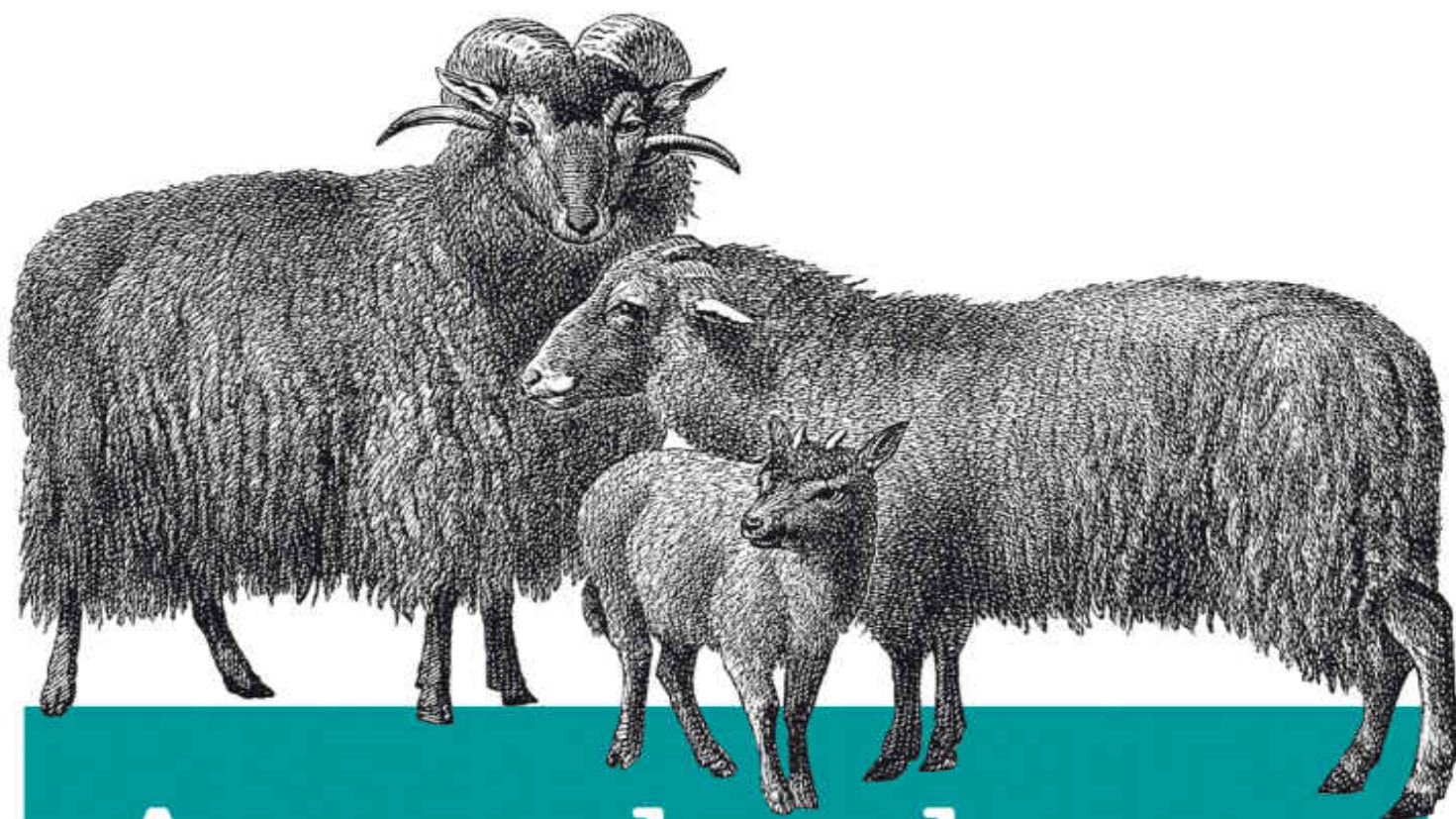


O'REILLY®



Aprendendo a desenvolver aplicações web

DESENVOLVA RAPIDAMENTE COM AS TECNOLOGIAS JAVASCRIPT MAIS MODERNAS

novatec

Semmy Purewal

Semmy Purewal

Novatec

Authorized Portuguese translation of the English edition of titled Learning Web App Development, ISBN 9781449370190 © 2014 Semmy Purewal. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Tradução em português autorizada da edição em inglês da obra Learning Web App Development, ISBN 9781449370190 © 2014 Semmy Purewal. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

© Novatec Editora Ltda. 2014.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Lúcia Kinoshita

Revisão gramatical: Marta Almeida de Sá

Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-738-1

Histórico de edições impressas:

Julho/2014 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Email: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

*Aos meus pais.
Obrigado por todo o seu apoio e pelo seu incentivo ao longo dos
anos!*

Sumário

Prefácio

Capítulo 1 . Fluxo de trabalho

Editores de texto

Instalando o Sublime Text

Básico sobre o Sublime Text

Controle de versões

Instalando o Git

Básico sobre a linha de comando Unix

Básico sobre o Git

Navegadores

Instalando o Chrome

Resumo

Práticas e leituras adicionais

Sublime Text

Emacs e Vim

Linha de comando Unix

Mais a respeito do Git

GitHub

Capítulo 2 . Estrutura

Hello, HTML!

Tags versus conteúdo

<p> de parágrafo

Comentários

Cabeçalhos e âncoras e listas, oh, céus!

Generalizações

Document Object Model e as árvores

Usando a validação de HTML para identificar problemas

Aplicação web Amazeriffic

[Identificando a estrutura](#)

[Visualizando a estrutura por meio de uma árvore](#)

[Implementando a estrutura com o nosso fluxo de trabalho](#)

[Estruturando o conteúdo principal](#)

[Estruturando o rodapé](#)

[Resumo](#)

[Práticas e leituras adicionais](#)

[Memorização](#)

[Diagramas de árvore](#)

[Criando a página de FAQ para o Amazeriffic](#)

[Mais sobre o HTML](#)

Capítulo 3 ■ Estilo

[Hello, CSS!](#)

[Conjuntos de regras](#)

[Comentários](#)

[Padding, borda e margem](#)

[Seletores](#)

[Classes](#)

[Pseudoclasses](#)

[Seletores mais complexos](#)

[Regras em cascata](#)

[Herança](#)

[Layouts com floats](#)

[Propriedade clear](#)

[Trabalhando com fontes](#)

[Eliminando as inconsistências dos navegadores](#)

[Usando o CSS Lint para identificar problemas em potencial](#)

[Interagindo e resolvendo problemas com as ferramentas para desenvolvedores do Chrome](#)

[Estilizando o Amazeriffic](#)

[A grade](#)

[Criando as colunas](#)

[Adicionando e manipulando fontes](#)

[Algumas modificações adicionais](#)

[Resumo](#)

[Práticas e leituras adicionais](#)

[Memorização](#)

[Exercícios com os seletores CSS](#)

[Estilize a página de FAQ do Amazeriffic](#)

[Regras em cascata](#)

[Capacidade de ser responsivo e bibliotecas responsivas](#)

Capítulo 4 ■ Interatividade

[Hello, JavaScript!](#)

[Nossa primeira aplicação interativa](#)

[A estrutura](#)

[O estilo](#)

[Interatividade](#)

[Generalização da jQuery.](#)

[Criando um projeto](#)

[Comentários](#)

[Seletores](#)

[Manipulação do DOM](#)

[Eventos e programação assíncrona](#)

[Generalizações para o JavaScript](#)

[Interagindo com o JavaScript no JavaScript Console do Chrome](#)

[Variáveis e tipos](#)

[Funções](#)

[Seleção](#)

[Iteração](#)

[Arrays](#)

[Usando o JSLint para identificar possíveis problemas](#)

[Acrescentando interatividade ao Amazeriffic](#)

[Iniciando](#)

[A estrutura e o estilo](#)

[A interatividade](#)

[Resumo](#)

[Práticas e leituras adicionais](#)

[Plug-ins da jQuery.](#)

[Seletores jQuery.](#)

[FizzBuzz](#)

[Exercícios com arrays](#)
[Projeto Euler](#)
[Outras referências ao JavaScript](#)

Capítulo 5 ■ A ponte entre o cliente e o servidor

[Hello, objetos JavaScript!](#)
[Representando um jogo de baralho](#)
[Generalizações](#)
[A comunicação entre computadores](#)
[JSON](#)
[AJAX](#)
[Acessando um arquivo JSON externo](#)
[Passando por cima de restrições de segurança dos navegadores](#)
[Função getJSON](#)
[Um array JSON](#)
[E daí?](#)
[Obtendo imagens do Flickr](#)
[Adicionando um recurso de tags ao Amazeriffic](#)
[A função map](#)
[Acrescentando uma aba Tags](#)
[Criando a UI](#)
[Criando uma estrutura de dados intermediária para as tags](#)
[As tags como parte de nossa entrada](#)
[Resumo](#)
[Práticas e leituras adicionais](#)
[Exercício com objetos](#)
[Outras APIs](#)

Capítulo 6 ■ Servidor

[Configurando o ambiente](#)
[Instalando o VirtualBox e o Vagrant](#)
[Criando a sua máquina virtual](#)
[Conectando-se à sua máquina virtual usando o SSH](#)
[Hello, Node.js!](#)

[Modelos mentais](#)

[Clientes e servidores](#)

[Hosts e guests](#)

[Aspectos práticos](#)

[Hello, HTTP!](#)

[Os módulos e o Express](#)

[Instalando o Express com o NPM](#)

[Nosso primeiro servidor Express](#)

[Enviando a sua aplicação cliente](#)

[Generalizações](#)

[Contabilizando tuítes](#)

[Obtendo as suas credenciais do Twitter](#)

[Conectando-se com a API do Twitter](#)

[O que está acontecendo aqui?](#)

[Armazenando contadores](#)

[Modularizando o nosso contador de tuítes](#)

[Importando o nosso módulo no Express](#)

[Criando um cliente](#)

[Criando um servidor para o Amazeriffic](#)

[Criando os nossos diretórios](#)

[Inicializando um repositório Git](#)

[Criando o servidor](#)

[Executando o servidor](#)

[Enviando informações ao servidor](#)

[Resumo](#)

[Práticas e leituras adicionais](#)

[O JSHint e o CSS Lint usando o NPM](#)

[Generalizando o nosso código de contagem de tuítes](#)

[API para pôquer](#)

Capítulo 7 ■ **Armazenamento de dados**

[NoSQL versus SQL](#)

[Redis](#)

[Interagindo com o cliente de linha de comando do Redis](#)

[Instalando o módulo Redis por meio de um arquivo package.json](#)

[Interagindo com o Redis em nosso código](#)

[Inicializando os contadores a partir dos dados armazenados no Redis](#)

[Utilizando mget para obter vários valores](#)

[MongoDB](#)

[Interagindo com o cliente de linha de comando do MongoDB](#)

[Modelando dados com o Mongoose](#)

[Armazenando os Todos do Amazeriffic](#)

[Resumo](#)

[Práticas e leituras adicionais](#)

[Outras referências a bancos de dados](#)

Capítulo 8 ■ Plataforma

[O Cloud Foundry](#)

[Criando uma conta](#)

[Preparando a sua aplicação para a implantação](#)

[Fazendo a implantação de nossa aplicação](#)

[Obtendo informações sobre suas aplicações](#)

[Atualizando a sua aplicação](#)

[Apagando aplicações do Cloud Foundry](#)

[As dependências e o package.json](#)

[Associando o Redis à sua aplicação](#)

[Associando o MongoDB à sua aplicação](#)

[Resumo](#)

[Práticas e leituras adicionais](#)

[Outras plataformas](#)

Capítulo 9 ■ Aplicação

[Efetuando a refatoração de nosso cliente](#)

[Generalizando conceitos significativos](#)

[Incluindo o AJAX em nossas abas](#)

[Livrando-se dos hacks](#)

[Tratando erros do AJAX](#)

[Efetuando a refatoração de nosso servidor](#)

[Organização do código](#)

[Verbos HTTP, CRUD e REST](#)

[Definindo rotas de acordo com o ID](#)

[Utilizando a jQuery em solicitações put e delete](#)

[Códigos de resposta HTTP](#)

[Model-View-Controller](#)

[Adicionando usuários ao Amazeriffic](#)

[Criando o modelo para os usuários](#)

[Criando o controlador para os usuários](#)

[Definindo as rotas](#)

[Aperfeiçoando as ações de nosso controlador de Todos](#)

[Resumo](#)

[Práticas e leituras adicionais](#)

[Adicionando uma página de administração de usuários](#)

[Visões que utilizam EJS e Jade](#)

[Crie outra aplicação](#)

[Ruby On Rails](#)

[Sobre o autor](#)

[Colofão](#)

Prefácio

No início de 2008, após ter feito cerca de seis anos de pós-graduação e de lecionar em tempo parcial, me vi na esperança de conseguir um emprego como professor universitário de ciência da computação em tempo integral. Não levei muito tempo para perceber que empregos desse tipo são realmente difíceis de conseguir, e ter um bom emprego como professor universitário tem quase tanto a ver com a sorte quanto com qualquer outro fator. Desse modo, fiz o que qualquer acadêmico que se preza faria diante de um mercado de trabalho acadêmico assustador: decidi me preparar para arrumar um emprego aprendendo a desenvolver aplicações web.

Isso pode soar um pouco estranho. Afinal de contas, eu havia estudado ciência da computação durante cerca de nove anos até aquela época então e havia ensinado aos alunos a desenvolver softwares durante aproximadamente seis anos. Eu *já não deveria saber* como desenvolver aplicações web? O fato é que existe uma enorme distância entre a engenharia de software prática e cotidiana e a programação da forma como é ensinada nos departamentos de ciência da computação nas faculdades e nas universidades. Com efeito, o meu conhecimento de desenvolvimento web era limitado ao HTML e a um pouco de CSS que eu havia aprendido por conta própria naquela época.

Felizmente, eu tinha diversos amigos que estavam trabalhando ativamente no mundo do desenvolvimento web, e a maioria deles parecia estar falando sobre um framework (relativamente) novo chamado Ruby on Rails. Parecia ser um bom tema no qual concentrar meus esforços. Desse modo, comprei vários livros sobre o assunto e comecei a ler tutoriais online para me preparar.

E, após alguns meses realmente tentando entender o assunto, quase desisti.

Por quê? Porque a maioria dos livros e dos tutoriais partia da pressuposição de que eu já estava desenvolvendo aplicações web havia anos! E, apesar de ter uma base anterior bem sólida em programação de computadores, achei todos os materiais extremamente concisos demais e era difícil acompanhá-los. Por exemplo, você pode ter assistido a uma quantidade enorme de aulas de ciência da computação sem jamais ter se deparado com o padrão de projeto Model-View-Controller (Modelo-Visão-Controlador), e alguns livros supunham que você já entendia desse assunto no primeiro capítulo!

Apesar de tudo, consegui aprender o suficiente sobre desenvolvimento de aplicações web a ponto de conseguir alguns trabalhos temporários de consultoria que me sustentassem até que consegui um cargo como professor. E, nesse processo, descobri que eu gostava tanto dos aspectos práticos da área que continuei a prestar consultoria, além de ensinar.

Após alguns anos realizando ambas as atividades, tive a oportunidade de dar minha primeira aula de Web Application Development (Desenvolvimento de aplicações web) na University of North Carolina, em Asheville. Minha ideia inicial foi *começar* com o Ruby on Rails, porém, quando comecei a ler os livros e tutorias mais recentes, percebi que eles não haviam melhorado muito ao longo dos anos. Isso não quer dizer que eles não sejam bons recursos para as pessoas que já têm um conhecimento prévio a respeito do básico; eles simplesmente não pareciam ser adequados aos estudantes aos quais eu estava lecionando.

Infelizmente, mas não de forma surpreendente, os livros acadêmicos sobre desenvolvimento web são muito piores! Muitos deles contêm conceitos e *idioms*¹ obsoletos e não abordam os assuntos de modo a tornar plataformas como o Ruby on Rails mais acessíveis. Cheguei até mesmo a revisar um livro atualizado em

2011 que continuava usando layouts baseados em tabela e a tag ``!

Não tive outra opção senão desenvolver meu curso a partir do zero, criando, eu mesmo, todo o material. Na época, eu havia feito pequenos trabalhos em algumas consultorias com o Node.js (JavaScript do lado do servidor), então achei que seria interessante tentar dar um curso que incluísse a mesma linguagem no cliente e no servidor. Além do mais, fiz com que o meu objetivo fosse proporcionar uma base suficiente aos alunos para que eles pudessem começar a estudar o Ruby on Rails por conta própria, caso decidissem continuar.

Este livro é constituído, em sua maior parte, pelo material que desenvolvi enquanto lecionei esse curso na UNCA. Ele apresenta o desenvolvimento de uma aplicação web básica a partir do zero, baseada em banco de dados e que usa JavaScript. O processo inclui um fluxo de trabalho básico para o desenvolvimento web (usando um editor de texto e controle de versões), o básico das tecnologias do lado do cliente (HTML, CSS, jQuery, JavaScript), o básico das tecnologias do lado do servidor (Node.js, HTTP, bancos de dados), o básico da implantação na nuvem (Cloud Foundry) e algumas práticas essenciais de boa codificação (funções, MVC, DRY). Ao longo do caminho, iremos explorar alguns dos fundamentos da linguagem JavaScript, como programar usando arrays e objetos, e os modelos mentais que acompanham esse tipo de desenvolvimento de software.

Opções de tecnologia

Para o controle de versões, escolhi o Git porque... bem... é o Git e ele é incrível. Além do mais, meus alunos tiveram a oportunidade de aprender a usar o GitHub, que está se tornando imensamente popular. Embora eu não discuta o GitHub neste livro, ele é bem fácil de entender, uma vez que você compreenda o Git.

Decidi usar a jQuery do lado do cliente porque ela continua sendo

relativamente popular e eu gosto de trabalhar com essa biblioteca. Não usei nenhum outro framework no cliente, embora eu mencione o Twitter Bootstrap e o Zurb Foundation no capítulo 3. Optei por me manter longe dos frameworks modernos do lado do cliente, como o Backbone ou o Ember, porque eu os acho confusos para as pessoas que estão simplesmente começando. Entretanto, assim como no caso do Rails, você deverá ser capaz de mergulhar facilmente de cabeça nesses frameworks após a leitura deste livro.

Do lado do servidor, escolhi o Express porque ele é (relativamente) leve e flexível. Decidi não incluir templating do lado do cliente e do servidor porque acho essencial aprender a fazer tudo manualmente antes.

Preferi não usar bancos de dados relacionais porque me pareceu que eu não seria capaz de dar uma visão geral significativa sobre o assunto no período de tempo que eu havia reservado para esse aspecto no curso. Em vez disso, escolhi o MongoDB porque ele é amplamente utilizado na comunidade Node.js e usa o JavaScript como linguagem de query. E eu realmente gosto muito do Redis, portanto ele também foi incluído.

Selecionei a Cloud Foundry como a plataforma de implantação porque, das três que considereei (incluindo o Heroku e o Nodejitsu), ela é a única que oferece um período de trial gratuito e não exige um cartão de crédito para a instalação de serviços externos. Apesar disso, as diferenças entre as plataformas não são enormes, e mudar de uma para outra não deve ser muito difícil.

Este livro é adequado a você?

Este livro não foi concebido para transformar você em um “ninja” ou em uma “estrela do rock”, e nem mesmo em um bom programador de computadores em particular. Ele não vai prepará-lo para que você tenha um emprego imediato, nem posso prometer que ele vá mostrar a “maneira certa” de fazer as coisas.

Por outro lado, o livro proporcionará uma base sólida sobre os assuntos essenciais necessários para entender de que maneira as peças de uma aplicação web moderna se encaixam, e oferecerá um ponto de partida para estudos adicionais sobre o assunto. Se conseguir acompanhar este livro, você aprenderá tudo o que eu gostaria de já ter sabido quando comecei a trabalhar com o Rails.

Você tirará o máximo de proveito deste livro se tiver um pouco de experiência com programação e nenhuma experiência anterior com desenvolvimento web. No mínimo, é provável que você já deva ter visto construções básicas de programação como instruções `if-else`, loops (laços), variáveis e tipos de dados. No entanto não irei supor que você tenha qualquer experiência com programação orientada a objetos, nem com qualquer linguagem de programação em particular. O conhecimento prévio necessário poderá ser adquirido se você seguir os tutoriais da Khan Academy (www.khanacademy.org/) ou da Code Academy (www.codecademy.com/), ou se fizer um curso de programação em alguma instituição local.

Além de ser usado para estudos por conta própria, espero que este livro possa servir como um livro a ser adotado em cursos de desenvolvimento de aplicações web ou, quem sabe, em um curso de nível universitário de um semestre (14 semanas).

Aprendendo com este livro

Desenvolver aplicações web definitivamente é uma habilidade que você deverá adquirir por meio da prática. Com isso em mente, escrevi este livro para que fosse lido de maneira ativa. Isso quer dizer que você irá tirar o máximo de proveito dele se estiver sentado diante de um computador enquanto lê o livro e se você realmente digitar todos os exemplos.

É claro que essa abordagem em particular está repleta de perigos – há o risco de os exemplos de código não funcionarem se você não os digitar exatamente da forma como aparecem. Para atenuar esse

risco, criei um repositório no GitHub contendo todos os exemplos do livro na ordem em que serão trabalhados. Eles podem ser visualizados na web em <http://www.github.com/semmypurewal/LearningWebAppDev>. Como os exemplos completos se encontram nesse local, tentei evitar a inclusão de listagens completas ao longo do livro para evitar a redundância.

Além do mais, deixei grandes partes do projeto sem conclusão. Quando faço isso é porque quero que você tente finalizá-las por conta própria. Sugiro que você faça isso antes de ver os exemplos completos que publiquei online. Todo capítulo termina com um conjunto de problemas práticos e referências para mais informações, portanto sugiro que você efetue a complementação com eles também.

Ensinando com este livro

Quando leciono usando este material em um curso de 14 semanas, normalmente gasto de duas a três semanas no material contido nos três primeiros capítulos e de três a quatro semanas no material dos últimos três capítulos. Isso significa que invisto a maior parte do tempo nos três capítulos intermediários, que abordam a programação JavaScript, a jQuery, o AJAX e o Node.js. Os alunos a quem leciono parecem ter mais dificuldades com arrays e objetos, portanto invisto um tempo adicional nesses assuntos porque acho que eles são bastante essenciais na programação de computadores em geral.

Com certeza, eu abordo os tópicos de uma maneira mais ligada à *ciência da computação* do que a maioria dos livros sobre o assunto, portanto o livro pode ser bastante apropriado a uma disciplina de ciência da computação. Especificamente, discuto modelos mentais como árvores e sistemas hierárquicos e procuro enfatizar abordagens funcionais de programação nos locais em que elas façam sentido (embora eu não procure chamar a atenção para isso

durante a narrativa). Se você se vir lecionando em um curso de ciência da computação, poderá optar por focar mais claramente nesses aspectos do material.

Atualmente, não tenho nenhum plano para postar soluções para os problemas práticos (embora isso possa mudar caso eu receba muitas solicitações), portanto você pode se sentir à vontade para usá-los como lições de casa ou como projetos a serem desenvolvidos fora da sala de aula.

Onde buscar ajuda

Como mencionado anteriormente, existe um repositório no GitHub (<http://github.com/semmypurewal/LearningWebAppDev>) com todos os exemplos de código contidos neste livro. Além do mais, você pode dar uma olhada em <http://learningwebappdev.com> para ver se há alguma errata ou outras atualizações à medida que estas forem necessárias.

Também procuro estar bem acessível e ficaria feliz em ajudar você se houver necessidade. Sinta-se à vontade para me tuitar (@semmypurewal) com perguntas/comentários rápidos ou enviar um email a qualquer momento (me@semmy.me) com perguntas mais extensas. Também sugiro que você utilize o recurso “issues” (problemas) de nosso repositório no GitHub para fazer perguntas. Farei o melhor que puder para responder o mais rápido possível.

Comentários gerais sobre o código

Dei o melhor de mim para usar programação idiomática e ser claro sempre que possível. Apesar disso, esses dois objetivos às vezes são conflitantes. Sendo assim, há ocasiões em que não faço as tarefas “da maneira certa” por motivos pedagógicos. Espero que esses pontos sejam evidentes por si só aos desenvolvedores experientes e que eles não provoquem nenhum dano aos desenvolvedores inexperientes no longo prazo.

Todo o código deve funcionar bem nos navegadores web modernos, e eu testei tudo no Chrome. Obviamente, não posso garantir o funcionamento nas versões mais antigas do Internet Explorer. Por favor, me avise se encontrar qualquer problema de compatibilidade com navegadores no Internet Explorer 10+ ou nas versões modernas de qualquer outro navegador.

