melhor co municação com o servidor. Os *transports* de comunicação que ele executa são:

1) *WebSocket*;

2) *Adobe Flash Socket*;

3) *AJAX long polling*;

4) *AJAX multipart streaming*;

5) *Forever iframe*;

6) *JSONP Polling*;

Se o navegador do usuário possuir compatibilidade com*WebSockets* ou *FlashSoc kets* (utilizando *Adobe Flash Player* do navegador), será realizada uma comunicação bidirecional. Caso contrário, será emulada uma comunicação unidirecional, que em curtos intervalos de tempo faz requisições AJAX no servidor. É claro que o desem penho 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 o desenvolvedor.

**6.3 Implementando um chat real-time**

Vamos ver como funciona na prática? Criaremos o web chat no ntalk, com o qual o usuário enviará mensagens para os usuários online da agenda de contatos. Inte graremos o frameworks: Socket.IO no Express. Primeiro, instalaremos o módulo atualizando o package.json:

"dependencies": {

"express": "3.4.7",

"express-load": "1.1.8",

"ejs": "0.8.5",

"socket.io": "0.9.16"

}

63

6.3. Implementando um chat real-time Casa do Código

Utilizaremos a versão 0.9.16 do Socket.IO, agora vamos instalá-lo executando no console: npm install

Vamos adaptar o app.js com o novo módulo. A função *listen* do servidor web será realizada via módulo http através da função server.listen(3000) para que o Socket.IO utilize-a para criar seu ponto de comunicação através do protocolo HTTP.

var express = require('express')

, app = express()

, load = require('express-load')

, error = require('./middleware/error')

, server = require('http').createServer(app)

, io = require('socket.io').listen(server);

// ...código do stack de configurações...

// ...código da função load()...

io.sockets.on('connection', function (client) {

client.on('send-server', function (data) {

var msg = "<b>"+data.nome+":</b> "+data.msg+"<br>";

client.emit('send-client', msg);

client.broadcast.emit('send-client', msg);

});

});

server.listen(3000, function(){

console.log("Ntalk no ar.");

});

Instanciamos os módulos: express, http e socket.io. É ne cessário seguir esta ordem, pois o Socket.IO funciona a partir de uma instância do módulo http. Toda brincadeira começa a partir do evento: io.sockets.on('connection'). Esta função espera um cliente interagir com um de seus eventos internos. Por enquanto, o único evento interno é o client.on('send-server'), cuja sua execução ocorre quando um cliente en via uma mensagem para o servidor. Perceba que pouco código foi incluído e o servi dor responde a seus clientes através das funções client.emit('send-client') e client.broadcast.emit('send-client'). Resumindo, o fluxo básico é o cliente enviar uma mensagem para o servidor e o servidor responde o próprio cliente (via client.emit()) e seus demais clientes conectados (via

64