Na maioria das vezes, eu adotei JavaScript idiomático², porém há alguns lugares em que não segui isso à risca. Por exemplo, preferi usar aspas duplas no lugar de aspas simples para strings, principalmente porque trabalho com a suposição de que os alunos possam ter experiência anterior com Java/C++. Optei por usar aspas em torno dos nomes de propriedades em objetos literais para que o JSON não ficasse muito diferente dos objetos JavaScript. Também uso \$ como primeiro caractere nas variáveis que apontam para objetos jQuery. Acho que isso mantém a clareza e torna o código um pouco mais legível para os iniciantes.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica termos novos, URLs, endereços de email, nomes e extensões de arquivos.

Largura constante

Usada para listagens de programas, assim como dentro de parágrafos, para se referir a elementos de programas, como nomes de variáveis ou de funções, bancos de dados, tipos de dados, variáveis de ambiente, comandos e palavras-chave.

Largura constante em negrito

Mostra comandos ou outro texto que devam ser digitados literalmente pelo usuário.

Largura constante em itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou determinados pelo contexto.

Este elemento significa uma dica ou uma sugestão.



Este elemento significa uma observação geral.



Este elemento significa um aviso ou uma precaução.

Uso de exemplos de código de acordo com a política da O'Reilly

Materiais suplementares (exemplos de código, exercícios etc.) estão disponíveis para download em www.github.com/semmypurewal/LearningWebAppDev.

Este livro está aqui para ajudá-lo a fazer seu trabalho. De modo geral, se este livro incluir exemplos de código, você pode usar o código em seus programas e em sua documentação. Você não precisa nos contatar para pedir permissão, a menos que esteja reproduzindo uma parte significativa do código.

Por exemplo, escrever um programa que use diversas partes de código deste livro não requer permissão. Porém vender ou distribuir um CD-ROM de exemplos de livros da O'Reilly requer permissão. Responder a uma pergunta mencionando este livro e citar o código de exemplo não requer permissão. Em contrapartida, incluir uma quantidade significativa de código de exemplos deste livro na documentação de seu produto requer permissão.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: “*Learning Web App Development* de Semmy Purewal (O'Reilly). Copyright 2014 Semmy Purewal, 978-1-449-37019-0”.

Se você achar que o seu uso dos exemplos de código está além do razoável ou da permissão concedida, sinta-se à vontade em nos contatar em permissions@oreilly.com.

Temos uma página web para este livro, na qual incluimos erratas, exemplos e quaisquer outras informações adicionais. Essa página pode ser acessada em <http://oreil.ly/learning-web-app>.

Como entrar em contato com a Novatec

Envie seus comentários e suas dúvidas sobre este livro para:

novatec@novatec.com.br

Temos uma página web para este livro, na qual incluimos erratas, exemplos e demais informações adicionais.

- Página da edição em português

<http://www.novatec.com.br/catalogo/7522347-aprende-desenv-web>

- Página da edição original em inglês

<http://oreil.ly/learning-web-app>

Para obter mais informações sobre os livros da Novatec, acesse nosso site em:

<http://www.novatec.com.br>

Agradecimentos

Obrigado ao pessoal simpático do departamento de Ciência da Computação da UNC Asheville por ter me permitido lecionar essa disciplina duas vezes. E, é claro, obrigado aos alunos que a cursaram, pela paciência que tiveram comigo e com o material envolvido.

Agradeço à minha editora Meg Blanchette por ter feito o seu melhor para me manter no prazo e – é claro – pela sua constante paciência com os atrasos. Vou sentir falta de nossas trocas semanais de emails!

Obrigado a Simon St. Laurent por ter me oferecido diversos conselhos no início e por ter me ajudado a fazer com que a ideia

fosse aprovada pela O'Reilly.

Sylvan Kavanaugh e Mark Philips fizeram uma leitura cuidadosa de todos os capítulos e ofereceram vários feedbacks muito úteis durante o processo. Emily Watson leu os quatro primeiros capítulos e ofereceu diversas sugestões inteligentes para melhorias. Mike Wilson leu os últimos quatro capítulos e deu conselhos técnicos de valor inestimável. Tenho uma dívida enorme de gratidão com todos vocês e espero poder retribuir-lhes esse favor algum dia.

Bob Benites, Will Blasko, David Brown, Rebekah David, Andrea Fey, Eric Haughee, Bruce Hauman, John Maxwell, Susan Reiser, Ben Rosen e Val Scarlata leram várias revisões do material e ofereceram sugestões úteis. Sou sinceramente grato pelo tempo e pelo esforço dedicados. Vocês são demais!

Apesar de toda a excelência dos revisores e dos amigos que analisaram o material, é praticamente impossível escrever um livro como este sem que haja alguns erros técnicos, erros de digitação e práticas ruins que se infiltram por meio de algumas frestas. Assumo a total responsabilidade por tudo isso.

1 N.T.: Maneiras típicas de realizar determinadas tarefas.

2 N.T.: JavaScript Idiomático é um conjunto de padrões para a escrita de código JavaScript.

CAPÍTULO 1

Fluxo de trabalho

Criar aplicações web é uma tarefa complicada que envolve várias partes móveis e a interação entre componentes. Para aprendermos a fazer isso é necessário separar essas partes em porções que sejam administráveis e tentar entender como elas se encaixam. Surpreendentemente, o fato é que o componente com o qual interagimos com mais frequência nem mesmo envolve código!

Neste capítulo, iremos explorar o fluxo de trabalho do desenvolvimento de aplicações web, que corresponde ao processo usado para desenvolver nossas aplicações. Ao fazer isso, aprenderemos o básico sobre algumas das ferramentas que tornam o processo mais administrável e (principalmente) menos complicado.

Essas ferramentas incluem um editor de texto, um sistema de controle de versões e um navegador web. Não estudaremos nenhum deles em profundidade, porém aprenderemos o suficiente para podermos dar início à programação web do lado do cliente. Na capítulo 2, veremos o fluxo de trabalho realmente em ação à medida que estudarmos o HTML.

Se você tiver familiaridade com essas ferramentas, poderá dar uma olhada no resumo e nos exercícios ao final do capítulo e então seguir adiante.

Editores de texto

A ferramenta com a qual você irá interagir com mais frequência é o seu editor de texto. Essa tecnologia essencial, embora às vezes

menosprezada, é realmente a ferramenta mais importante de sua caixa de ferramentas porque é o programa que você utiliza para interagir com o seu código. Como o seu código compõe os blocos de construção concretos de sua aplicação, é realmente importante que sua criação e modificação sejam de fácil execução. Além disso, normalmente você irá editar vários arquivos simultaneamente, portanto é importante que o seu editor de texto ofereça o recurso de permitir navegar rapidamente pelo seu sistema de arquivos.


No passado, pode ser que você tenha gastado um bom tempo escrevendo artigos ou editando documentos do tipo texto com programas como o Microsoft Word ou o Google Docs. Não é desses tipos de editores de que estou falando. Esses editores focam mais na formatação do texto do que na facilidade de editá-los. O editor de texto que usaremos tem bem poucos recursos que nos permitem formatar o texto, porém apresenta uma abundância de recursos que nos ajudam a manipulá-lo de modo eficiente.

Na outra extremidade do espectro estão os IDEs (Integrated Development Environments, ou Ambientes Integrados de Desenvolvimento), como o Eclipse, o Visual Studio e o XCode. Esses produtos geralmente têm recursos que facilitam a manipulação do código, mas têm também recursos importantes para o desenvolvimento corporativo de softwares. Não teremos a oportunidade de usar nenhum desses recursos neste livro, portanto vamos manter a simplicidade.

Sendo assim, que tipos de editor de texto devemos explorar? Duas classes principais de editores são comumente utilizadas no desenvolvimento de aplicações web modernas. A primeira corresponde à dos editores GUI (Graphical User Interface, ou Interface Gráfica de Usuário). Pelo fato de eu estar supondo que você tem um pouco de conhecimento prévio de programação e de computação, é bem provável que você já tenha tido experiência com um ambiente GUI Desktop. Desse modo, você deve se sentir relativamente à vontade com esses editores. Eles respondem bem

ao mouse como dispositivo de entrada e apresentam menus familiares que permitem interagir com o seu sistema de arquivos, como você faria em qualquer outro programa. Exemplos de editores de texto GUI incluem o TextMate, o Sublime Text e o Coda.

A outra classe de editores de texto é a dos editores do tipo terminal. Esses editores foram criados antes que as GUIs ou os mouses sequer existissem, portanto aprendê-los pode ser desafiador para as pessoas que estão acostumadas a interagir com um computador por meio de uma GUI ou de um mouse. Por outro lado, esses editores podem ser muito mais eficientes se você estiver disposto a investir tempo para aprender a usar um deles. Os editores mais comumente utilizados, que se enquadram nessa categoria, são o Emacs (mostrado na figura 1.1) e o Vim (mostrado na figura 1.2).

The image is a screenshot of the Emacs text editor running in a terminal window. The window title is "semmy - Emacs - 128x40". The editor displays an HTML document with the following content:

```
<!doctype html>
<html>
  <head>
    <title>My First Web App</title>
  </head>
  <body>
    <h1>Hello, World!</h1>

    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
    tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
    veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
    commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
    velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
    occaecat cupidatat non proident, sunt in culpa qui officia deserunt
    mollit anim id est laborum.</p>
  </body>
</html>
```

 At the bottom of the window, a status bar shows: "UU:----F1 example2.html All (1,0) Git-master (HTML) Mon Sep 2 5:18PM 1.18" and "Loading vc-git...done".

Figura 1.1 – Um documento HTML aberto no Emacs.

A screenshot of a Vim editor window. The title bar at the top reads "semmy - vim - 128x40". The editor displays an HTML document with the following content:

```
[doctype html]
<html>
  <head>
    <title>My First Web App</title>
  </head>

  <body>
    <h1>Hello, World!</h1>

    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
    tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
    veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
    commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
    velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
    occaecat cupidatat non proident, sunt in culpa qui officia deserunt
    mollit anim id est laborum.</p>

  </body>
</html>
```

At the bottom of the window, a status line shows the file path and cursor position: "LearningWebApp/code/Chapter2/Examples/example2.html" 19L, 635C.

Figura 1.2 – Um documento HTML aberto no Vim.

Neste livro, focaremos no uso de um editor de texto GUI chamado Sublime Text, porém recomendo a todos que adquiram um pouco de experiência, seja com o Emacs ou com o Vim. Se você continuar em sua jornada de desenvolvimento de aplicações web, é bastante provável que você vá conhecer algum outro desenvolvedor que utilize um desses editores.

Instalando o Sublime Text

O Sublime Text (ou Sublime, para ser mais conciso) é um editor de texto popular com vários recursos que o tornam excelente para o desenvolvimento web. Além do mais, ele tem a vantagem de ser multiplataforma, o que significa que, de modo geral, ele pode funcionar da mesma maneira, independentemente de você estar usando Windows, Linux ou Mac OS. Ele não é gratuito, porém você pode baixar uma cópia gratuita para avaliação e usá-la pelo período de tempo que desejar. Se você gostar do editor e usá-lo bastante, sugiro que compre uma licença.

Para instalar o Sublime, acesse <http://www.sublimetext.com> e clique no link Download na parte superior. Nesse local, você encontrará instaladores para todas as principais plataformas. Apesar de o Sublime Text 3 estar em fase de testes beta (na época desta publicação), sugiro que você tente usá-lo. Eu o utilizei em todos os exemplos e nas imagens de tela deste livro.

Básico sobre o Sublime Text

Depois de ter o Sublime instalado e de executá-lo, você verá uma tela que se parece com a da figura 1.3.

A primeira tarefa a ser feita é criar um arquivo novo. Faça isso acessando o menu File (Arquivo) e clicando em New (Novo). Isso também pode ser feito teclando Ctrl-N no Windows e no Linux ou usando Command-N no Mac OS. Agora digite `Hello World!` no editor. O editor terá um aspecto semelhante ao da figura 1.4.

A aparência do ambiente do Sublime pode ser alterada pelo menu do Sublime Text ao acessar Preferences → Color Scheme (Preferências → Esquema de cores). Experimente usar alguns esquemas diferentes de cores e descubra aquele que for mais confortável aos seus olhos. Provavelmente, é uma boa ideia investir algum tempo explorando as opções de tema porque você passará muito tempo olhando para o seu editor de texto. Observe que também é possível alterar o tamanho da fonte no submenu Font (Fontes) em Preferences (Preferências) para deixar o texto mais legível.

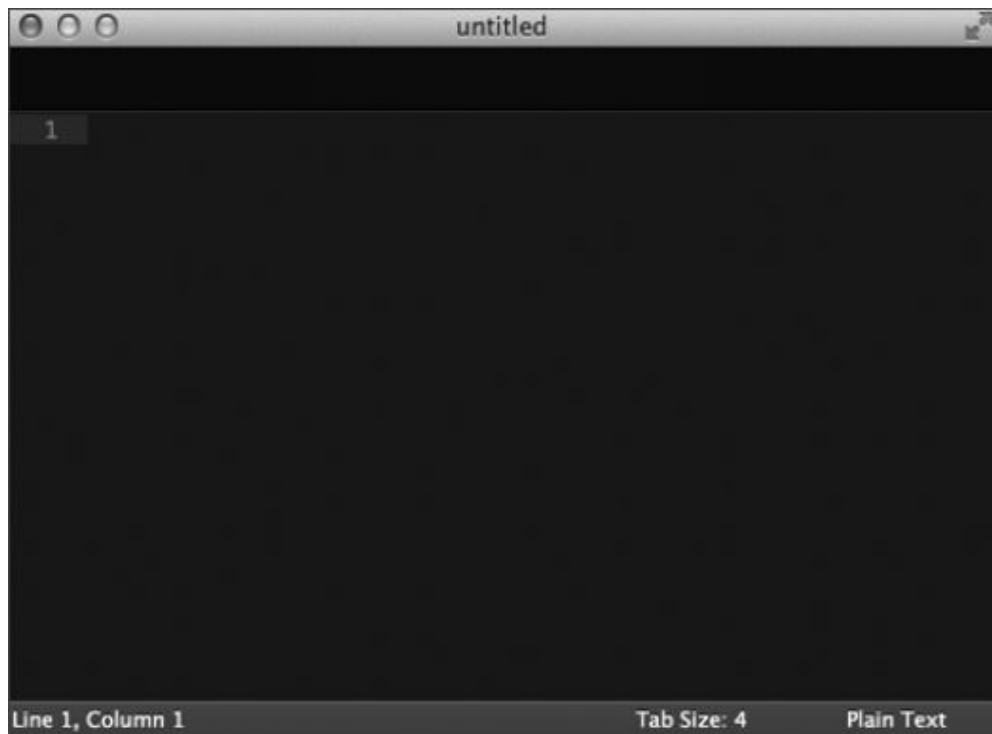


Figura 1.3 – O Sublime Text ao ser aberto pela primeira vez.



Figura 1.4 – O Sublime após um novo arquivo ter sido aberto e Hello World! ter sido digitado no arquivo.

Você deve ter percebido que o Sublime alterou o nome da aba de

“untitled” para “Hello World!” à medida que você digitou esse texto. Ao salvar o arquivo, o nome default será o nome que aparece na aba, porém é provável que você vá querer alterá-lo para que o nome do arquivo não inclua nenhum espaço em branco. Após o arquivo ter sido salvo com um nome diferente, a aba na parte superior será alterada para refletir o verdadeiro nome do arquivo. Observe que, ao fazer alterações subsequentes, você verá o X à direita da aba mudar para um círculo verde – isso significa que você tem alterações que ainda não foram salvas.

Após ter alterado o seu tema e ter salvado o seu arquivo como *hello*, o editor terá a aparência mostrada na figura 1.5.



Figura 1.5 – O Sublime após o tema ter sido alterado para Solarized (light) e o arquivo ter sido salvo como hello.



Como iremos trabalhar a partir da linha de comando, é uma boa ideia evitar espaços ou caracteres especiais nos nomes dos arquivos. Ocasionalmente, salvaremos arquivos usando o caractere underscore (_) em vez do espaço, porém procure não usar nenhum outro caractere que não seja numérico ou alfabético.

Passaremos um bom tempo editando códigos no Sublime, portanto,

obviamente, queremos garantir que nossas alterações estarão sendo salvas regularmente. Como eu espero que todos tenham um pouco de experiência com código, vou supor que você já conhece o processo de editar, salvar e editar. Por outro lado, há um processo essencial associado, com o qual muitos programadores novos não têm experiência, e esse processo se chama controle de versões.

Controle de versões

Suponha que você esteja escrevendo um texto longo de ficção em um processador de texto. Você salva periodicamente o seu trabalho para evitar desastres, mas, de repente, alcança um ponto muito importante no enredo de sua história e percebe que há uma parte significativa do passado de seu protagonista que está faltando. Você decide preencher alguns detalhes lá atrás, próximo ao início de sua história. Então você retorna ao início, porém percebe que há duas possibilidades para o personagem. Como sua história ainda não está totalmente delineada, você decide criar uma versão preliminar de ambas as possibilidades para ver que rumo a história irá tomar. Sendo assim, você copia o seu arquivo em dois lugares e salva um deles como um arquivo chamado *StoryA* e o outro como um arquivo chamado *StoryB*. Você cria a versão preliminar das duas opções para a sua história, cada qual em seu arquivo.

Acredite ou não, isso acontece com os programas de computador com muito mais frequência do que acontece com os romances. Com efeito, à medida que prosseguir, você perceberá que boa parte do tempo de codificação será gasto fazendo algo que se chama *codificação exploratória*. Isso significa que você está simplesmente tentando descobrir o que deve ser feito para que um determinado recurso funcione da maneira que deveria, antes de realmente começar a efetuar a codificação. Às vezes, a fase de codificação exploratória pode dar origem a mudanças que se espalham por várias linhas em diversos arquivos de código de sua aplicação. Mesmo os programadores iniciantes irão perceber isso mais cedo

do que tarde e, com frequência, irão implementar uma solução semelhante àquela que acabou de ser descrita. Por exemplo, os iniciantes podem copiar o diretório de código corrente para outro diretório, fazer uma pequena mudança no nome e prosseguir. Se eles perceberem que cometeram um erro, sempre será possível retornar para a cópia anterior.

Essa é uma abordagem rudimentar para o *controle de versões*. O controle de versões é um processo que permite manter pontos de verificação (checkpoints) nomeados em seu código, de modo que você sempre poderá referenciá-los novamente (ou até mesmo restaurá-los), se for necessário. Além disso, o controle de versões é uma ferramenta essencial para a colaboração com os demais desenvolvedores. Não iremos enfatizar essa ideia com tanta frequência neste livro, mas é uma boa ideia tê-la em mente.

Muitas ferramentas profissionais de controle de versões estão disponíveis e cada qual possui o seu próprio conjunto de recursos e de particularidades. Alguns exemplos comuns incluem o Subversion, o Mercurial, o Perforce e o CVS. Na comunidade de desenvolvimento web, porém, o sistema de controle de versões mais popular chama-se Git.

Instalando o Git

O Git tem instaladores simples, tanto para o Mac OS quanto para o Windows. Para o Windows, usaremos o projeto msysgit, que está disponível no GitHub (<http://msysgit.github.io/>), conforme mostrado na figura 1.6. Os instaladores continuam disponíveis no Google Code e possuem links a partir da página do GitHub. Após ter feito o download do instalador, dê um clique duplo nele e siga as instruções para instalar o Git em seu sistema.

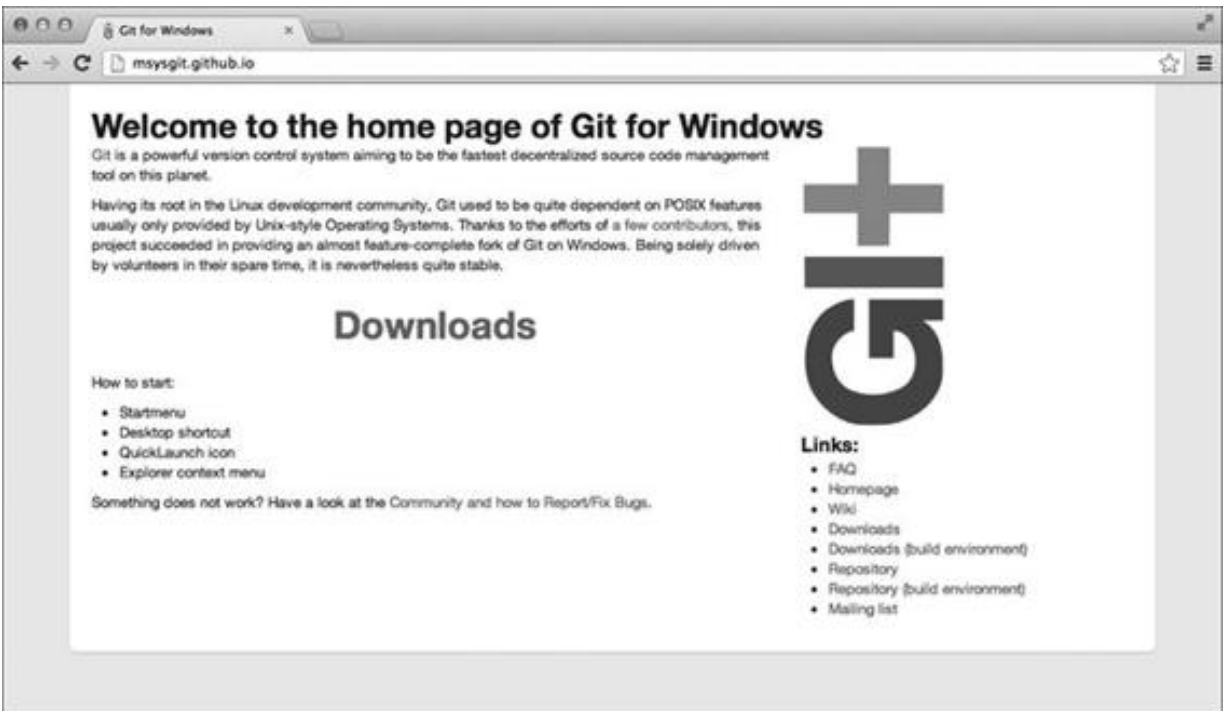


Figura 1.6 – Página inicial do msysgit.

No Mac OS, prefiro usar o instalador do Git para OS X (<http://code.google.com/p/git-osx-installer/>) mostrado na figura 1.7. Basta fazer o download da imagem de disco previamente empacotada, montá-la e, em seguida, dar um clique duplo no instalador. Na época desta publicação, o instalador informava que era destinado ao Mac OS Snow Leopard (10.5), mas, em meu sistema Mountain Lion (10.8), ele funcionou corretamente.



Figura 1.7 – Página inicial do Git para OS X.

Se você estiver usando Linux, o Git pode ser instalado por meio de seu sistema de gerenciamento de pacotes.

Básico sobre a linha de comando Unix

Há interfaces gráficas de usuário para o Git, porém aprender a usá-lo por meio da linha de comando é muito mais eficaz. Antes de aprender a fazer isso, porém, você terá de aprender a navegar pelo seu sistema de arquivos usando alguns comandos Unix básicos.

Conforme já mencionei anteriormente, estou supondo que você tenha conhecimentos anteriores de computação e de programação, portanto é bem provável que você já tenha interagido com um ambiente GUI desktop. Isso significa que você já teve de usar o ambiente desktop para explorar os arquivos e as pastas armazenados em seu computador. Geralmente, isso é feito por meio de um navegador de sistema de arquivos, por exemplo, o Finder no Mac OS ou o Windows Explorer no Windows.

Navegar pelo sistema de arquivos de seu computador a partir da

linha de comando é muito semelhante a navegar por ele usando o navegador de arquivos de seu sistema. Os arquivos continuam existindo e eles estão organizados em pastas, porém nós nos referimos a essas pastas como *diretórios*. Todas as mesmas tarefas podem ser facilmente executadas da mesma maneira que são feitas no navegador de arquivos: você pode entrar ou sair de um diretório, visualizar os arquivos contidos em um diretório e até mesmo abrir e editar arquivos se você tiver familiaridade com o Emacs ou o Vim. A única diferença é que não há um feedback visual contínuo da GUI e você não será capaz de efetuar a interação por meio de um mouse.

Se você estiver no Windows, você fará o que se segue no prompt do Git Bash, instalado com o projeto msysgit descrito na seção anterior. O Git Bash é um programa que simula um terminal Unix no Windows e permite acessar os comandos do Git. Para iniciar o prompt do Git Bash, navegue até ele por meio de seu menu Start (Iniciar). Se o Mac OS estiver sendo executado, utilize o programa Terminal, que pode ser encontrado no diretório *Utilities* (Utilitários) em sua pasta *Applications* (Aplicativos). Se você estiver usando o Linux, a situação dependerá um pouco da versão em particular que estiver sendo usada, porém, geralmente, haverá um programa Terminal facilmente acessível em seus aplicativos. A janela de terminal padrão do Mac OS está sendo mostrada na figura 1.8.

Após abrir o terminal, um prompt de comandos será apresentado a você. A aparência poderá ser diferente conforme você estiver usando Windows ou Mac OS, porém o prompt normalmente conterá algumas informações sobre o seu ambiente de trabalho. Por exemplo, ele pode incluir o seu diretório corrente ou talvez o seu nome de usuário. No Mac OS, o meu prompt tem o seguinte aspecto:

```
Last login: Tue May 14 15:23:59 on ttys002  
hostname $ _
```

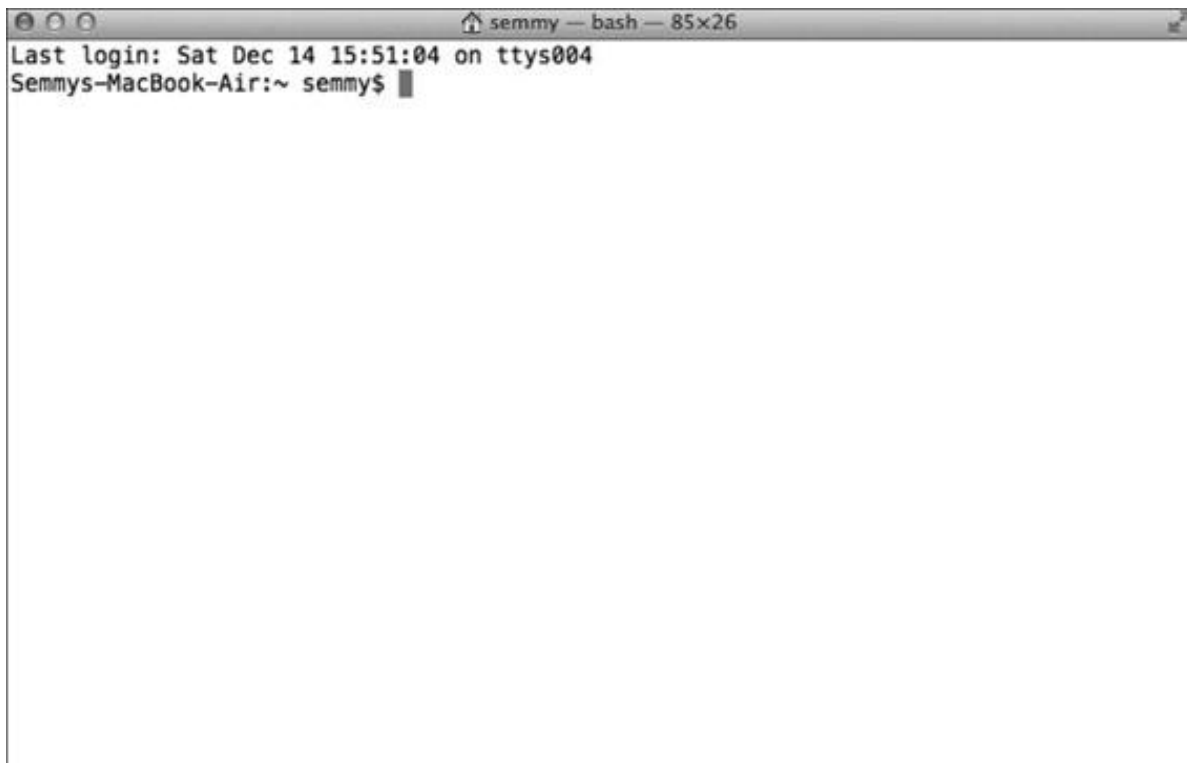


Figura 1.8 – Uma janela de terminal padrão no Mac OS.

Onde estou?

Um aspecto importante a se ter em mente é que, sempre que você estiver no prompt de um terminal, você estará em um diretório. A primeira pergunta que você deve se fazer quando estiver diante de uma interface de linha de comando é: “Em que diretório eu estou?”. Há duas maneiras de responder a essa pergunta a partir da linha de comando. A primeira consiste em usar o comando `pwd`, que quer dizer *print working directory* (mostrar o diretório de trabalho). A saída será algo do tipo:

```
hostname $ pwd
/Users/semmy
```

Embora eu use `pwd` ocasionalmente, com certeza prefiro usar o comando `ls`, que, de modo bem geral, se traduz como *listar o conteúdo do diretório corrente*. Isso me dá mais pistas visuais a respeito do local em que estou. No Mac OS, o resultado de um `ls` tem o seguinte aspecto:

```
hostname $ ls
```

Desktop Downloads Movies Pictures
Documents Library Music

Portanto `ls` é semelhante a abrir um Finder ou uma janela do Explorer em sua pasta home. O resultado desse comando indica que estou em meu diretório home porque vejo todos os seus subdiretórios sendo apresentados na tela. Se eu não reconhecer os subdiretórios contidos no diretório, usarei `pwd` para obter mais informações.

Mudando de diretório

A próxima tarefa que você deve aprender é navegar para um diretório diferente daquele em que você estiver no momento. Se você estiver na GUI de um navegador de arquivos, isso pode ser feito simplesmente por meio de um clique duplo no diretório corrente.

Na linha de comando, não é mais difícil; basta que você se lembre do nome do comando. É `cd`, que quer dizer *change directory* (mudar de diretório). Portanto, se você quiser ir para a sua pasta *Documents*, basta digitar:

```
hostname $ cd Documents
```

E se você quiser obter algum feedback visual a respeito do diretório em que estiver, use `ls`:

```
hostname $ ls  
Projects
```

Esse comando informa que há um subdiretório em seu diretório *Documents*, e que esse subdiretório se chama *Projects*. Observe que um diretório *Projects* pode não existir em seu diretório *Documents*, a menos que você tenha criado um anteriormente. Pode ser também que você vá ver outros arquivos ou diretórios listados, caso tenha usado o seu diretório *Documents* para armazenar outros itens no passado. Agora que você mudou de diretório, a execução de `pwd` informará qual é a sua nova localização:

```
hostname $ pwd  
/Users/semmy/Documents
```

O que acontecerá se você quiser voltar para o seu diretório home? Na GUI de seu navegador de arquivos, há geralmente um botão de retorno que permite mover-se para um novo diretório. No terminal, um botão desse tipo não existe. Porém você pode continuar a usar o comando `cd` com uma pequena alteração: use dois pontos-finais (`..`) no lugar de um nome de diretório para retroceder um diretório:

```
hostname $ cd ..
hostname $ pwd
/Users/semmy
hostname $ ls
Desktop Downloads Movies Pictures
Documents Library Music
```

Criando diretórios

Por fim, você deve aprender a criar um diretório para armazenar todos os seus projetos deste livro. Para isso, utilize o comando `mkdir`, que quer dizer *make directory* (criar diretório):

```
hostname $ ls
Desktop Downloads Movies Pictures
Documents Library Music
hostname $ mkdir Projects
hostname $ ls
Desktop Downloads Movies Pictures
Documents Library Music Projects
hostname $ cd Projects
hostname $ ls
hostname $ pwd
/Users/semmy/Projects
```

Nessa interação com o terminal, inicialmente dê uma olhada no conteúdo de seu diretório home para garantir que você sabe em que local você está usando o comando `ls`. Depois disso, utilize `mkdir` para criar o diretório *Projects*. Então use `ls` para confirmar se o diretório foi criado. A seguir, utilize `cd` para entrar no diretório *Projects* e, então, `ls` para listar o conteúdo. Observe que o diretório está vazio no momento, portanto `ls` não apresenta nenhuma saída. Por fim, use `pwd` para confirmar se você está realmente no diretório *Projects*.

Esses quatro comandos Unix básicos são suficientes para começar, porém você aprenderá outros à medida que prosseguirmos. Incluí uma tabela prática no final deste capítulo que descreve e sintetiza esses comandos. É uma boa ideia tentar memorizá-los.

Sistemas de arquivo e árvores

O desenvolvimento web (e a programação em geral) é uma forma de arte bastante abstrata. Falando de modo genérico, isso significa que, para fazer isso de forma eficaz e eficiente, você deve aperfeiçoar suas habilidades de raciocínio abstrato. Uma boa parte de ter um raciocínio abstrato está associada à capacidade de relacionar modelos mentais a novas ideias e estruturas rapidamente. E um dos melhores modelos mentais que pode ser aplicado a uma ampla gama de situações é o diagrama de árvore.

Um diagrama de árvore é simplesmente uma maneira de visualizar qualquer tipo de estrutura hierárquica. E, como o sistema de arquivos do Unix é uma estrutura hierárquica, começar a praticar nossas visualizações mentais com esse sistema é uma boa ideia. Por exemplo, considere um diretório chamado *Home* que contenha três outros diretórios: *Documents*, *Pictures* e *Music*. Há cinco imagens no diretório *Pictures*. No diretório *Documents*, há outro diretório chamado *Projects*.

Um diagrama de árvore para essa estrutura é semelhante ao que está sendo mostrado na figura 1.9.

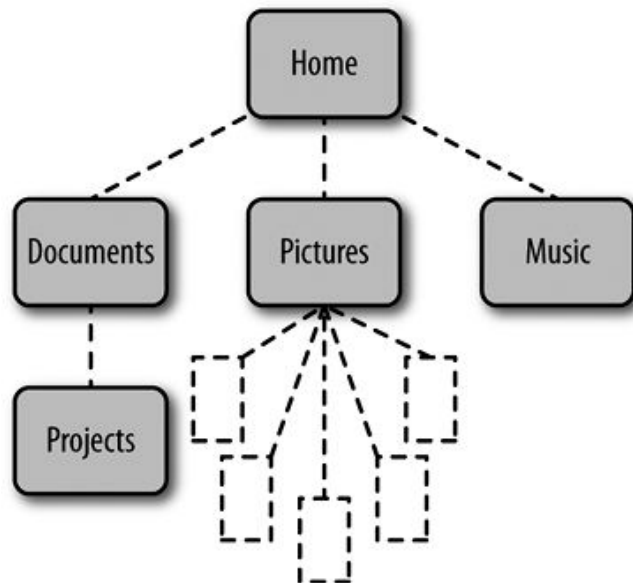


Figura 1.9 – Um diagrama de árvore que representa uma hierarquia de arquivos.

Manter esse modelo mental em sua mente enquanto você estiver navegando pelo sistema de arquivos é uma boa ideia. Com efeito, eu recomendo adicionar um asterisco (ou algo semelhante) que indique qual é o seu diretório corrente e fazê-lo se mover à medida que você navegar pelo sistema de arquivos.

Falando de modo geral, se você tentar associar um diagrama de árvore a qualquer estrutura hierárquica, é bem provável que você achará mais fácil entendê-la e analisá-la. Como boa parte de ser um programador eficiente resulta de sua habilidade de criar modelos mentais rapidamente, é uma boa ideia praticar a associação desses diagramas de árvore a sistemas hierárquicos do mundo real sempre que essa associação fizer sentido. Faremos isso com alguns exemplos ao longo do restante do livro.

Básico sobre o Git

Agora que podemos navegar pela linha de comando, estamos prontos para aprender a manter o nosso projeto sob um controle de versões usando o Git.

Configurando o Git pela primeira vez

Como mencionei anteriormente, o Git, na realidade, foi concebido para proporcionar uma colaboração em larga escala entre diversos programadores. Embora o Git vá ser usado para nossos projetos pessoais, ele deverá ser configurado para que possamos manter um controle de nossas alterações usando algumas informações de identificação, especificamente o nosso nome e o nosso endereço de email. Abra o seu terminal e digite os comandos a seguir (mudando o meu nome e o meu endereço de email pelo seu, é claro):

```
hostname $ git config --global user.name "Semmy Purewal"
hostname $ git config --global user.email "semmy@semmy.me"
```

É necessário fazer isso somente uma única vez em nosso sistema! Em outras palavras, não é preciso executar esses comandos sempre que quisermos criar um projeto que vamos monitorar com o Git.

Agora estamos prontos para começar a controlar um projeto usando o Git. Iniciaremos navegando até a nossa pasta *Projects* caso ainda não estejamos nela:

```
hostname $ pwd
/Users/semmy
hostname $ cd Projects
hostname $ pwd
/Users/semmy/Projects
```

A seguir, criaremos um diretório chamado *Chapter1* e listaremos o seu conteúdo para confirmar que ele existe. Em seguida, entraremos no diretório:

```
hostname $ mkdir Chapter1
hostname $ ls
Chapter1
hostname $ cd Chapter1
hostname $ pwd
/Users/semmy/Projects/Chapter1
```

Inicializando um repositório Git

Agora podemos submeter o diretório *Chapter1* ao controle de versões ao inicializar um repositório Git com o comando `git init`. O Git

responderá nos informando que criou um repositório vazio:

```
hostname $ pwd
/Users/semmy/Projects/Chapter1
hostname $ git init
Initialized empty Git repository in /Users/semmy/Projects/Chapter1/.git/
```

Agora tente digitar o comando `ls` novamente para ver os arquivos criados pelo Git no diretório e você perceberá que ainda não há nada lá! Isso não é totalmente verdade – o diretório `.git` está presente, porém não podemos vê-lo porque arquivos prefixados com um ponto (`.`) são considerados ocultos. Para resolver isso, podemos usar `ls` com a flag `-a` (all) habilitada, digitando o seguinte:

```
hostname $ ls -a
. .. .git
```

Esse comando lista todo o conteúdo do diretório, inclusive os arquivos prefixados com um ponto. Você verá até mesmo o diretório corrente listado (que corresponde a um único ponto) e o diretório pai (que corresponde aos dois pontos).

Se você tiver interesse, liste o conteúdo do diretório `.git` e você verá o sistema de arquivos que o Git preparou para você:

```
hostname $ ls .git
HEAD config hooks objects
branches description info refs
```

Não teremos a oportunidade de fazer nada nesse diretório, portanto podemos ignorá-lo por enquanto, sem grandes preocupações. Contudo teremos a oportunidade de interagir novamente com arquivos ocultos, portanto lembrar-se da flag `-a` do comando `ls` será útil.

Determinando o status de nosso repositório

Vamos abrir o Sublime Text (se ele ainda estiver aberto por causa da seção anterior, feche-o e abra-o novamente). A seguir, abra o diretório que submetemos ao controle de versões. Para isso, basta selecionar o diretório na caixa de diálogo Open (Abrir) do Sublime, em vez de selecionar um arquivo específico. Ao abrir um diretório

inteiro, um painel de navegação de arquivos será aberto à esquerda da janela do editor – a aparência deverá ser semelhante à da figura 1.10.

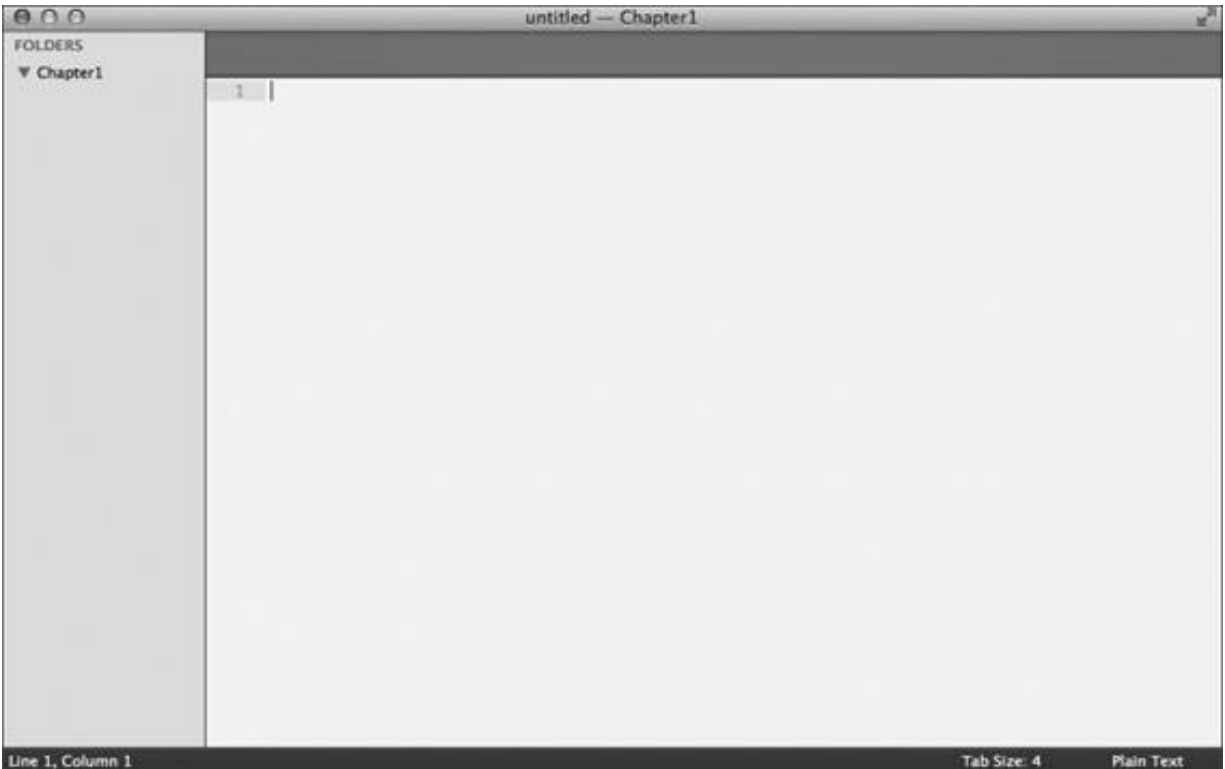


Figura 1.10 – O Sublime com o diretório Chapter1 aberto.

Para criar um arquivo novo no diretório *Chapter1*, dê um clique com o botão da direita do mouse (ou use Command-click no Mac OS) em *Chapter1* no painel de navegação de arquivos e selecione New File (Arquivo Novo) no menu de contexto. Isso fará um novo arquivo ser aberto como antes, porém, ao salvá-lo, o diretório *Chapter1* será utilizado por padrão. Vamos salvar esse arquivo como *index.html*.

Depois de nomeá-lo, dê um clique duplo no arquivo e adicione a linha “Hello World!” na parte superior do arquivo, como mostrado na figura 1.11.



Figura 1.11 – O Sublime depois que o arquivo index.html foi adicionado, editado e salvo.

Vamos ver o que aconteceu com o nosso repositório Git. Volte para a sua janela do terminal e verifique se você está no diretório correto:

```
hostname $ pwd
/Users/semmy/Projects/Chapter1
hostname $ ls
index.html
```

Agora digite `git status` e você verá uma resposta que se parecerá com:

```
hostname $ git status
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
# index.html
```

Há muitas informações aqui. Estamos mais interessados na seção

chamada `Untracked files`. Esses são os arquivos que estão em nosso diretório de trabalho, mas que não estão sob o controle de versões no momento.

Observe que nosso arquivo *index.html* está lá, pronto para que façamos o seu commit em nosso repositório Git.

Nossos primeiros commits!

Estamos interessados em controlar as alterações de nosso arquivo *index.html*. Para isso, devemos seguir as instruções fornecidas pelo Git e adicionar o arquivo ao repositório usando o comando `git add`:

```
hostname $ git add index.html
```

Observe que o Git não dá nenhuma resposta. Isso não é um problema. Podemos verificar se tudo correu bem digitando `git status` novamente:

```
hostname $ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file:   index.html
#
```

Isso nos dá o feedback que estávamos procurando. Note que *index.html* agora está listado embaixo do título `Changes to be committed` (Alterações aguardando commit).

Depois de termos adicionado os novos arquivos ao repositório, queremos fazer o commit do estado inicial do repositório. Para isso, usaremos o comando `git commit`, juntamente com a flag `-m` e uma mensagem significativa sobre o que foi alterado desde o último commit. Com frequência, o nosso commit inicial terá o seguinte aspecto:

```
hostname $ git commit -m "Initial commit"
[master (root-commit) 147deb5] Initial commit
```

```
1 file changed, 1 insertion(+)
create mode 100644 index.html
```

Esse comando cria um snapshot de nosso projeto no tempo. Sempre podemos retornar a esse estado posteriormente caso algo dê errado ao longo do caminho. Se digitarmos `git status` agora, veremos que *index.html* não aparece mais porque está sendo controlado e nenhuma alteração foi feita. Se não houver nenhuma alteração desde o último commit, é sinal de que temos um “diretório de trabalho limpo”:

```
hostname $ git status
# On branch master
nothing to commit (working directory clean)
```



É fácil esquecer-se de incluir o `-m` e uma mensagem ao efetuar o commit. Entretanto, se isso acontecer, é bem provável que você se veja no editor de texto Vim (que, normalmente, é o editor de texto default do sistema). Se isso acontecer, você pode sair do editor digitando dois-pontos (:) e em seguida `q!` e teclando Enter.

A seguir, vamos modificar *index.html* fazendo uma pequena alteração. Adicionaremos uma segunda linha contendo “Goodbye World!”. Vá em frente: faça isso e salve o arquivo usando o atalho de teclado apropriado. Agora vamos ver como o `git status` responde a essa mudança:

```
hostname $ git status
# On branch master
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified: index.html
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Observe que o Git nos informa que *index.html* foi modificado, porém não está preparado (não está no stage) para o próximo commit. Para adicionar nossas modificações ao repositório, inicialmente devemos executar `git add` no arquivo modificado e, em seguida, efetuar o `git commit` de nossas alterações. Podemos verificar se a

adição ocorreu corretamente digitando `git status` antes do commit. Essa interação se parecerá com algo como:

```
hostname $ git add index.html
hostname $ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# modified:   index.html
#
hostname $ git commit -m "Add second line to index.html"
[master 1c808e2] Add second line to index.html
1 file changed, 1 insertion(+)
```

Visualizando o histórico de nosso repositório

Até agora fizemos dois commits em nosso projeto e podemos retornar a esses snapshots a qualquer momento. Em “Práticas e leituras adicionais”, faço uma referência a um link que mostrará como fazer uma reversão para um commit anterior e iniciar a codificação a partir desse ponto. Mas, por enquanto, há outro comando que pode ser útil. Podemos dar uma olhada em nosso histórico de commits usando `git log`:

```
hostname $ git log
commit 1c808e2752d824d815929cb7c170a04267416c04
Author: Semmy Purewal <semmy@semmy.me>
Date: Thu May 23 10:36:47 2013 -0400

    Add second line to index.html

commit 147deb5dbb3c935525f351a1154b35cb5b2af824
Author: Semmy Purewal <semmy@semmy.me>
Date: Thu May 23 10:35:43 2013 -0400

    Initial commit
```

Assim como os quatro comandos Unix que aprendemos na seção anterior, é realmente importante memorizar esses quatro comandos Git. Uma tabela prática em “Resumo” inclui esses comandos.

Salvar versus efetuar commit

Caso esteja confuso, gostaria de reservar um momento para estabelecer claramente a diferença entre salvar um arquivo (por meio de seu editor de texto) e efetuar o commit de uma alteração. Ao salvar um arquivo, ele será sobrescrito no disco de seu computador. Isso significa que, a menos que o seu editor de texto disponibilize algum tipo de histórico de revisões, não será mais possível acessar a versão antiga de seu arquivo.

Efetuar um commit em um repositório Git permite manter o controle de todas as alterações feitas desde a última vez que o commit do arquivo foi efetuado. Isso quer dizer que sempre é possível retornar a uma versão anterior de um arquivo se você descobrir que cometeu um erro irreversível no estado atual de seu arquivo.

A essa altura, provavelmente pode parecer que o Git armazena o seu código como uma sequência linear de commits. Isso faz sentido agora porque você aprendeu um subconjunto de comandos do Git que permite criar um repositório em que cada commit segue exatamente outro commit. Nós nos referimos ao primeiro commit como o commit *pai* e ao segundo como o commit *filho*. Um repositório Git com quatro commits tem o aspecto semelhante àquele mostrado na figura 1.12.

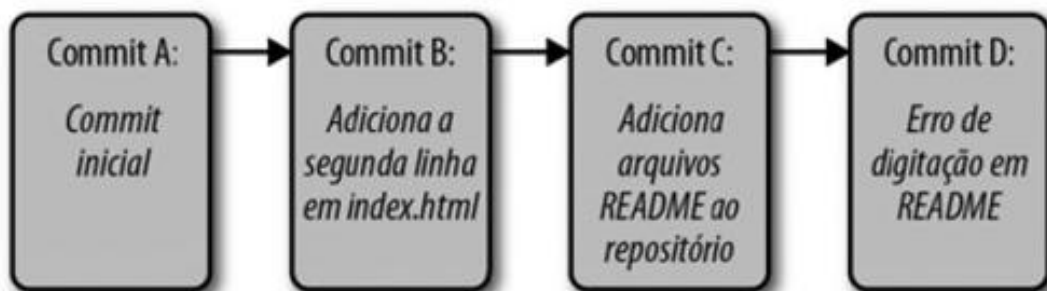


Figura 1.12 – Um repositório Git com quatro commits.

Vale a pena notar, porém, que um commit corresponde a uma série de instruções para conduzir o seu projeto à próxima versão. Em outras palavras, um commit do Git não armazena realmente todo o conteúdo de seu repositório da maneira como seria feito caso você fizesse uma cópia de um diretório para outro. Em vez disso, o Git

armazena somente o que deve ser alterado: por exemplo, um commit pode armazenar informações como “adicione uma linha com *Goodbye World*” em vez de armazenar o arquivo completo. Portanto é melhor pensar no repositório Git como uma sequência de instruções. É por isso que escrevemos nossas mensagens de commit no tempo *imperativo presente* – você pode pensar em um commit como uma série de instruções que leva o seu projeto de um estado para o próximo.

Por que tudo isso é importante? Na realidade, um repositório Git pode ter uma estrutura muito mais complexa. Um commit pode ter mais de um filho e – com efeito – mais de um pai. A figura 1.13 mostra um exemplo de um repositório Git mais complexo, em que ambos os fatos mencionados anteriormente são verdadeiros.

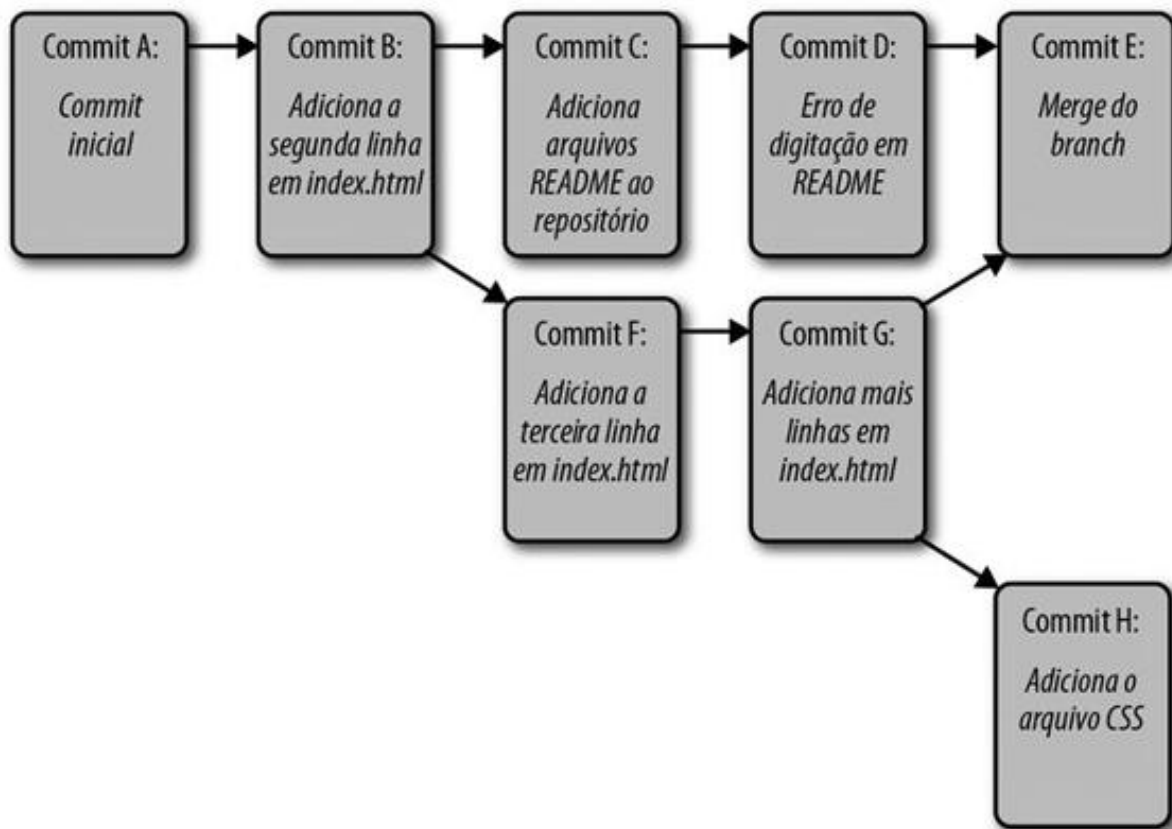


Figura 1.13 – Um repositório Git mais complexo.

Neste momento, não conhecemos nenhum comando Git que nos permita criar uma estrutura como essa, porém, se você prosseguir

em sua jornada para o desenvolvimento de aplicações web, em algum momento será necessário aprender esses comandos. A questão é que isso deve motivar você a começar a pensar em seu repositório Git de modo mais visual, de modo que, quando a situação *realmente* ficar complicada, você não se sinta desesperado.

Navegadores

A última ferramenta com a qual iremos interagir regularmente é o navegador web. Como estamos aprendendo a criar aplicações que serão executadas em navegadores web, é essencial aprendermos a usar de forma eficaz o nosso navegador como ferramenta de desenvolvimento, e não apenas como uma janela para a Internet.

Existem vários navegadores web excelentes, incluindo o Firefox, o Safari e o Chrome. Recomendo que você se torne proficiente no uso das ferramentas para desenvolvedores disponíveis em todos esses navegadores. Porém, para manter todos sincronizados e deixar tudo mais simples, usaremos o Google Chrome como nossa opção de navegador.

Instalando o Chrome

Não importa se você esteja no Windows, no Mac OS ou no Linux, o Google Chrome poderá ser facilmente instalado ao acessar a sua página web (<http://www.google.com/chrome>). O processo de instalação irá variar, é claro, porém as instruções são bem claras. Após ter instalado o Chrome e tê-lo executado pela primeira vez, ele deverá ter um aspecto semelhante ao da figura 1.14.



Figura 1.14 – Janela default do Chrome.

Resumo

Um dos aspectos mais importantes no desenvolvimento de aplicações web é acostumar-se com um fluxo de trabalho eficiente e eficaz. Um fluxo de trabalho moderno envolve três ferramentas importantes: um editor de texto, um sistema de controle de versões e um navegador web. O Sublime Text é um editor de texto popular, multiplataforma e útil para a edição de códigos-fonte. O Git é um sistema de controle de versões comumente utilizado que possui uma interface de linha de comando. O Chrome é um excelente navegador web para o desenvolvimento web.

Antes de prosseguirmos, você deve ter todas as ferramentas anteriormente descritas instaladas em seu computador. Memorize também os comandos das tabelas 1.1 e 1.2, que permitem navegar pelo seu sistema de arquivos e interagir com o Git a partir da linha de comando.

Tabela 1.1 – Comandos Unix

--	--

Comando	Descrição
pwd	Mostra o seu diretório corrente.
ls	Lista o conteúdo de seu diretório corrente.
ls -a	Faz a listagem, incluindo todos os arquivos ocultos.
cd [dir]	Muda para o diretório chamado [dir].
mkdir [dir]	Cria um novo diretório chamado [dir].

Tabela 1.2 – Comandos do Git

Comando	Descrição
git init	Inicializa o seu repositório.
git status	Mostra o status de seu repositório.
git add [arquivos(s)]	Prepara [arquivos] (coloca no stage) para o próximo commit.
git commit -m [msg]	Faz o commit dos arquivos que estão no stage com a mensagem [msg].
git log	Mostra o histórico de commits.

Práticas e leituras adicionais

Memorização

Ao ensinar e aprender, com frequência, a *memorização* tem uma conotação negativa. Em minha opinião, esse ponto de vista, na maioria das vezes, é equivocado, particularmente quando relacionado à programação de computadores. Se você assumir a mentalidade do tipo “bem, posso simplesmente consultar quando precisar”, você gastará mais tempo pesquisando sobre assuntos básicos do que focando nos aspectos mais desafiadores que surgirem. Pense, por exemplo, em como seria muito mais difícil efetuar divisões longas se você não tivesse memorizado as tabelas de tabuada!

Com isso em mente, vou incluir uma seção de “Memorização” no final dos primeiros capítulos, a qual englobará os itens básicos que você deve memorizar antes de prosseguir para o próximo capítulo. Para este capítulo, esses itens estão todos relacionados à linha de comando do Git e do Unix. Realize essas tarefas repetidamente até poder executá-las sem consultar qualquer documentação:

1. Crie uma pasta nova usando a linha de comando.
2. Entre nessa pasta usando a linha de comando.
3. Crie um arquivo texto em seu editor de texto e salve-o como *index.html* no novo diretório.
4. Inicialize um repositório Git a partir da linha de comando.
5. Adicione e faça o commit desse arquivo no repositório a partir da linha de comando.

Qual é a melhor maneira de memorizar essa sequência de tarefas? É simples: realize-as repetidamente. Vou acrescentar novos itens a essas tarefas ao longo dos próximos capítulos, portanto é importante dominar esses passos agora.

Sublime Text

Conforme mencionado anteriormente, você gastará bastante tempo em seu editor de texto, portanto ir um pouco além do básico provavelmente é uma boa ideia. O site do Sublime contém uma ótima página de suporte (<http://www.sublimetext.com/support>) com links para documentações e vídeos que demonstram os recursos avançados do editor. Sugiro que você explore a página e veja se você consegue aperfeiçoar suas habilidades com o Sublime.

Emacs e Vim

Quase todo desenvolvedor web, em algum momento, terá de editar um arquivo em um servidor remoto. Isso significa que você não poderá usar um editor de texto que exija uma GUI. O Emacs e o Vim

são editores incrivelmente eficazes que facilitam bastante a realização dessa tarefa, porém a curva de aprendizado para ambos é relativamente íngreme. Se você tiver tempo, realmente vale a pena aprender o básico sobre ambos os editores, porém me parece que o Vim tornou-se mais comum entre os desenvolvedores web nos últimos anos (confesso: sou um usuário do Emacs).

A página inicial do GNU contém uma visão geral excelente do Emacs, incluindo um tutorial para iniciantes (<http://www.gnu.org/software/emacs/tour>). A O'Reilly também tem diversos livros sobre o Emacs e o Vim, incluindo: *Learning the vi and Vim Editors* (<http://shop.oreilly.com/product/9780596529833.do>) de Arnold Robbins, Elbert Hannah e Linda Lamb, e *Learning GNU Emacs* (<http://shop.oreilly.com/product/9780596006488.do>) de Debra Cameron, James Elliott, Marc Loy, Eric S. Raymond e Bill Rosenblatt.

Aprender a realizar as seguintes tarefas em ambos os editores será vantajoso para você:

1. Abrir e sair do editor.
2. Abrir, editar e salvar um arquivo.
3. Abrir vários arquivos simultaneamente.
4. Criar um arquivo novo de dentro do editor e salvá-lo.
5. Pesquisar um arquivo em busca de uma dada palavra ou expressão.
6. Recortar e colar porções de texto entre dois arquivos.

Se investir tempo nessas tarefas, você terá uma boa noção do editor com o qual você irá preferir passar mais tempo.

Linha de comando Unix

É necessário muito tempo para dominar a linha de comando Unix, mas você já aprendeu o suficiente para começar. De acordo com a minha experiência, é muito melhor aprender os comandos no

contexto da solução de problemas específicos, porém há alguns outros comandos básicos que eu uso regularmente. Por meio de pesquisa no Google, aprenda a respeito de alguns desses comandos comuns: `cp`, `mv`, `rm`, `rmdir`, `cat` e `less`. Todos eles serão muito úteis em diversas ocasiões.

Mais a respeito do Git

O Git é uma ferramenta incrivelmente eficiente – mal tocamos a superfície de seus recursos. Felizmente, Scott Chacon escreveu *Pro Git* (<http://git-scm.com/book>, Apress, 2009), um ótimo livro que descreve vários aspectos do Git com muitos detalhes. Os dois primeiros capítulos descrevem vários recursos que ajudarão você a acompanhar este livro de maneira mais eficiente, incluindo efetuar a reversão para versões cujos commits foram previamente efetuados em seu repositório.

O terceiro capítulo do livro de Chacon aborda o conceito de branching em detalhes. O branching está um pouco além do escopo deste livro, porém já o mencionei rapidamente antes. Sugiro que você explore esse assunto porque a habilidade de criar um branch de forma fácil e rápida em seu repositório é realmente um dos melhores recursos do Git.

GitHub

O GitHub é um serviço online que irá hospedar os seus repositórios Git. Se você mantiver o seu código aberto, ele será gratuito. Se quiser criar repositórios Git privados, o plano mais barato do GitHub custa cerca de 7 dólares ao mês. Sugiro que você se cadastre em um plano gratuito e explore a maneira de armazenar repositórios Git no GitHub.

A página de ajuda do GitHub (<http://help.github.com/>) descreve a criação de uma conta GitHub e como conectá-la ao seu repositório Git. A página contém também inúmeras informações úteis tanto sobre o Git quanto sobre o GitHub em geral. Use-a como ponto de

partida.

CAPÍTULO 2

Estrutura

Ao longo dos dois próximos capítulos, vamos apresentar uma visão geral de dois tópicos relativamente importantes do lado do cliente: o HTML e o CSS. Como não há nenhuma maneira de podermos discutir ambos em detalhes, esses dois capítulos serão compostos, em sua maior parte, de uma série de tutoriais práticos que ajudarão você a aprender HTML e CSS o suficiente para compreender os exemplos de código presentes no restante do livro. A seção “Práticas e leituras adicionais” incentivará você a explorar outros recursos.

Se já tiver familiaridade com o HTML e com o CSS, você poderá prosseguir sem grandes preocupações para o capítulo 4, que começa com o JavaScript do lado do cliente. Se quiser, você pode dar uma olhada rápida nos capítulos e ler o resumo que está no final antes de fazer isso.

Hello, HTML!

O HTML, que quer dizer HyperText Markup Language (Linguagem de Marcação de Hipertexto), é uma tecnologia que nos permite especificar a estrutura dos elementos visuais (às vezes, chamada de *interface de usuário*) de uma aplicação web. O que quero dizer quando digo estrutura? Vamos dar uma olhada em um exemplo simples.

Para começar, usaremos a linha de comando para criar um diretório chamado *Chapter2* em nosso diretório *Projects*. Lembre-se de que usaremos o comando `mkdir` para isso. A seguir, vamos abrir esse

diretório no Sublime Text usando o menu File (Arquivo) ou as teclas de atalho. Crie um arquivo novo chamado *hello.html* nesse diretório. Digite o conteúdo exatamente da maneira como você está vendo a seguir:

```
<!doctype html>
<html>
  <head>
    <title>My First Web App</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

Tags versus conteúdo

À medida que digitar, um dos pontos que você deve perceber é que o documento é constituído de dois tipos de conteúdo. Um tipo é formado por texto normal, como “My First Web App” e “Hello, World!”. O outro tipo de conteúdo, como `<html>` e `<head>`, é cercado por sinais de menor e de maior, e nós nos referimos a esses elementos como *tags*. As tags são uma espécie de *metadados*, e esses metadados são usados para aplicar uma estrutura ao conteúdo da página.

Inicie o Chrome e abra o arquivo em seu navegador web usando a opção Open File (Abrir arquivo) do menu File (Arquivo). Você verá algo semelhante ao que está sendo apresentado na figura 2.1.



Dominar o uso de atalhos de teclado é uma boa ideia, pois isso tornará o seu fluxo de trabalho mais eficiente. O atalho de teclado para abrir um arquivo no Chrome é **Command-O** se você estiver no Mac OS. No Linux ou no Windows, é **Ctrl-O**.

Observe que as tags não aparecem, porém os demais conteúdos sim. O conteúdo “My First Web App” aparece como o título da aba, enquanto o conteúdo “Hello, World” aparece no corpo da janela.

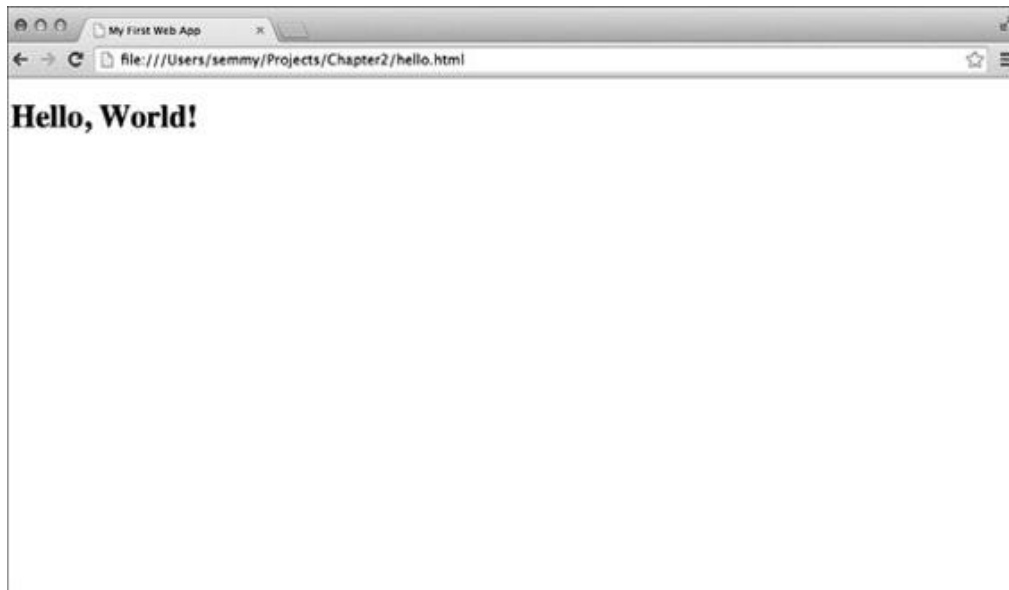


Figura 2.1 – hello.html aberto no Chrome.

<p> de parágrafo

Agora faremos uma pequena modificação adicionando um parágrafo com uma porção de texto *lorem ipsum*, que é somente um texto de preenchimento que pode ser substituído pela cópia verdadeira posteriormente. O texto pode ser recortado e colado da página da Wikipedia que contém o lorem ipsum (http://en.wikipedia.org/wiki/Lorem_ipsum):

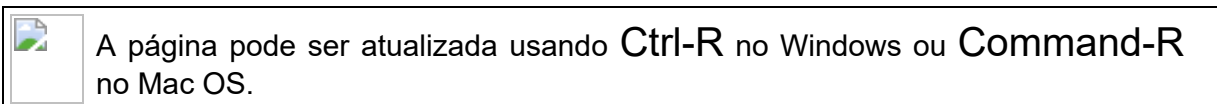
```
<!doctype html>
<html>
  <head>
    <title>My First Web App</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
      eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
      ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
      aliquip ex ea commodo consequat. Duis aute irure dolor in
      reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
      pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
      culpa qui officia deserunt mollit anim id est laborum.</p>
  </body>
```

</html>

Após termos feito as alterações, podemos salvá-las no documento. Agora podemos retornar ao navegador e recarregar a página clicando na seta circular ao lado da barra de endereço do Chrome. Você deverá ver o corpo do navegador atualizar-se com o conteúdo, conforme mostrado na figura 2.2.



Figura 2.2 – example1.html modificado, aberto no Chrome.



A página pode ser atualizada usando **Ctrl-R** no Windows ou **Command-R** no Mac OS.

Esse será o nosso fluxo de trabalho normal ao editar páginas web. Abriremos o arquivo em nosso editor de texto, faremos pequenas alterações e atualizaremos o navegador web para verificar as mudanças.

Comentários

Os comentários são uma maneira conveniente de fazer anotações em nosso HTML. Um comentário HTML é iniciado com `<!--` e finalizado com `-->`. Aqui está um exemplo simples implementado a

partir do exemplo da seção anterior:

```
<!doctype html>
<html>
  <head>
    <title>Comment Example</title>
  </head>
  <body>
    <!-- Este é o cabeçalho principal -->
    <h1>Hello World!</h1>

    <!-- Este é o parágrafo principal -->
    <p>I'm a main paragraph, most likely associated with the h1 tag since
      I'm so close to it!</p>
  </body>
</html>
```

Como os programas de computador são criados para serem lidos por seres humanos, é sempre uma boa ideia fazer anotações nos códigos mais complicados. Você verá exemplos de comentários HTML espalhados pelo livro, e é provável que se depare com HTML comentado na web.

Cabeçalhos e âncoras e listas, oh, céus!

Agora que já vimos alguns exemplos de tags básicas e de comentários, que outros tipos de tags nós podemos incluir em nossa marcação?

Em primeiro lugar, podemos generalizar a tag `<h1>` criando as tags `<h2>`, `<h3>`, `<h4>`, `<h5>` e `<h6>`. Elas representam diferentes níveis de cabeçalho e normalmente são reservadas para conteúdos importantes da página. O conteúdo do cabeçalho *mais* importante deve estar contido em uma tag `<h1>`, enquanto conteúdos de cabeçalhos menos importantes devem aparecer nas demais tags:

```
<!doctype html>
<html>
  <head>
    <title>Heading Tag Examples</title>
  </head>
```

```

<body>
  <!-- Este é o cabeçalho principal -->
  <h1>This is very important!</h1>

  <!--
    Aqui está um conteúdo que pode estar associado à
    parte importante
  -->
  <p>Important paragraph</p>

  <h2>This is a less important header</h2>
  <p>And here is some less important content</p>

</body>
</html>

```

Outra tag importante em um documento HTML é a tag `<a>`, que quer dizer *anchor* (âncora) e é usada para criar links. As tags de âncora representam uma característica única do *hipertexto* porque elas podem efetuar o link com outras informações, seja na página corrente ou em outra página web totalmente diferente. Para usar as tags de âncora, também devemos incluir um *atributo* HTML `href`, que informa ao navegador o local para onde ele deve ir quando um link for clicado. O atributo `href` deve ser inserido dentro da tag de abertura:

```

<!doctype html>
<html>
  <head>
    <title>Link Examples</title>
  </head>

  <body>
    <!--
      O atributo href nos informa para onde ir quando o elemento âncora
      for clicado.
    -->
    <p>Here is a <a href="http://www.google.com">link</a> to Google!</p>
    <p>
      <a href="http://www.example.com">
        And this is a link that is a little longer
      </a>
    </p>

```

```
<p>
  And here is a link to
  <a href="http://www.facebook.com">www.facebook.com</a>
</p>
</body>
</html>
```

Ao abrirmos essa página no navegador web, teremos algo semelhante ao que está sendo mostrado na figura 2.3.

Todo o texto sublinhado na página pode ser clicado, e, quando isso ocorrer, você será conduzido à página especificada no atributo href.

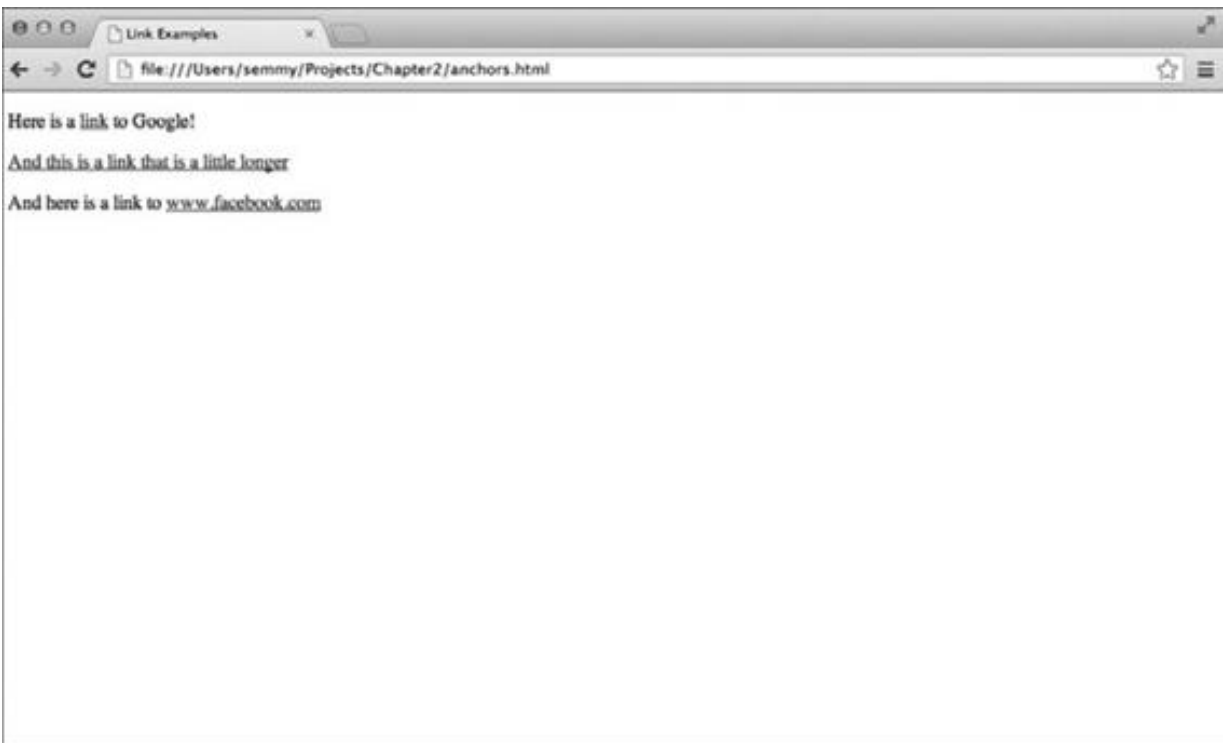


Figura 2.3 – Uma página com links que usam tags de âncora.

Um problema com esse exemplo é que ele utiliza elementos do tipo parágrafo para listar conteúdos. Não seria melhor se tivéssemos uma tag específica que representasse uma lista? O fato é que temos duas delas! A tag `` e a tag `` representam *listas ordenadas* e *listas não ordenadas*, respectivamente. Nessas listas, temos tags `` que representam *itens de lista*. No exemplo anterior, não parece que a ordem dos links seja muito importante, portanto uma lista não ordenada será melhor:


```

<!doctype html>
<html>
  <head>
    <title>List Examples</title>
  </head>
  <body>
    <h1>List Examples!</h1>

    <!-- Encapsularemos os links em uma tag ul -->
    <ul>
      <li>
        Here is a <a href="http://www.google.com">link</a> to Google!
      </li>
      <li>
        <a href="http://www.example.com">
          And this is a link that is a little longer
        </a>
      </li>
      <li>
        And here is a link to
        <a href="http://www.facebook.com">
          www.facebook.com
        </a>
      </li>
    </ul>

    <!-- Também podemos criar uma tag para lista ordenada -->
    <h3>How to make an ordered list</h3>
    <ol>
      <li>Start by opening your ol tag</li>
      <li>Then add several list items in li tags</li>
      <li>Close your ol tag</li>
    </ol>
  </body>
</html>

```

Ao atualizarmos o nosso navegador, sua aparência deverá ser semelhante à da figura 2.4.

Observe como ambas as listas possuem marcadores na frente de cada item da lista, porém, no caso da lista ordenada, esses marcadores correspondem a um valor numérico.

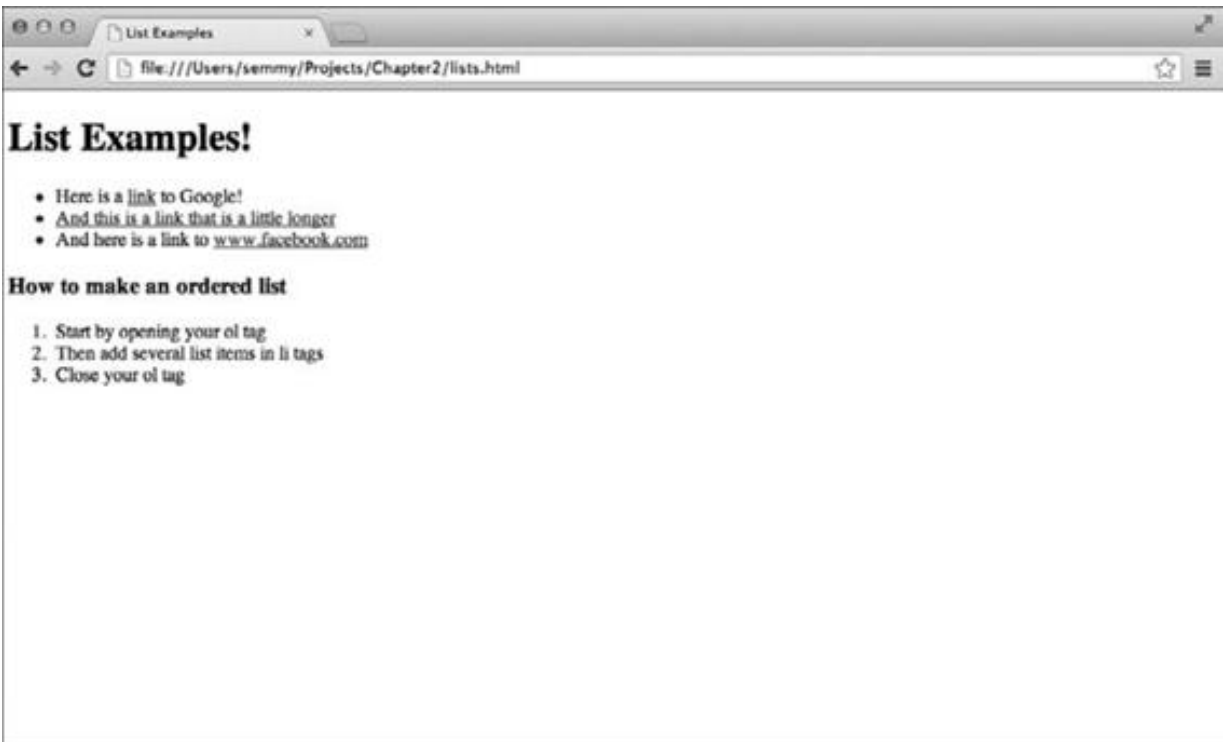


Figura 2.4 – Uma página contendo uma lista não ordenada e uma lista ordenada.

Generalizações

Podemos efetuar algumas generalizações a partir dos primeiros exemplos que já vimos. A primeira generalização refere-se ao fato de que todo conteúdo de texto normal está cercado por tags HTML.

Em segundo lugar, você provavelmente deve ter notado que fizemos a indentação das tags HTML contidas em outras tags HTML. Isso foi feito porque o HTML é um método hierárquico para estruturar documentos. Usamos a indentação como um indício visual para nos lembrarmos em que ponto da hierarquia nós estamos. É por isso que a tag `<head>` e a tag `<body>` estão indentadas dentro da tag `<html>`, e a tag `<h1>` e as tags `<p>` também estão indentadas em relação à tag `<body>`. Em alguns casos, mantivemos os links na mesma linha que o conteúdo, enquanto em outros, separamos a linha. No HTML, espaços em branco não são importantes na maioria das vezes.

Por fim, você verá que, pelo fato de estarmos criando um

documento HTML, iremos adicionar ou modificar uma pequena quantidade de conteúdo, salvar o que desenvolvermos e então iremos para a janela do navegador para recarregar a página. Como isso será feito com muita frequência, exercitar esse processo algumas vezes é uma boa ideia. Para começar, adicione outros parágrafos de texto *lorem ipsum* ao corpo do documento e vá até o navegador para recarregar a página.



É interessante aprender a usar os atalhos de teclado para recarregar a página e alternar entre as janelas ativas de seu ambiente, pois você fará isso com muita frequência. No Windows e na maioria dos ambientes Linux, você pode usar **Ctrl-Tab** para alternar entre as janelas ativas e **Ctrl-R** para recarregar a página. No Mac OS, utilize **Command-Tab** e **Command-R**.

Document Object Model e as árvores

As tags HTML definem uma estrutura hierárquica chamada *Document Object Model* (Modelo de Objeto de Documento), ou DOM, para ser mais conciso. O DOM é uma maneira de representar objetos que podem ser definidos por meio de HTML e com os quais é possível interagir, posteriormente, por meio de uma linguagem de scripting como o JavaScript. As tags HTML definem *elementos* do DOM que são entidades existentes no DOM.

Já estamos escrevendo o nosso HTML de maneira a nos ajudar a visualizar o DOM. É por isso que estamos indentando nossas tags contidas em outras tags, pois isso nos dá uma noção de hierarquia. Embora seja útil para o nosso código, isso nem sempre funciona tão claramente como seria de esperar. Por exemplo, considere o HTML a seguir:

```
<!doctype html>
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello World!</h1>
```

```
<div>
  <ol>
    <li>List Item</li>
    <li>List Item</li>
    <li>List Item</li>
  </ol>

  <p>This is a paragraph.</p>
  <p>This is a <span>second</span> paragraph.</p>
</div>

<ul>
  <li>List Item <span>1</span></li>
  <li>List Item <span>2</span></li>
  <li>List Item <span>3</span></li>
</ul>
</body>
</html>
```

Esse código inclui diversas tags que ainda não vimos, porém não é essencial que você entenda as suas funções por enquanto. O que importa é que você perceba que, apesar de esse código estar claramente indentado, existem algumas tags que não estão separadas em linhas diferentes. Por exemplo, os elementos `span` estão contidos na mesma linha que os elementos `li`. Isso não é um problema porque a tag `` contém um único caractere, mas ele não demonstra o relacionamento de forma tão clara quanto a estrutura indentada o faz. Portanto precisamos de outra maneira para pensar nesse exemplo.

No capítulo anterior, discutimos o uso de diagramas de árvore para criar modelos mentais do sistema de arquivos de nosso computador. O fato é que também podemos usar diagramas de árvore para criar modelos mentais do DOM. Esse modelo mental será prático no futuro, quando interagirmos com o DOM por meio de JavaScript. Como exemplo, podemos usar um diagrama de árvore para representar o código anterior, conforme mostrado na figura 2.5.

Observe que esse diagrama cria uma clara representação do conteúdo do DOM. Essa representação também deixa claro alguns

dos relacionamentos: nós nos referimos aos elementos do DOM nas partes mais baixas da árvore como *descendentes* dos elementos do DOM que estão nas partes mais altas da árvore se houver um caminho que os conecte. Os descendentes imediatos são chamados de *elementos filhos*, enquanto o elemento acima de um elemento filho é chamado de *elemento pai*.

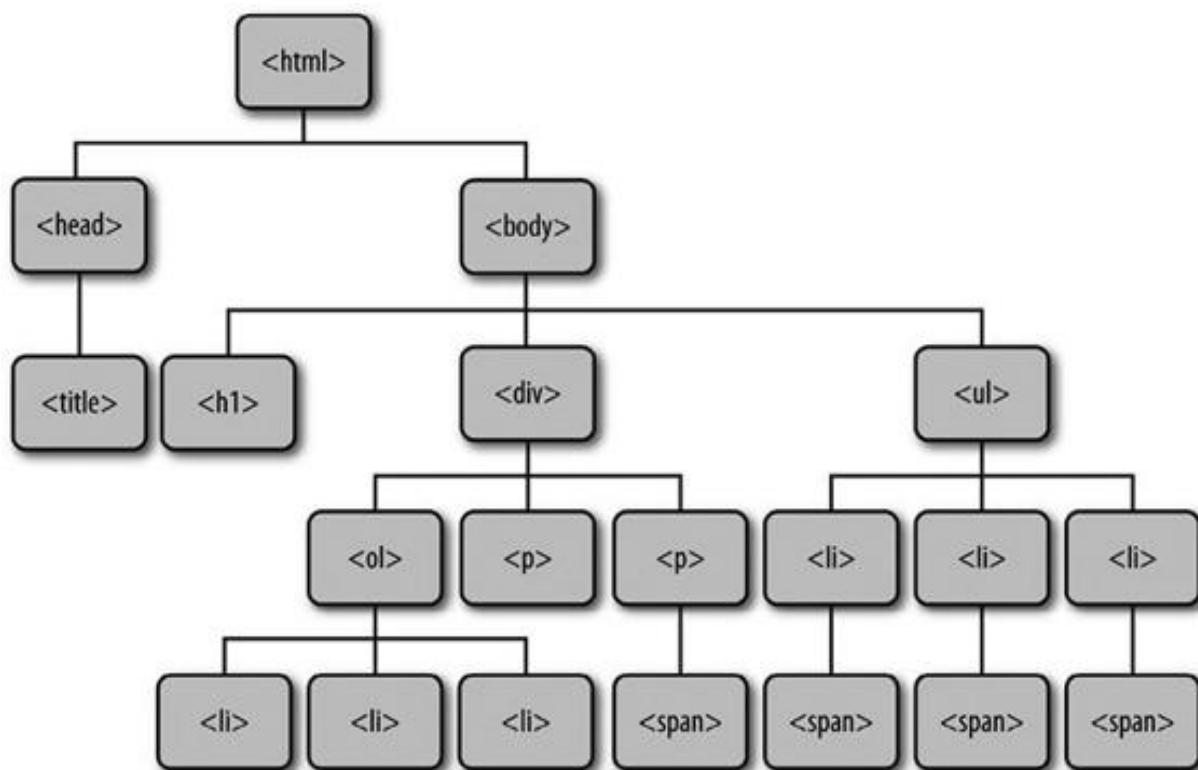


Figura 2.5 – Uma representação em árvore do DOM anterior.

Nesse exemplo, todos os elementos são descendentes dos elementos `html`, e o elemento `ul` é descendente do elemento `body`. O elemento `ul` não é descendente do elemento `head` porque não há nenhum caminho que comece no elemento `head` e termine no elemento `ul` sem que seja necessário subirmos um nível na hierarquia. O elemento `ul` possui três filhos (cada um dos elementos `li`), e cada elemento `li` possui um elemento `span` como filho.

Aprenderemos mais sobre esses relacionamentos à medida que seguirmos adiante, porém, por enquanto, é bom adquirir um pouco de prática para pensar no DOM dessa maneira.

Usando a validação de HTML para identificar problemas

Como mencionei na seção anterior, o conteúdo textual de nosso documento HTML normalmente está cercado por um par de tags. Uma tag de abertura é algo do tipo `<html>` e uma tag de fechamento é algo como `</html>`. O nome propriamente dito da tag é o que especifica o tipo de elemento do DOM que ela representa.

Isso pode causar problemas se o nosso documento se tornar muito longo. Por exemplo, considere o documento HTML a seguir, que corresponde a uma breve generalização de nosso exemplo anterior, com algumas tags novas.

```
<!doctype html>
<html>
  <head>
    <title>My First Web App</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
    <nav>
      <div>Login</div>
      <div>FAQ</div>
      <div>About Us</div>
    </nav>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
      eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
      ad minim veniam, <span>quis nostrud exercitation</span> ullamco
      laboris nisi ut aliquip ex ea commodo consequat.</p>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
      eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
      <span>ad minim veniam, quis nostrud exercitation ullamco laboris nisi
      ut aliquip ex ea commodo consequat. Duis aute irure dolor in
      reprehenderit in voluptate <span>velit esse cillum dolore eu
      fugiat</span> nulla pariatur.</p>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
      eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
  </body>
```

</html>

Esse documento HTML contém um erro, mas se esse documento for aberto em seu navegador, você não irá percebê-lo. Passe alguns instantes observando e veja se você consegue identificá-lo.

Se você o encontrou, parabéns! Você tem uma ótima visão! Caso contrário, não se sinta mal. O erro está no segundo parágrafo. Há uma tag `` de abertura dentro da segunda sentença, porém ela não foi fechada. A maioria das pessoas tem dificuldade para encontrar erros desse tipo quando estão começando a trabalhar com o HTML. Felizmente, há uma maneira muito boa e automatizada de encontrar erros em documentos HTML.

Um programa de *validação* é um programa que verifica automaticamente se o seu código está de acordo com determinados padrões básicos. Se você já usou linguagens de programação como o Java ou o C++ no passado, você deve ter trabalhado com um compilador. Se o seu código tiver um erro, o compilador informará você a esse respeito quando o código for compilado. Linguagens como o HTML são um pouco mais flexíveis no sentido de que um navegador deixará passar pequenas quantidades de erros. Um programa de validação irá identificar esses erros, mesmo quando um navegador não o fizer.

No entanto, se o navegador apresenta a página exatamente da mesma maneira, com ou sem uma tag `` de fechamento, por que devemos nos importar? O fato é que a única maneira de podermos garantir que a página terá *sempre* a mesma aparência em todos os navegadores é se mantivermos o nosso HTML correto. É por isso que uma ferramenta de validação de HTML é muito prática.

Não é necessário instalar nenhum software para usar uma ferramenta de validação de HTML. Por enquanto, vamos começar acessando a página inicial do Markup Validation Service do W3C (<http://validator.w3.org/>). Na época desta publicação, a página tinha um aspecto semelhante ao da figura 2.6.

Observe que há uma aba que diz “Validate by Direct Input” (Validar

por meio de inserção direta). Ao clicarmos nessa aba, podemos recortar e colar parte de nosso código HTML para o campo de texto que será apresentado. Após termos colado algum código, devemos clicar no botão Check (Verificar) grande. Começaremos submetendo o nosso exemplo anterior que contém o *lorem ipsum*. Se o nosso HTML não tiver nenhum erro, veremos algo semelhante ao que está sendo mostrado na figura 2.7.

Se o código não passar pela validação, veremos erros específicos sendo listados. Por exemplo, se submetermos o código que não tem a tag de fechamento para o elemento `span` no segundo parágrafo, veremos algo semelhante ao que está sendo mostrado na figura 2.8.

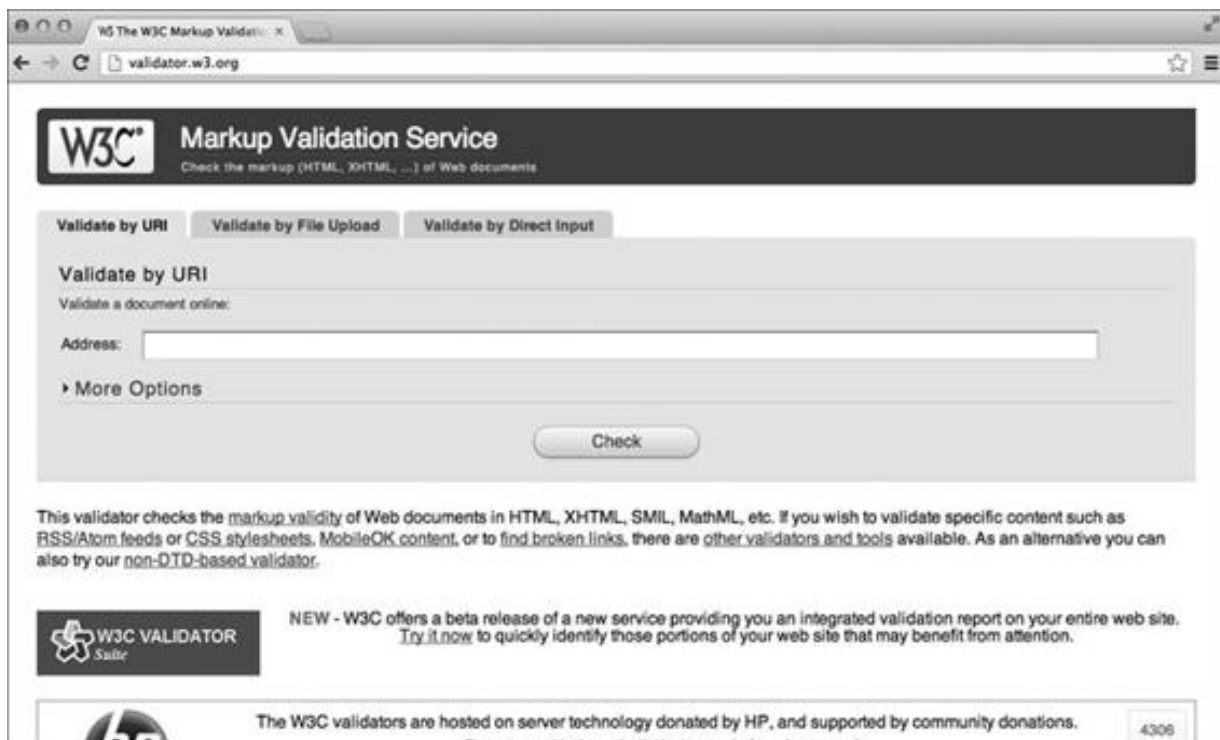


Figura 2.6 – Página inicial do Markup Validation Service do W3C.



Figura 2.7 – A ferramenta de validação de HTML do W3C após ter sido executada em nosso exemplo com lorem ipsum.

Se fizermos uma pequena rolagem para baixo, poderemos ver os erros especificamente listados. A ferramenta de validação não é suficientemente inteligente para nos dizer exatamente em que local está o problema, porém, se entendermos o nosso código o bastante, devemos ser capazes de localizá-lo. A figura 2.9 mostra a descrição de nossos erros, feita pela ferramenta de validação.



Ao validar o seu HTML, é bem provável que você vá obter três warnings (avisos), mesmo quando não houver nenhum erro. O primeiro desses warnings informa que a ferramenta de validação está utilizando o verificador de conformidade com o HTML5. Apesar de o padrão HTML5 ser relativamente estável, pode haver mudanças, e esse warning serve somente para avisar você sobre isso.

Os outros dois warnings referem-se à codificação de caracteres e, por enquanto, podem ser ignorados. Se você estiver interessado, um dos warnings possui um link para um breve tutorial sobre codificação de caracteres, que mostra como especificar uma codificação de caracteres em seu documento HTML.

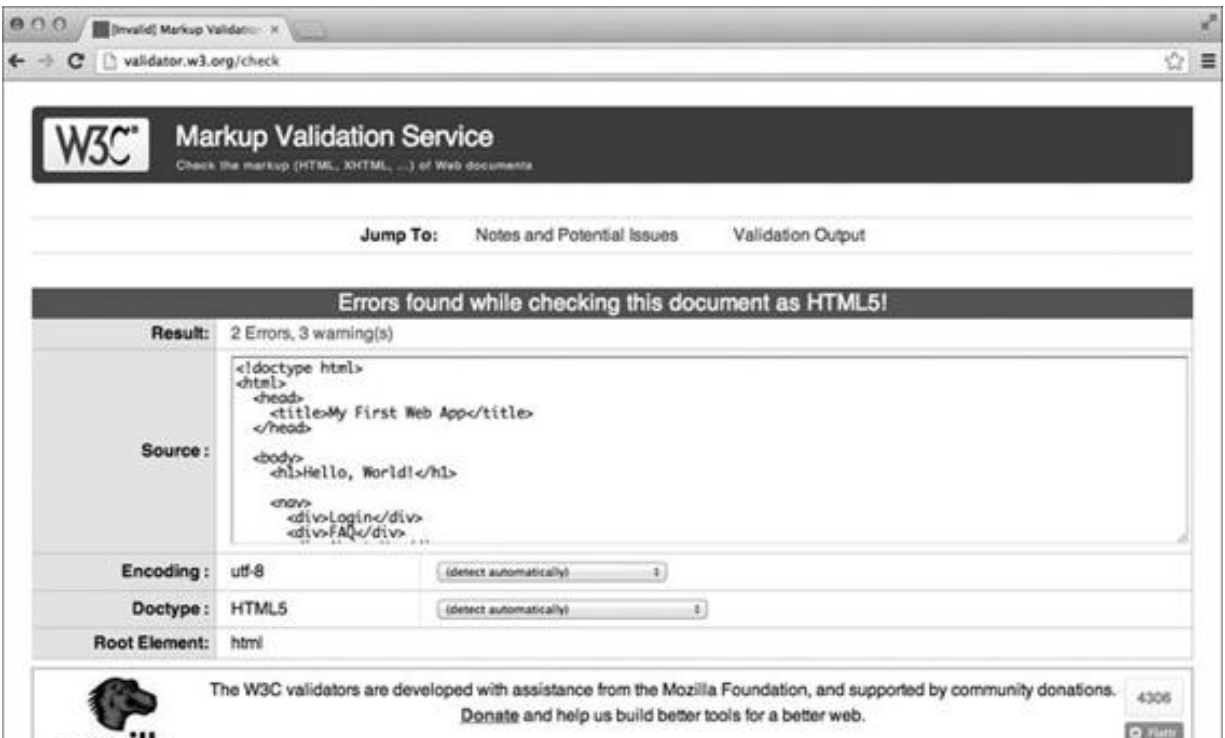


Figura 2.8 – A ferramenta de validação de HTML do W3C, após ter sido executada em nosso exemplo que contém um erro.

É sempre uma boa ideia submeter periodicamente o nosso código HTML ao programa de validação para ver se ele está correto. Ao longo do restante deste capítulo, vou pedir a você que verifique periodicamente o seu HTML junto à ferramenta de validação, e você deve realmente fazer isso.

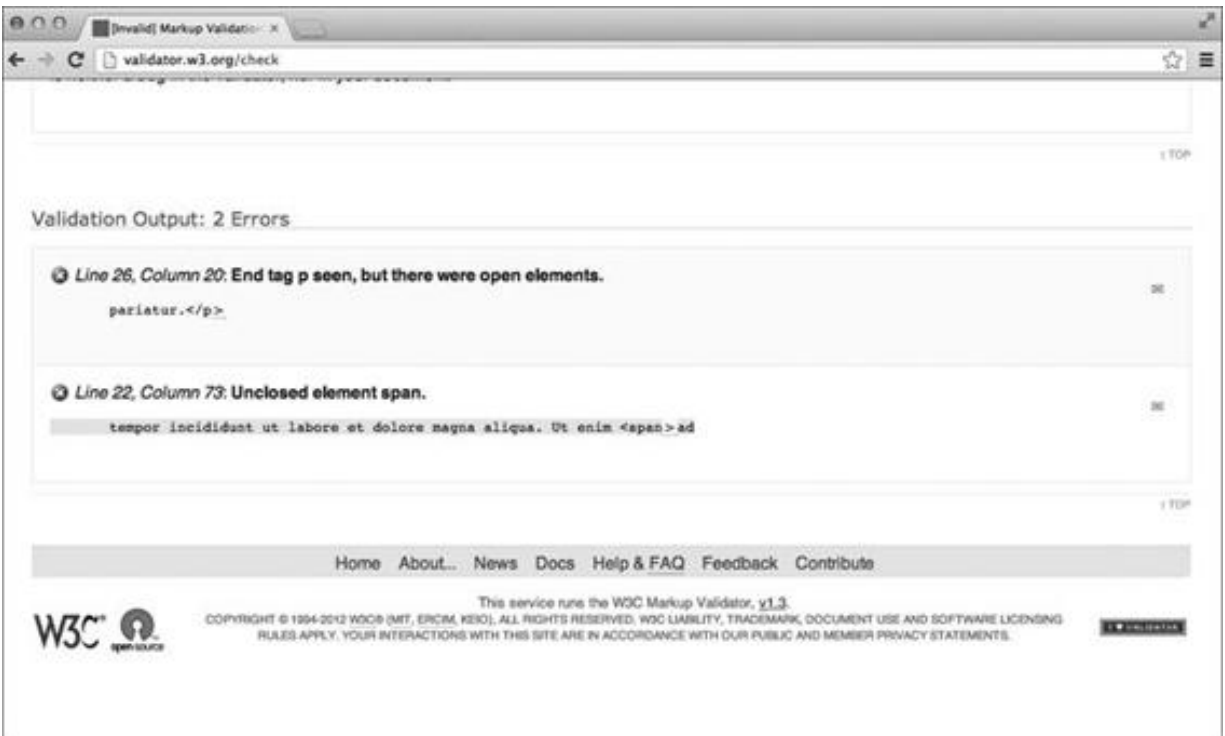


Figura 2.9 – A ferramenta de validação de HTML do W3C mostrando nossos erros.

Aplicação web Amazeriffic

No restante desta seção, criaremos uma página de abertura (splash page) HTML para uma aplicação web fictícia. Temos alguns objetivos ao fazer isso. Em primeiro lugar, exercitaremos o fluxo de trabalho que aprendemos no capítulo anterior em um projeto de verdade. Em segundo lugar, aprenderemos sobre mais algumas tags HTML importantes e o que elas representam.

Identificando a estrutura

Nossa aplicação web fictícia chama-se Amazeriffic, que é uma combinação das palavras *amazing* (maravilhoso) e *terrific* (incrível) e, apesar de ser simplório, não é menos ridículo do que muitos dos nomes de empresas que surgem no Vale do Silício hoje em dia. Como produto, o Amazeriffic controla e classifica um conjunto de tarefas (você pode pensar nele como um organizador de listas de

atividades por fazer). Mais adiante no livro, trabalharemos realmente na implementação de um projeto como esse, mas, por enquanto, focaremos na página inicial do produto até termos um domínio sobre o HTML. A página que criaremos terá uma aparência semelhante à da figura 2.10.

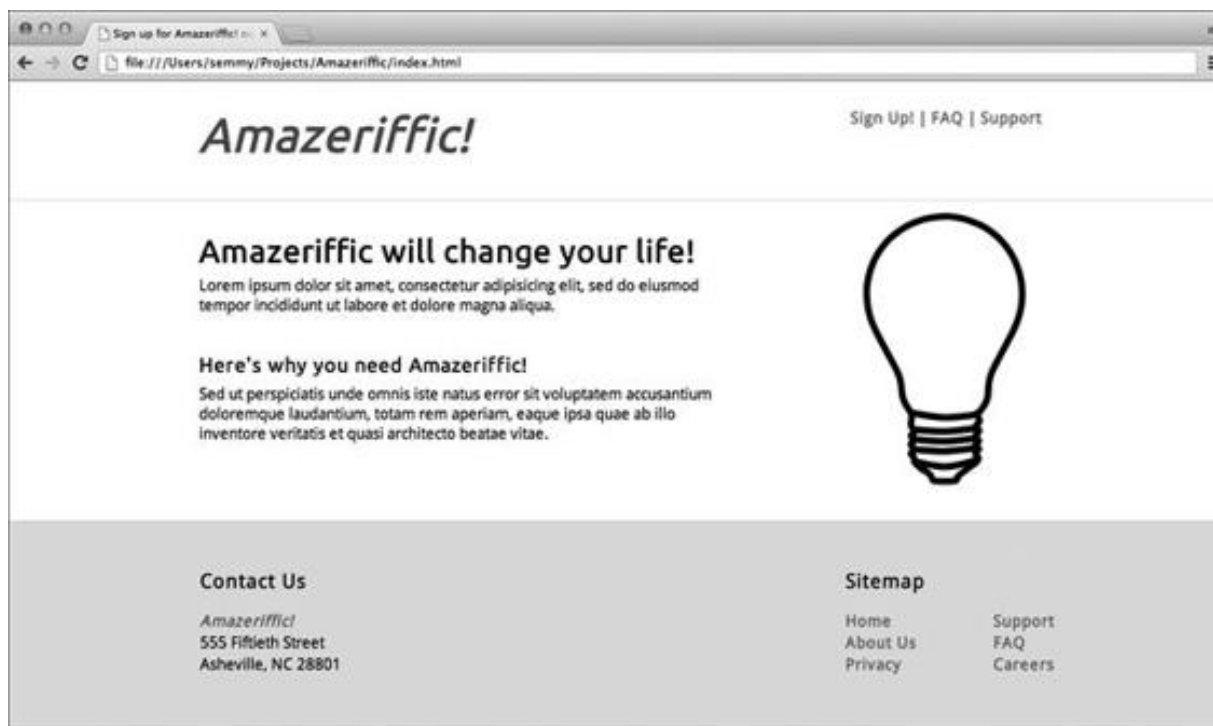


Figura 2.10 – A página de abertura do Amazeriffic, que implementaremos neste e no próximo capítulo.

Lembre-se de que o HTML tem tudo a ver com a *estrutura* de um documento. Isso significa que, apesar de vermos inúmeros elementos *estilísticos* aqui (por exemplo, as diversas fontes, as cores e até o layout), é melhor ignorá-los por enquanto porque, na maioria das vezes, eles não exercerão influência no HTML. Por ora, focaremos exclusivamente na estrutura do documento.

Antes de sequer sentarmos para iniciar o código, é interessante verificar se podemos identificar as diversas partes da estrutura. Usando um lápis e um papel de rascunho, faça um esboço desse layout e dê nomes aos elementos estruturais da melhor maneira que puder. Se você não tiver a mínima ideia do que eu estou falando,

desenhe caixas ao redor dos elementos maiores e dos elementos menores da página e dê-lhes um nome representativo que descreva a sua função no documento.

Na figura 2.11, podemos ver uma versão do mockup (esboço) anterior contendo anotações, com linhas tracejadas ao redor dos elementos. Podemos perceber facilmente que alguns elementos estão contidos em outros elementos. Isso cria um relacionamento que especifica quais elementos serão descendentes de outros elementos no DOM, e isso nos ajuda, de modo geral, a perceber como deverá ser o HTML.

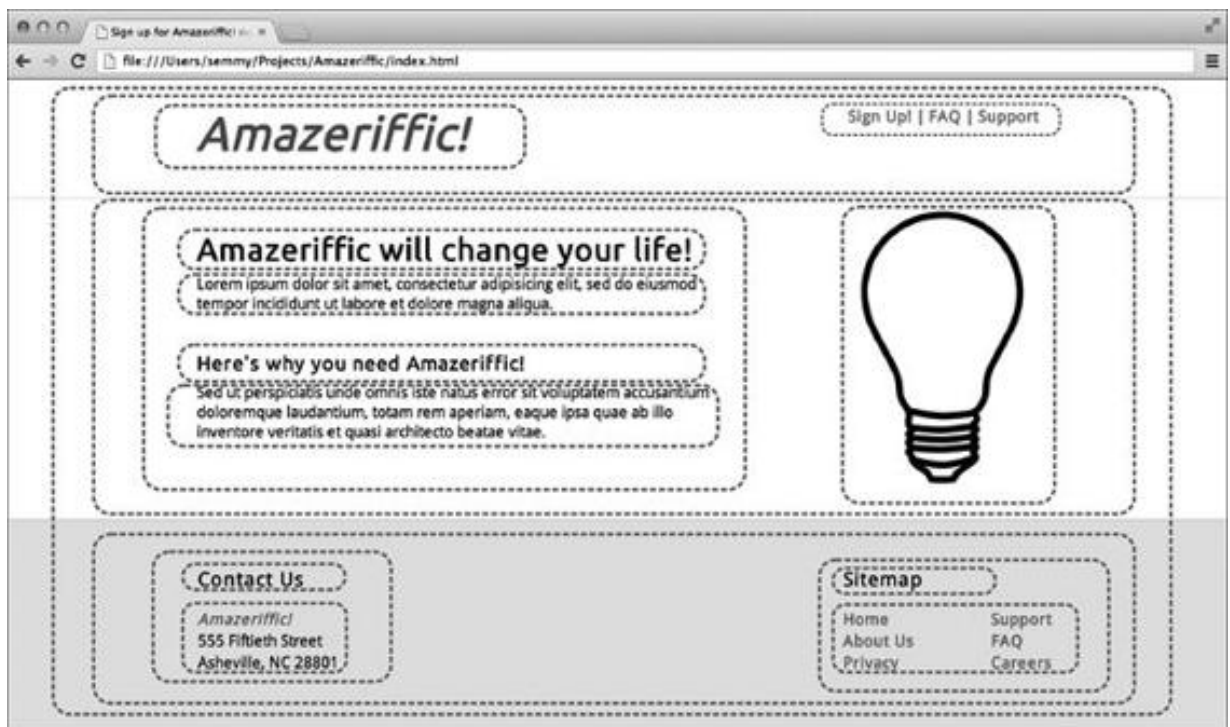


Figura 2.11 – O mockup do Amazeriffic, com anotações para ilustrar a estrutura.

Observe que eu poderia ir um passo além e dar nome às partes circuladas. Por exemplo, é relativamente óbvio o local em que estão o cabeçalho, o logo, os links de navegação, o rodapé, as informações para contato, o mapa do site, o conteúdo principal, o subconteúdo e a imagem. Todos eles representam algum elemento estrutural da página.

Visualizando a estrutura por meio de uma árvore

Depois de termos identificado todos os elementos estruturais, precisamos analisar como eles se encaixam. Para isso, podemos criar um diagrama de árvore para a estrutura, que especificará o conteúdo de todos os diversos elementos. A figura 2.12 mostra como é a aparência de uma representação em árvore dessa estrutura.

Implementando a estrutura com o nosso fluxo de trabalho

Após termos uma representação em árvore (seja no papel ou em nossa mente), é muito fácil codificar o HTML se conhecermos as tags que representam esses elementos estruturais de conteúdo. Como ainda não vimos todas as tags de que precisaremos, vou introduzi-las à medida que prosseguirmos.

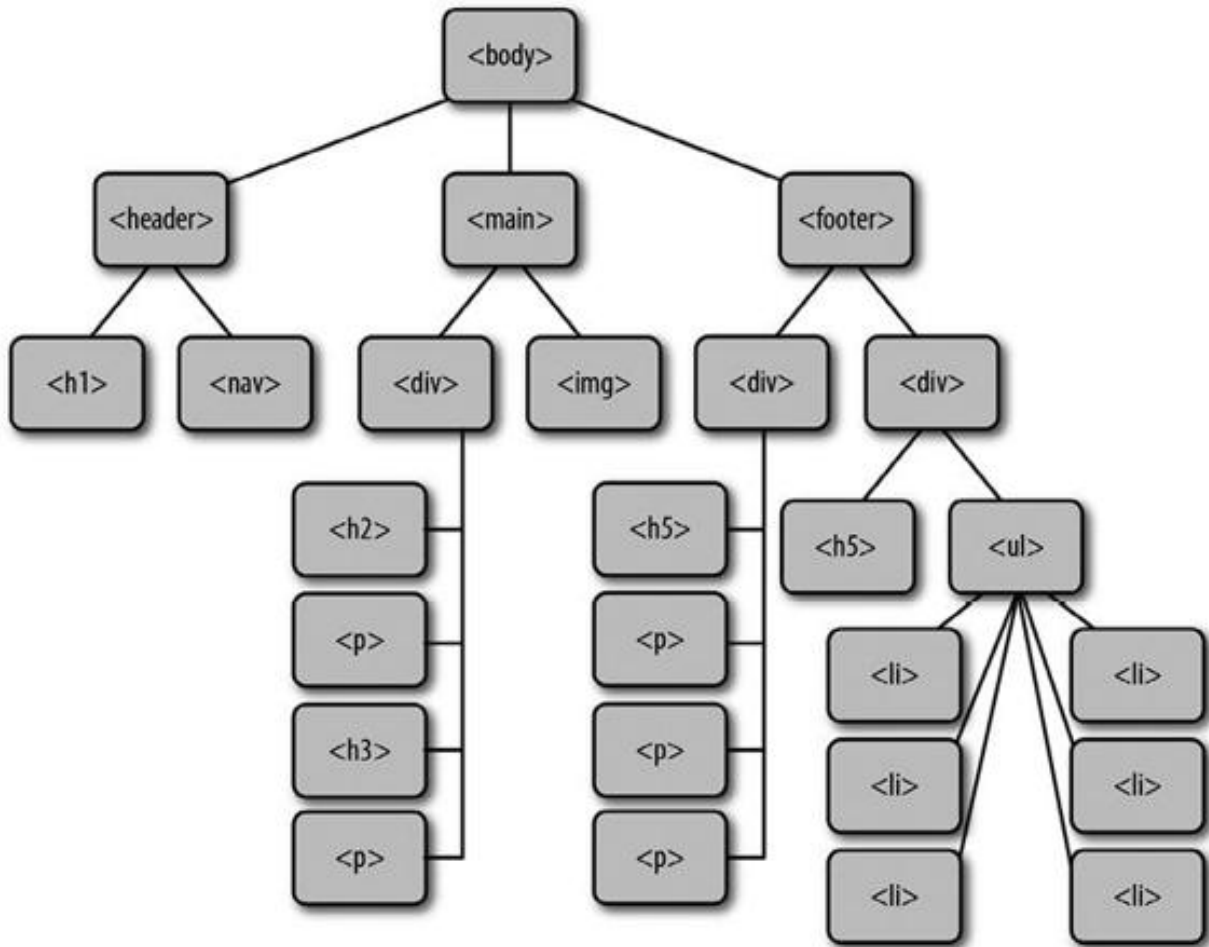


Figura 2.12 – A estrutura do Amazeriffic na forma de um diagrama de árvore.

Em primeiro lugar, iremos criar um diretório para armazenar o nosso projeto. Se você seguiu as instruções do primeiro capítulo, já deve haver um diretório *Projects* em seu diretório home (se você estiver no Mac OS ou no Linux) ou em sua pasta *Documents* (se você estiver no Windows). Queremos começar navegando até esse diretório a partir do aplicativo Terminal no Mac OS ou do Git Bash no Windows. Utilizaremos o comando `cd`:

```
hostname $ cd Projects
```

Quando estivermos nesse diretório, criaremos um diretório para o nosso projeto Amazeriffic. Qual comando nós usaremos para isso? Você está certo! Usaremos o comando `mkdir`:

```
hostname $ pwd
```

```
/Users/semmy/Projects
```

```
hostname $ mkdir Amazeriffic
```

Então podemos obter um pouco de feedback visual por meio do comando `ls` para saber se o diretório foi realmente criado e, por fim, navegaremos até esse diretório usando o comando `cd`.

```
hostname $ ls
```

```
Amazeriffic
```

```
hostname $ cd Amazeriffic
```

Nesse ponto, devemos estar no diretório de nosso novo projeto (podemos confirmar isso usando o comando `pwd`). A próxima tarefa importante a ser feita é submeter esse diretório ao controle de versões. Usaremos `git init` para criar um projeto Git:

```
hostname $ git init
```

```
Initialized empty Git repository in /Users/semmy/Projects/Amazeriffic/.git/
```

Agora que criamos um diretório e o colocamos sob o controle de versões, estamos prontos para iniciar a codificação! Abra o Sublime e, em seguida, abra o diretório *Amazeriffic* usando os atalhos de teclado descritos no capítulo anterior.

A seguir, podemos criar um novo documento HTML clicando com o botão da direita do mouse sobre o diretório no painel de navegação e selecionando New File (Arquivo Novo). Isso fará com que um arquivo sem nome seja criado, o qual poderá ser renomeado simplesmente se digitarmos *index.html*. Após o arquivo ter sido criado e ter recebido um nome, podemos abri-lo com um clique duplo nele. Vamos adicionar “Hello World” ao documento para que ele tenha algum conteúdo que possa ser visualizado no navegador.

Depois que um documento básico for criado, poderemos iniciar o Chrome e abrir a página. Se tudo correr bem, devemos ver “Hello World” no navegador.

A seguir, implementaremos o esqueleto de um documento HTML para começar. Substitua “Hello World” em seu arquivo *index.html* por este código:

```
<!doctype html>
```



```
<html>
  <head>
    <title>Amazeriffic</title>
  </head>
  <body>
    <h1>Amazeriffic</h1>
  </body>
</html>
```

Salve o arquivo e carregue-o novamente no navegador. Feito isso, você deverá ver algo semelhante ao que está sendo apresentado na figura 2.13.

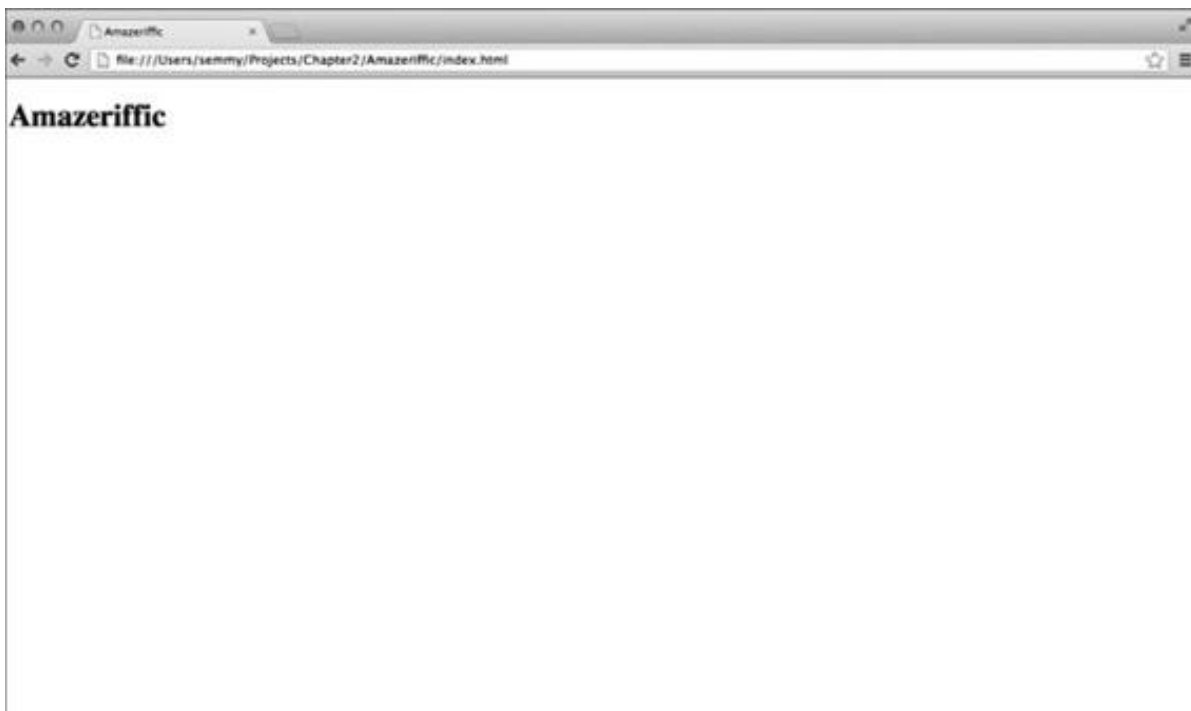


Figura 2.13 – O Amazeriffic depois que adicionamos alguns elementos básicos à página.

Feito isso, realizaremos o nosso primeiro commit. Inicialmente, verificaremos o status de nosso diretório de trabalho. É um bom hábito a ser adquirido, pois um feedback visual sempre ajuda. Entre outras coisas, isso nos dirá se alteramos inadvertidamente um arquivo que não tínhamos a intenção de mudar. Por enquanto, porém, vemos que o único arquivo alterado é *index.html*:

```
hostname $ git status
```

```
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
# index.html
```

Depois disso, adicionaremos e faremos o commit do arquivo *index.html*.

```
hostname $ git add index.html
hostname $ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
# (use "git rm --cached <file>..." to unstage)
#
# new file: index.html
hostname $ git commit -m "Add default index.html to repository."
[master (root-commit) fd60796] Add default index.html to the repository.
1 file changed, 10 insertions(+)
create mode 100644 index.html
```

Agora estamos prontos para realmente criar a estrutura da página. Se dermos uma olhada na árvore, veremos que temos um cabeçalho, uma seção de conteúdo principal e um rodapé. Cada um deles é filho do elemento `body`. O fato é que o HTML possui uma tag que representa todas essas três seções de um documento. As tags `<header>` e `<footer>` representam elementos que aparecem na parte superior e inferior de um documento, e a tag `<main>` representa a área de conteúdo principal de um documento:

```
<!doctype html>
<html>
  <head>
    <title>Amazeriffic</title>
  </head>
  <body>
```

```
<header>
  <h1>Amazeriffic</h1>
</header>
<main>
</main>
<footer>
</footer>
</body>
</html>
```

Observe que também movemos a tag `<h1>` que contém o logo do Amazeriffic para dentro da tag de cabeçalho (header). Isso porque o logo é um filho do cabeçalho em nosso diagrama de árvore.

A seguir, veremos que o canto superior direito da página possui uma pequena seção de navegação com links para uma página de Sign Up (Cadastrar), uma página de FAQ (Perguntas Frequentes) e uma página de Support (Suporte). É claro que o HTML tem uma tag que suporta um elemento de navegação, que se chama `nav`. Portanto iremos acrescentar essa seção à nossa tag `<header>`:

```
<header>
  <h1>Amazeriffic</h1>
  <nav>
    <a href="#">Sign Up!</a> |
    <a href="#">FAQ</a> |
    <a href="#">Support</a>
  </nav>
</header>
```



Observe que o elemento `nav` contém vários links separados pelo símbolo `|`. Esse símbolo está logo acima da tecla Enter em seu teclado, localizado juntamente com a barra invertida. É preciso pressionar Shift para digitar o símbolo `|`.

Os links no elemento `nav` estão contidos em tags `<a>`. Conforme mencionamos anteriormente, as tags `<a>` contêm atributos `href`, que normalmente contêm um link para a página para a qual devemos ser direcionados ao clicarmos no link. Como não estamos realmente efetuando o link para lugar algum nesse exemplo, usamos o símbolo `#` como um placeholder temporário no lugar do link.

Pelo fato de termos concluído a seção `<header>`, provavelmente é

uma boa ideia efetuar o commit em nosso repositório Git. Pode ser interessante executar um `git status` antes para ver os arquivos alterados em nosso repositório. Em seguida, execute um `git add` e um `git commit` com uma mensagem significativa de commit.

Estruturando o conteúdo principal

Agora que completamos a seção `<header>`, podemos prosseguir para a seção `<main>`. Veremos que, assim como no cabeçalho, há duas partes principais na estrutura que compõe a seção principal. Temos o conteúdo do lado esquerdo e a imagem que está do lado direito. O conteúdo do lado esquerdo está dividido em duas seções separadas, portanto precisamos garantir que isso será levado em consideração.

Para criar a estrutura do conteúdo do lado esquerdo, usaremos quatro tags novas. Usaremos duas tags de cabeçalho (`<h2>` e `<h3>`), que representam textos de cabeçalho menos importantes que o que está em uma tag `<h1>`. Também usaremos a tag `<p>`, que representa um conteúdo de parágrafo. Além disso, usaremos a tag ``, que representa uma lista não ordenada, juntamente com suas tags `` relacionadas, que correspondem aos itens da lista.

E, por fim, usaremos a tag `` para incluir a imagem da lâmpada em nosso layout. Observe que a tag `` não possui uma tag de fechamento associada a ela. Isso ocorre porque o HTML5 inclui um conjunto de elementos chamados de elementos *vazios* (void). Os elementos vazios normalmente não têm conteúdo e não exigem uma tag de fechamento.

Também veremos que a tag `` possui um atributo obrigatório chamado `alt`. Esse atributo contém uma descrição textual da imagem. Isso é importante para tornar nossa página acessível às pessoas com deficiência visual que, com frequência, utilizam leitores de tela ao navegar pela Internet.



A imagem da lâmpada pode ser baixada a partir de <http://www.learningwebappdev.com/lightbulb.png>. Para que essa imagem

apareça em sua página, é necessário salvá-la no mesmo diretório em que o seu arquivo *index.html* estiver.

Após termos adicionado o conteúdo estruturado à tag `<main>`, ele terá o seguinte aspecto:

```
<h2>Amazeriffic will change your life!</h2>
<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
    eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
<h3>Here's why you need Amazeriffic</h3>
<ul>
  <li>It fits your lifestyle</li>
  <li>It's awesome</li>
  <li>It rocks your world</li>
</ul>

```

Quando recarregarmos a página em nosso navegador, sua aparência deverá ser semelhante à da figura 2.14.

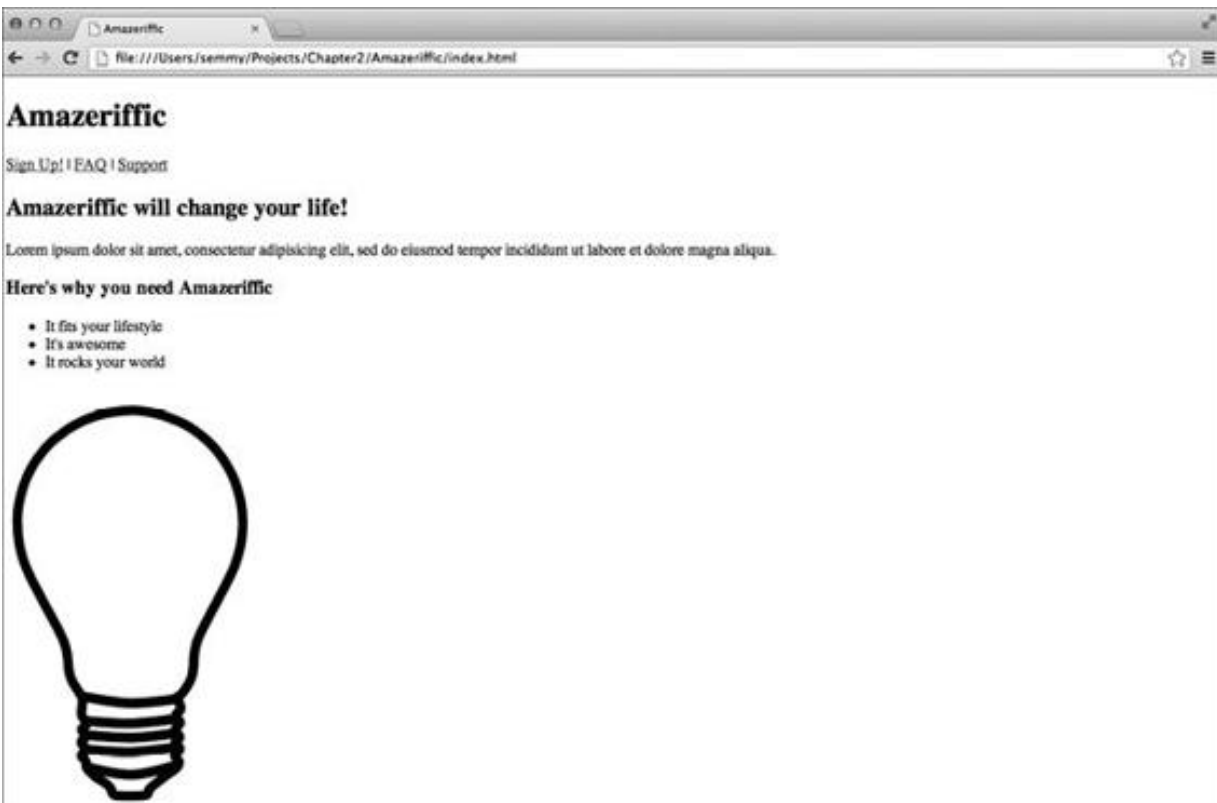


Figura 2.14 – O Amazeriffic após a estruturação do elemento principal.

A essa altura, será uma boa ideia passar o seu código pela ferramenta de validação e garantir que nada tenha sido inadvertidamente omitido. Depois que você estiver satisfeito com o código, faça outro commit no repositório Git e prosseguiremos para o rodapé.

Estruturando o rodapé

O rodapé contém duas seções principais, assim como as demais seções. Uma das seções inclui informações de contato da empresa e a outra corresponde a um conjunto de links ao qual iremos nos referir como *mapa do site* (sitemap). Além do mais, o mapa do site está dividido em duas colunas.

Em primeiro lugar, não há nenhum elemento HTML que represente informações de contato. Isso não é um problema porque o HTML nos oferece duas tags genéricas chamadas `<div>` e ``, que nos permitem criar elementos representando estruturas que podemos definir por conta própria. Aprenderemos a diferença entre os elementos `div` e `span` no próximo capítulo.

Nesse caso, temos duas estruturas separadas no rodapé: as informações de contato (“Contact”) e o mapa do site (“Sitemap”). Desse modo, criaremos dois elementos `<div>`, cada qual com um atributo `class` que especifica o tipo do elemento. A essa altura, você pode pensar em um atributo de classe como um atributo utilizado para acrescentar significados às tags `<div>` e `` genéricas.

Além do mais, usaremos outra tag `` para criar uma lista não ordenada para o mapa do site. O HTML resultante que cria a estrutura de nosso rodapé tem o seguinte aspecto:

```
<footer>
  <div class="contact">
    <h5>Contact Us</h5>
    <p>Amazeriffic!</p>
    <p>555 Fiftieth Street</p>
    <p>Asheville, NC 28801</p>
  </div>
```

```
<div class="sitemap">
  <h5>Sitemap</h5>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About Us</a></li>
    <li><a href="#">Privacy</a></li>
    <li><a href="#">Support</a></li>
    <li><a href="#">FAQ</a></li>
    <li><a href="#">Careers</a></li>
  </ul>
</div>
</footer>
```

Adicione o conteúdo do rodapé ao seu documento HTML, passe-o pela ferramenta de validação de HTML para garantir que você não cometeu nenhum erro e faça o commit do arquivo no repositório Git. Retornaremos a esse exemplo no capítulo 3, quando iremos estilizá-lo.

Resumo

Neste capítulo, aprendemos a estruturar a interface do usuário de nossa aplicação usando o HTML. O HTML é uma linguagem de marcação que nos permite utilizar tags para definir uma estrutura conhecida como DOM. O navegador usa o DOM para criar uma apresentação visual da página.

O DOM é uma estrutura hierárquica e pode ser facilmente representado por meio de um diagrama de árvore. Às vezes, é útil pensar no DOM na forma de uma árvore porque ela representa mais claramente os relacionamentos entre descendentes, filhos e pais dos elementos.

A validação é uma ferramenta útil que nos ajuda a evitar erros simples e as armadilhas relacionadas ao HTML.

Neste capítulo, aprendemos sobre diversas tags, que estão listadas na tabela 2.1. Todas elas representam elementos estruturais específicos, com exceção da tag `<div>`. Normalmente, associamos

um atributo de classe a uma tag `<div>` para atribuir-lhe algum tipo de significado semântico.

Tabela 2.1 – Tags HTML

Tag	Descrição
<code><html></code>	O contêiner principal de um documento HTML.
<code><head></code>	Contém informações meta sobre o documento.
<code><body></code>	Inclui o conteúdo que será renderizado no navegador.
<code><header></code>	O cabeçalho da página.
<code><h1></code>	O cabeçalho mais importante (somente um por documento).
<code><h2></code>	O segundo cabeçalho mais importante.
<code><h3></code>	O terceiro cabeçalho mais importante.
<code><main></code>	A área de conteúdo principal de seu documento.
<code><footer></code>	O conteúdo referente ao rodapé de seu documento.
<code><a></code>	Uma âncora, ou seja, um link para outro documento ou um elemento que pode ser clicado.
<code></code>	Uma lista de itens em que a ordem não importa.
<code></code>	Uma lista de itens em que a ordem importa.
<code></code>	Um elemento de uma lista.
<code><div></code>	Um contêiner para uma subestrutura.

Práticas e leituras adicionais

Lembre-se de que, se estiver tendo problemas em concluir os exemplos deste capítulo, você poderá ver o HTML completo em nossa [página](http://github.com/semmypurewal/LearningWebAppDev) do [GitHub](http://github.com/semmypurewal/LearningWebAppDev) (<http://github.com/semmypurewal/LearningWebAppDev>).

Memorização

Agora que já aprendemos o básico sobre o HTML, podemos acrescentar mais alguns passos às nossas metas de memorização. Além dos cinco passos mencionados no capítulo anterior, adicione os passos a seguir aos seus exercícios:

1. Abra o arquivo no Chrome usando os atalhos de teclado.
2. Modifique *index.html* para incluir as tags `<!doctype>`, `<html>`, `<head>` e `<body>`.
3. Adicione uma tag `<p>` que simplesmente contenha as palavras “Hello World”.
4. Carregue novamente o arquivo no Chrome e certifique-se de que ele será corretamente renderizado (se não for, corrija-o).
5. Faça o commit das alterações em seu repositório Git a partir da linha de comando.
6. Acrescente as tags `<header>`, `<main>` e `<footer>` à sua tag `<body>`.
7. Confirme se o arquivo será corretamente renderizado no Chrome.
8. Valide o arquivo com a ferramenta de validação de HTML.
9. Faça o commit das alterações de *index.html* em seu repositório Git.

Diagramas de árvore

Desenhe um diagrama de árvore para o documento HTML a seguir. Também usaremos este documento HTML para exercícios com alguns problemas que se encontram no final dos capítulos 3 e 4:

```
<!doctype html>
<html>
  <head>
  </head>
  <body>
    <h1>Hi</h1>
    <h2 class="important">Hi again</h2>
```

```
<p class="a">Random unattached paragraph</p>
<div class="relevant">
  <p class="a">first</p>
  <p class="a">second</p>
  <p>third</p>
  <p>fourth</p>
  <p class="a">fifth</p>
  <p class="a">sixth</p>
  <p>seventh</p>
</div>
</body>
</html>
```

Criando a página de FAQ para o Amazeriffic

Na barra de navegação do Amazeriffic, existe um link inativo para uma página de *FAQ* (Frequently Asked Questions, ou Perguntas Frequentes). Crie uma página que siga exatamente o mesmo estilo do cabeçalho e do rodapé, mas que tenha uma lista de perguntas e respostas como conteúdo principal. Utilize texto *lorem ipsum* como placeholder (a menos que você realmente queira criar perguntas e respostas).

Salve o arquivo como *faq.html*. Você pode criar um link para a página ao preencher o atributo `href` de sua tag `<a>` associada – configure-a com *faq.html*. Se as duas páginas forem incluídas no mesmo diretório, você deverá ser capaz de clicar a partir da página de abertura principal e chegar até a página de FAQ. De modo semelhante, você pode efetuar o link de volta para *index.html* a partir de *faq.html*.

No capítulo 3, iremos estilizar essa página.

Mais sobre o HTML

Ao longo deste livro, farei referências à documentação do Mozilla Developer Network (<https://developer.mozilla.org/en-US/docs/Web/HTML>) para que você possa obter mais informações

sobre determinados assuntos. Esse site inclui uma ótima visão geral sobre o HTML. Sugiro que você dê uma olhada para obter documentações e conhecer recursos avançados com mais detalhes.

CAPÍTULO 3

Estilo

No capítulo anterior, aprendemos a estruturar o conteúdo de um documento HTML e a usar alguns modelos mentais relacionados. Porém as páginas que criamos deixaram muito a desejar no que diz respeito ao design e ao estilo.

Neste capítulo, vamos tentar atenuar alguns desses problemas e aprender a alterar a maneira pela qual um documento HTML é apresentado usando o CSS (*Cascading Style Sheets*, ou Folhas de Estilo em Cascata). Conforme mencionamos no capítulo anterior, você terá informações suficientes para começar a trabalhar com o CSS, e a seção “Práticas e leituras adicionais” irá incentivá-lo a explorar outros recursos.

Hello, CSS!

Para pôr a mão na massa, vamos começar com um exemplo bem simples de HTML, semelhante aos do início do capítulo anterior. Abra uma janela do terminal e crie um diretório chamado *Chapter3* em nosso diretório *Projects*.

Agora abra o Sublime Text e vá para o diretório *Chapter3*, da mesma maneira que fizemos no capítulo anterior. Crie o arquivo HTML a seguir e salve-o no diretório como *index.html*:

```
<!doctype html>
<html>
  <head>
    <title>My First Web App</title>
    <link href="style.css" rel="stylesheet" type="text/css">
  </head>
```

```
<body>
  <h1>Hello, World!</h1>
  <p>This is a paragraph.</p>
</body>
</html>
```

Esse arquivo define uma página HTML bem simples, com um elemento `h1` e um elemento `p` contidos no elemento `body`. Você notará imediatamente que o exemplo inclui também uma nova tag contida na tag `<head>`. A tag `<link>` faz a ligação com um arquivo externo de folhas de estilo que descreve como o documento deve ser apresentado.

O arquivo *style.css* que está faltando pode ser criado usando o Sublime a partir da janela de navegação de arquivos do editor. Preencha *style.css* com o conteúdo a seguir:

```
body {
  background: lightcyan;
  width: 800px;
  margin: auto;
}
h1 {
  color: maroon;
  text-align: center;
}
p {
  color: gray;
  border: 1px solid gray;
  padding: 10px;
}
```

Esse é um exemplo simples de um arquivo CSS. Esse arquivo em particular define a cor de fundo (background) do elemento `body` com uma cor azul clara (lightcyan) e informa o navegador que o texto contido no elemento `h1` deve ser renderizado na cor marrom-avermelhada (maroon). Além disso, definimos o corpo com uma largura de 800 pixels, e a margem foi definida como `auto` para que o corpo ficasse centralizado na página. Por fim, definimos uma cor cinza para o texto contido no elemento `p` e criamos uma borda

estreita em torno dele.

Essencialmente, um arquivo CSS descreve de que modo elementos específicos do HTML devem ser apresentados pelo navegador. Por exemplo, a figura 3.1 mostra o arquivo HTML anterior renderizado com o arquivo CSS.

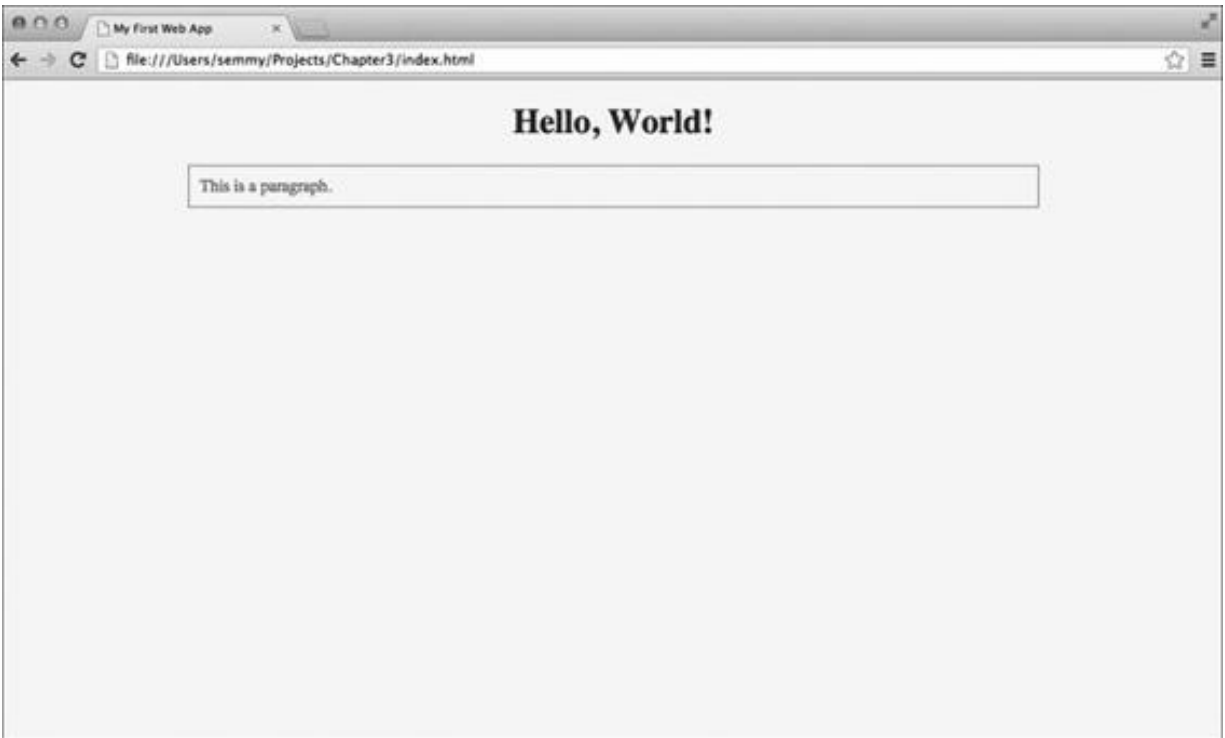


Figura 3.1 – index.html aberto no Chrome, com o acréscimo da folha de estilo.

Se não incluíssemos o arquivo CSS, a página teria a aparência mostrada na figura 3.2.

Se os arquivos *index.html* e *style.css* estiverem no mesmo diretório, você deverá ser capaz de abrir o primeiro em seu navegador e perceber que sua aparência será igual à da primeira imagem. Com efeito, se você criar uma segunda página HTML que tenha um conteúdo diferente e fizer a ligação com o mesmo arquivo CSS, ambas as páginas serão apresentadas da maneira especificada pelo CSS. Esse é um dos aspectos interessantes do CSS – ele permite que o estilo de várias páginas seja definido no mesmo lugar!

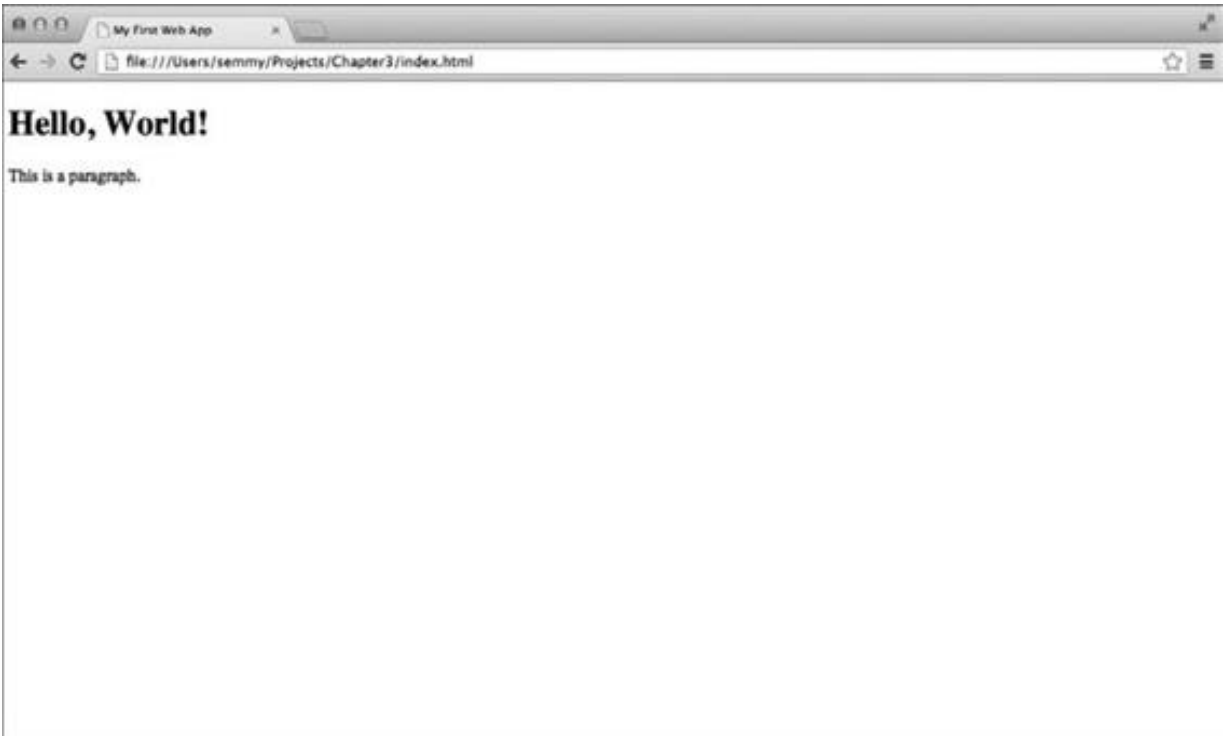


Figura 3.2 – *index.html* aberto no Chrome, sem o acréscimo da folha de estilo.

Conjuntos de regras

Um arquivo CSS é constituído de uma coleção de *conjuntos de regras* (rulesets), e um conjunto de regras é simplesmente uma coleção de regras de estilo aplicada a algum subconjunto de elementos do DOM (que, conforme você deve se lembrar, é o objeto hierárquico definido pelo documento HTML). Um conjunto de regras é composto de um *seletor* (podemos pensar nele como um nome de tag, por enquanto), uma chave de abertura, uma lista de regras e uma chave de fechamento. Cada regra é constituída de uma *propriedade* específica, seguida de dois-pontos, seguidos de um valor (ou de uma lista de valores separados por espaços), seguido de ponto e vírgula:

```
body {  
  width: 800px;  
  background: lightcyan;  
  color: #ff0000;
```

}

Esse é um exemplo de um conjunto de regras que será aplicado ao elemento `body` do DOM. Nesse caso, o seletor é `body`, que é simplesmente o nome do elemento HTML ao qual queremos aplicar o estilo. Essas regras serão aplicadas a todo o conteúdo do elemento `body`, assim como a qualquer elemento contido no elemento `body`. O conjunto de regras é composto de três regras: uma que especifica o valor da propriedade `width` (largura), uma que especifica o valor da propriedade `background` (cor de fundo) e outra que especifica um valor para a propriedade `color` (cor).



No CSS, há duas maneiras de especificar cores. A primeira utiliza nomes de cores CSS para as cores comumente utilizadas (<http://www.crockford.com/wrrrld/color.html>). A segunda maneira consiste em especificar as cores no CSS usando um código de cor hexadecimal. Um código de cor hexadecimal é constituído de seis dígitos hexadecimais (0–9 ou A–F). O primeiro par representa a quantidade de vermelho na cor, o segundo representa a quantidade de verde e o terceiro representa a quantidade de azul. Essas três cores são as cores primárias do modelo de cores RGB.

Um desenvolvedor CSS experiente tem um sólido conhecimento dos tipos de propriedades que podem ser definidos para um determinado elemento e dos tipos de valores que cada propriedade aceita. Diversas propriedades, por exemplo, a cor de fundo e a cor, podem ser utilizadas para estilizar a maior parte dos elementos HTML.

Comentários

Nosso código CSS também pode incluir comentários. Assim como os comentários HTML, os comentários CSS correspondem a anotações no código que são totalmente ignoradas pelo navegador. Por exemplo, podemos inserir comentários no conjunto anterior de regras CSS:

```
/* Estamos estilizando o corpo aqui */  
body {  
  width: 800px; /* Define a largura do corpo para 800 pixels */  
  background: lightcyan; /* Define a cor de fundo com uma cor azul clara */  
  color: #ff0000; /* Define a cor do primeiro plano como vermelho */
```


}

Algumas pessoas irão sugerir que você seja bastante pródigo com os seus comentários em todos os programas. Eu tenho a tendência de acreditar que você deve deixar que o seu código fale por si mesmo o máximo possível e que você deve minimizar os locais em que os comentários são necessários. Com efeito, desenvolvedores experientes de CSS acharão os comentários anteriores supérfluos, pois, uma vez tendo adquirido um pouco de conhecimento, esses tipos de comentário serão redundantes. Por outro lado, há ocasiões em que o que o seu código deve fazer não é óbvio, e, nesses casos, é uma boa ideia inserir comentários.

Se você está começando a trabalhar com o CSS, sempre recomendo errar para o lado de exceder nos comentários. Usarei os comentários de forma bem generosa na maior parte do restante deste capítulo para facilitar o entendimento.

Padding, borda e margem

Na maior parte das vezes, os elementos HTML são apresentados de uma entre duas maneiras. A primeira maneira é *inline*, que se aplica a elementos que têm o tipo `a` ou `span`, por exemplo. Isso significa que (entre outras coisas) o conteúdo irá aparecer na mesma linha que o conteúdo que o cerca:

```
<div>
```

```
  This is a paragraph and this <span>word</span> appears inline. This
```

```
  <a href="http://www.example.com">link</a> will also appear inline.
```

```
</div>
```

Se adicionarmos esse código à tag `<body>` do arquivo *index.html*, a página será renderizada de modo semelhante à imagem mostrada na figura 3.3.

Na realidade, é mais comum que os elementos sejam do tipo *bloco*, e não elementos *inline*. Isso significa que o conteúdo do elemento aparecerá em uma nova linha, fora do fluxo normal do texto. Elementos do tipo bloco que já vimos incluem os elementos `p`, `nav`,

main e div.

<div>

This is a paragraph and this **<div>word</div>** appears inline. This

link will also appear inline.

</div>



Figura 3.3 – Um exemplo de elementos inline dentro de um elemento bloco.

Esse código resultará em uma aparência diferente, conforme mostrado na figura 3.4.



Figura 3.4 – Um exemplo de elementos bloco dentro de outro elemento bloco.

Tanto elementos do tipo bloco quanto os elementos inline possuem propriedades relativas à cor de fundo e cor. No entanto, elementos do tipo bloco possuem três outras propriedades que são bem práticas para estilizar a página com layouts personalizados. Essas propriedades são: padding (preenchimento), borda e margem.

A propriedade padding representa o espaçamento entre o conteúdo do elemento e a borda, e a margem corresponde ao espaçamento entre o elemento e o elemento que o cerca. A propriedade borda representa o espaço entre o padding e a margem. Isso está sendo mostrado na figura 3.5.

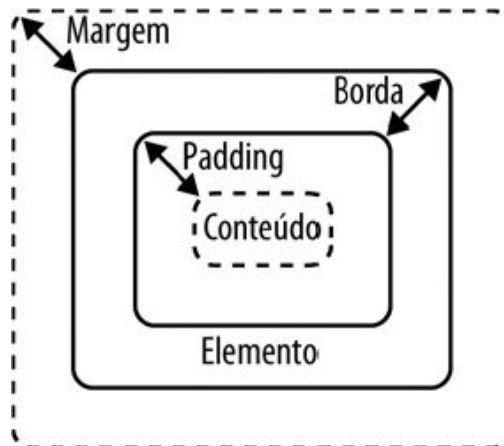


Figura 3.5 – A margem, a borda e o padding de um elemento do DOM referente a um bloco.

Aqui está um exemplo simples que pode ser usado para manipular o padding, a borda e a margem de alguns elementos diferentes. O primeiro arquivo é *margin_border_padding.html*:

```
<!doctype html>
<html>
  <head>
    <title>Chapter 3 -- Margin, Border, and Padding Example</title>
    <link href="margin_border_padding.css" rel="stylesheet" type="text/css">
  </head>
  <body>
    <div>
      <p>THIS IS A PARAGRAPH CONTAINED INSIDE A DIV</p>
    </div>
  </body>
</html>
```

E aqui está o arquivo *margin_border_padding.css*, que é referenciado no arquivo HTML anterior. Esses arquivos podem ser criados no mesmo diretório usando o Sublime e, em seguida, o arquivo *margin_border_padding.html* pode ser aberto no Chrome:

```
body {
  background: linen;
  width: 500px;
  margin: 200px auto;
}

div {
```

```
border: 5px solid maroon;  
text-align: center;  
padding: 5px;  
}  
p {  
  border: 2px dashed blue;  
}
```

Se você digitou tudo corretamente, a página deverá ter uma aparência semelhante à da figura 3.6.

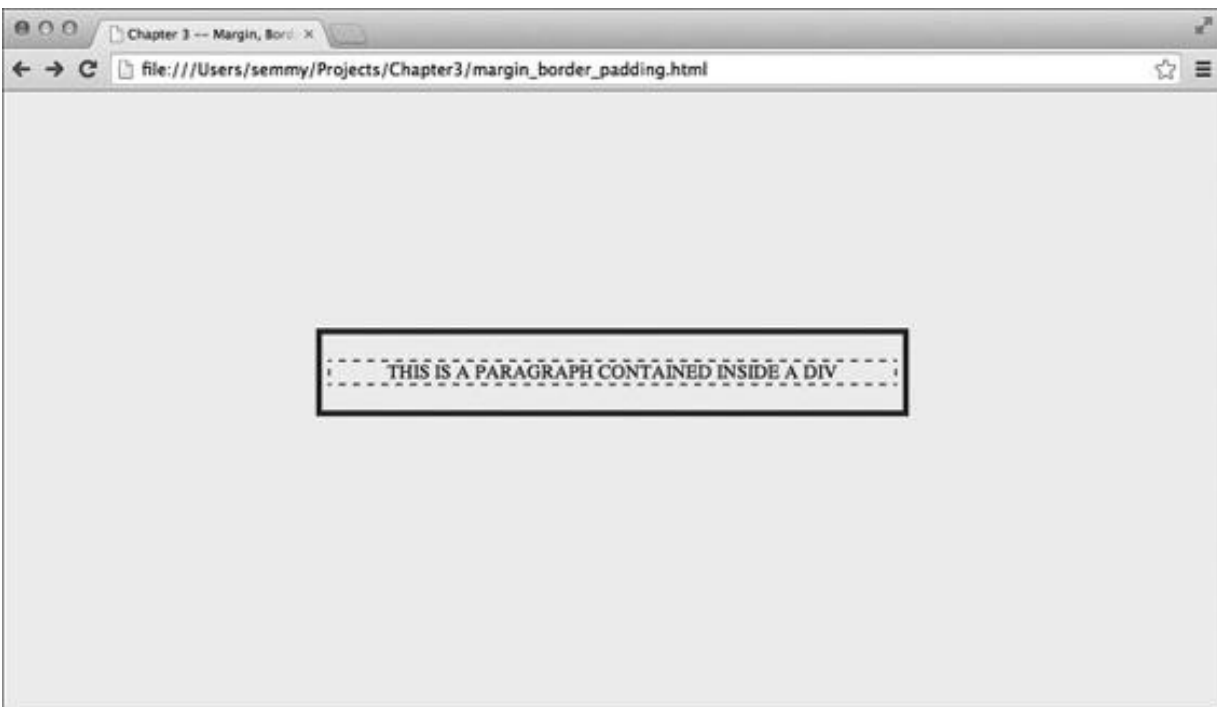


Figura 3.6 – Um exemplo para explorar a margem, o padding e a borda.

Você pode perceber que ambos os elementos (`div` e `p`) são do tipo bloco, portanto cada qual possui as propriedades borda, margem e padding próprias. Se você ainda não fez isso, aproveite a oportunidade para criar o HTML e o CSS que acabaram de ser apresentados. Em seguida, invista algum tempo alterando as propriedades relativas ao padding, à borda e à margem de cada elemento para ver como elas afetam a maneira como a página é apresentada.

Seletores

O aspecto mais importante do CSS resulta do uso eficiente dos seletores. Já vimos um seletor básico de *tipo*, que utiliza nomes de tag para selecionar todos os elementos que têm esse nome. Por exemplo, considere o HTML a seguir:

```
<body>
  <h1>Hello World!</h1>
  <p>This is a paragraph.</p>
  <p>This is a second paragraph.</p>
</body>
```

E vamos supor que esse HTML esteja sendo estilizado com o CSS a seguir:

```
h1 {
  background: black;
  color: white;
}
p {
  color: red;
  margin: 10px;
  padding: 20px;
}
```

O primeiro conjunto de regras estiliza o elemento `h1`, e o segundo estiliza *ambos* os elementos `p`. Em muitos casos, esse é exatamente o comportamento que esperamos. Em outros, porém, vamos querer estilizar as tags do primeiro e do segundo parágrafos de forma independente.

Classes

No capítulo 2, vimos que podemos adicionar um atributo de classe às tags `<div>` para diferenciá-las. O fato é que podemos adicionar uma classe a *qualquer* elemento do DOM. Por exemplo, podemos reescrever o HTML anterior para que tenha o seguinte aspecto:

```
<body>
  <h1>Hello World!</h1>
```

```
<p class="first">This is a paragraph.</p>
<p class="second">This is a second paragraph.</p>
</body>
```

Agora podemos selecionar a tag específica do parágrafo usando o seu nome de classe:

```
h1 {
  background: black;
  color: white;
}
p.first {
  color: red;
  margin: 10px;
  padding: 20px;
}
```

Nesse exemplo, o conjunto de regras `p.first` será aplicado somente ao primeiro parágrafo. Se a classe aparecer somente em um determinado tipo de elemento (que, normalmente, é o caso), podemos omitir o nome da tag e simplesmente utilizar a classe:

```
.first {
  color: red;
  margin: 10px;
  padding: 20px;
}
```

Pseudoclasses

No capítulo anterior, vimos que é possível criar elementos no DOM que podem ser clicados usando a tag `<a>`:

```
<body>
  <a href="http://www.example.com">Click Me!</a>
</body>
```

O elemento `a` pode ser estilizado, assim como qualquer outros elemento do DOM por meio do CSS. Por exemplo, podemos criar um arquivo CSS que altere a cor de todos os links:

```
a {
  color: cornflowerblue;
```

```
}
```

Entretanto normalmente é interessante ter links com cores diferentes, conforme o usuário já tenha ou não clicado nesses links. O CSS nos permite alterar a cor dos links já acessados por meio da adição de outro conjunto de regras:

```
a {  
  color: cornflowerblue;  
}  
a:visited {  
  color: tomato;  
}
```

Nesse exemplo, `visited` é um exemplo de uma *pseudoclas*se CSS do elemento `a`. Seu comportamento é muito semelhante ao de uma classe normal no sentido de que podemos estilizar os elementos que tenham essa classe, da mesma forma que fazemos com uma classe normal. A principal diferença é que o navegador implicitamente adiciona a classe por nós.

Um caso de uso comum para pseudoclasses CSS ocorre ao alterarmos o modo como um link é apresentado quando um usuário passa o mouse sobre ele. Isso pode ser feito por meio do uso da pseudoclas

`se` `hover` em um elemento `a`. No exemplo a seguir, modificaremos o exemplo anterior para que o link fique sublinhado somente quando o usuário estiver passando o mouse sobre ele:

```
a {  
  color: cornflowerblue;  
  text-decoration: none; /* remove o sublinhado default */  
}  
a:visited {  
  color: tomato;  
}  
a:hover {  
  text-decoration: underline;  
}
```

Seletores mais complexos

À medida que nosso diagrama de árvore para o DOM torna-se mais complexo, é necessário criar seletores mais complicados. Por exemplo, considere este HTML:

```
<body>
  <h1>Hello World!</h1>
  <div class="content">
    <ol>
      <li>List Item <span class="number">first</span></li>
      <li>List Item <span class="number">second</span></li>
      <li>List Item <span class="number">third</span></li>
    </ol>

    <p>This is the <span>first</span> paragraph.</p>
    <p>This is the <span>second</span> paragraph.</p>
  </div>

  <ul>
    <li>List Item <span class="number">1</span></li>
    <li>List Item <span class="number">2</span></li>
    <li>List Item <span class="number">3</span></li>
  </ul>
</body>
```

O HTML contém duas listas, diversos parágrafos e vários itens de lista. Podemos facilmente selecionar e estilizar elementos genéricos, conforme discutimos anteriormente. Por exemplo, se quisermos criar uma borda arredondada em torno do elemento da lista ordenada (ol), podemos aplicar o seguinte conjunto de regras:

```
ol {
  border: 5px solid darksalmon;
  border-radius: 10px;
}
```

Agora suponha que queremos fazer com que os itens da lista ordenada tenham a cor marrom. Para isso, nossa intuição imediata seria fazer o seguinte:

```
li {
  color: brown;
}
```

No entanto isso irá alterar os elementos li tanto da lista não

ordenada quanto da lista ordenada. Podemos ser mais específicos em nosso seletor ao selecionarmos *apenas* os elementos `li` da lista ordenada:

```
ol li {  
  color: brown;  
}
```

Se houver várias listas ordenadas na página, podemos ser mais específicos ainda e selecionar somente os elementos `li` que sejam descendentes do elemento `div` de classe `content`:

```
.content li {  
  color: brown;  
}
```

Agora suponha que desejamos configurar a cor de fundo dos primeiros elementos `li` de ambas as listas para amarelo. O fato é que existe uma pseudoclassee que representa os elementos correspondentes ao *primeiro filho* de seus pais:

```
li:first-child {  
  background: yellow;  
}
```

De modo semelhante, podemos selecionar os elementos que são o segundo, o terceiro, o quarto filho etc., usando a pseudoclassee `nth-child`:

```
li:nth-child(2) {  
  background: orange;  
}
```

Regras em cascata

O que acontece quando dois conjuntos diferentes de regras utilizam seletores cujo alvo é o mesmo elemento no CSS? Por exemplo, considere um elemento `p` com a classe `greeting`:

```
<p class="greeting">Hello, Cascading Rules!</p>
```

Agora suponha que tenhamos duas regras que selecionem esse elemento e que apliquem estilos diferentes:

```
p {  
  color: yellow;  
}  
  
p.selected {  
  color: green;  
}
```

Qual das regras será aplicada à classe anterior? O fato é que há um conjunto de *regras em cascata* que os navegadores aplicam quando surgem conflitos. Nesse caso, a regra mais específica (a classe) tem precedência. Entretanto o que aconteceria se fizéssemos algo deste tipo?

```
p {  
  color: yellow;  
}  
  
p {  
  color: green;  
}
```

Nesse caso, as regras em cascata especificam que o conjunto de regras que aparecer depois na lista do CSS será aplicado. Portanto, nesse caso, o(s) parágrafo(s) serão verdes (green). Se trocássemos as regras de lugar, o(s) parágrafo(s) seriam amarelos (yellow).

Herança

Caso você ainda não tenha percebido até agora, os descendentes herdam propriedades de seus ancestrais. Isso significa que, se criarmos um estilo para um elemento, todos os descendentes desse elemento no DOM também terão esse estilo, a menos que ele seja sobrescrito por outro conjunto de regras que tenha esse elemento como alvo. Por exemplo, se selecionarmos o corpo (body) e alterarmos a propriedade de cor (color), todos os elementos que forem descendentes do corpo (o que significa todos os elementos que aparecem na página) herdarão essa cor. Essa é uma parte essencial do CSS, e é por isso que manter uma visualização da hierarquia do DOM em mente ao estilizar os elementos pode ajudar:

```
body {
```

```

    background: yellow;
}
/**
 * Como h1 é descendente da tag body
 * ele terá uma cor de fundo amarela (yellow).
 */
h1 {
    color: red;
}
/**
 * h2 também é descendente de body, mas
 * iremos sobrescrever a cor de fundo para que
 * não seja amarela.
 */
h2 {
    background: green;
}

```

Embora a maioria das propriedades CSS funcione dessa maneira, vale a pena observar que nem todas são herdadas por padrão. As propriedades não herdadas mais evidentes estão relacionadas a elementos do tipo bloco (as regras relativas à margem, padding e borda não são herdadas de seus ancestrais):

```

body {
    margin: 0;
    padding: 0;
}
/**
 * h1 não herdar a margem e o padding de body,
 * mesmo que não especifiquemos nenhuma alternativa
 */
h1 {
}

```

Layouts com floats

Vimos propriedades que afetam o estilo básico dos elementos do DOM. Porém há outras propriedades mais genéricas que afetam o layout geral da página em relação a um único elemento. Essas

propriedades proporcionam um maior controle ao desenvolvedor no que diz respeito ao local em que os objetos aparecerão. Uma das propriedades desse tipo mais comumente utilizadas é a propriedade `float`. Essa propriedade flexível nos permite criar layouts mais complexos que o layout em forma de pilha, criado automaticamente pelo HTML.

A propriedade `float` de um elemento do DOM pode ser definida como `left` ou `right`. Isso desvia o elemento do fluxo normal (que, normalmente, faz com que os elementos do tipo bloco sejam empilhados uns sobre os outros) e o desloca o máximo para a esquerda ou para a direita dentro do elemento que o contém, supondo que haja espaço suficiente para isso. Por exemplo, considere o seguinte trecho de código HTML:

```
<body>
  <main>
    <nav>
      <p><a href="link1">link1</a></p>
      <p><a href="link2">link2</a></p>
      <p><a href="link3">link3</a></p>
      <p><a href="link4">link4</a></p>
    </nav>
    <p class="content"> Lorem ipsum dolor sit amet, consectetur
      adipiscing elit, sed do eiusmod tempor incididunt ut labore et
      dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
      exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
      Duis aute irure dolor in reprehenderit in voluptate velit esse cillum
      dolore eu fugiat nulla pariatur.
    </p>
  </main>
</body>
```

Nesse exemplo, um elemento `nav` e um elemento `p` estão contidos em um elemento `main`. Também criamos um elemento `p` separado para cada link, pois queremos que os elementos apareçam como blocos (um em cada linha). Sem nenhuma estilização, os elementos serão empilhados, um em cima do outro, verticalmente. A aparência será semelhante à da figura 3.7.

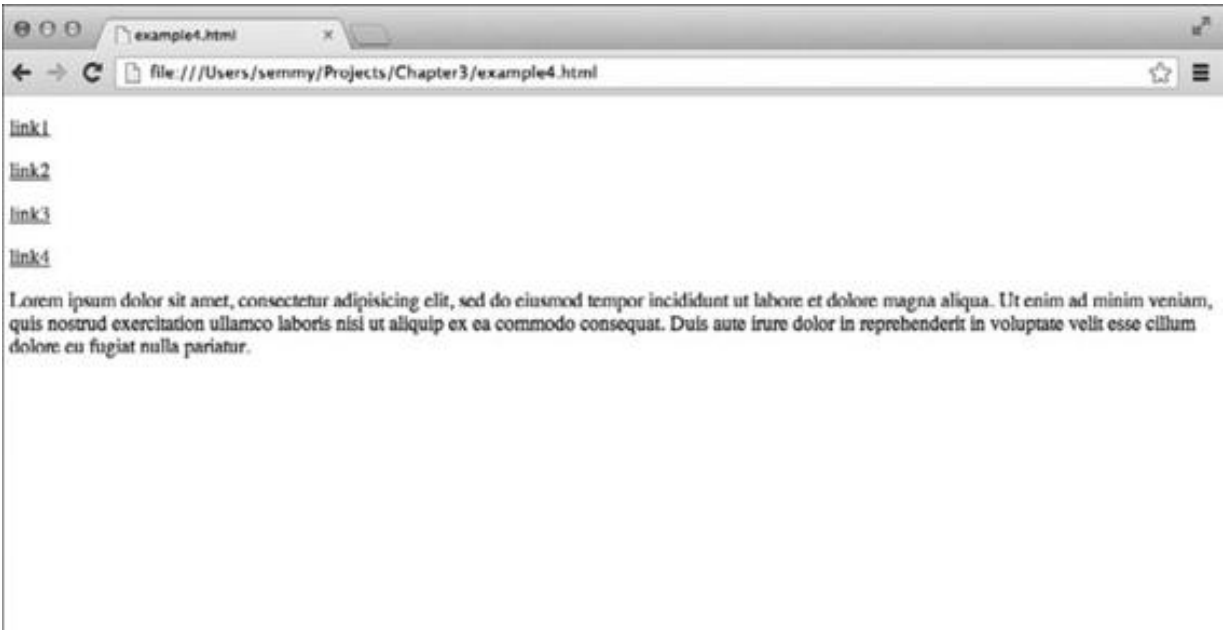
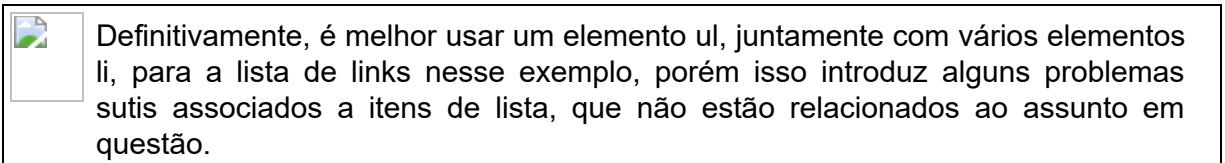


Figura 3.7 – A página renderizada antes da aplicação do CSS.



Agora suponha que tenhamos aplicado o CSS a seguir a esse HTML:

```
main {
  width: 500px;
  margin: auto;
  background: gray;
}
nav {
  /* remova o comentário da próx. linha para obter uma borda em torno de nav */
  /* border: 3px solid black; */
  width: 200px;
  float: right;
}
p.content {
  margin: 0; /* define a margem default como 0 */
}
```

O resultado deve ter um aspecto semelhante ao da figura 3.8.

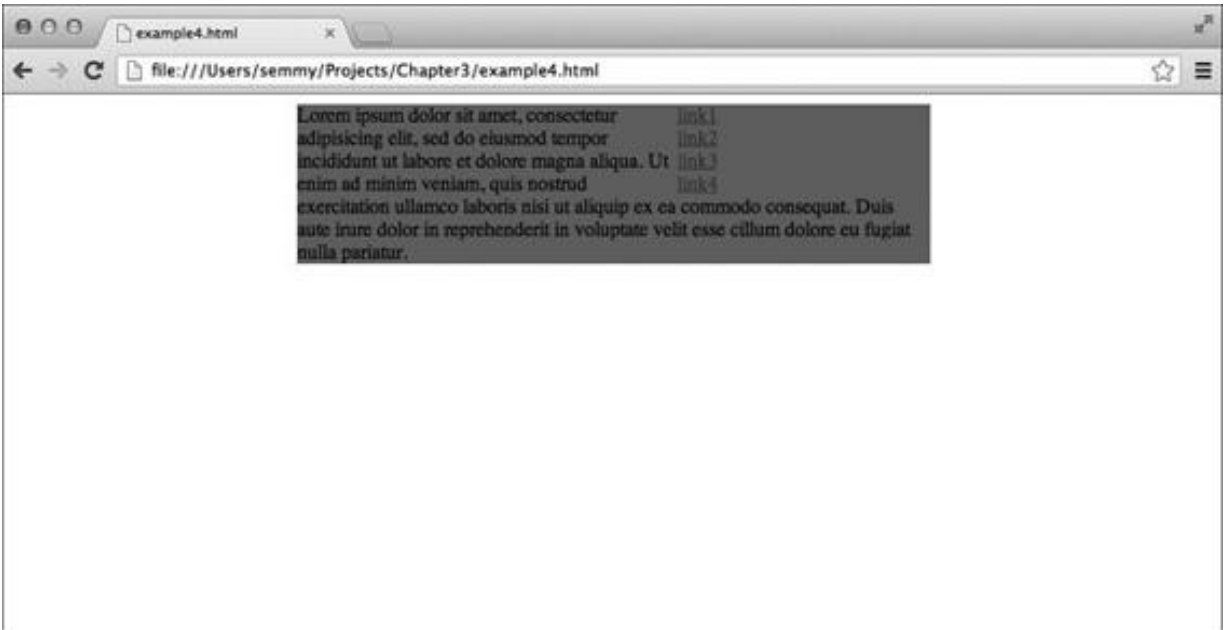


Figura 3.8 – Um exemplo de um elemento flutuando à direita.

Observe que configuramos o elemento `nav` com 200 pixels de largura e o fizemos flutuar à direita para criar um layout de barra lateral. Veja como o elemento `nav` é removido do layout-padrão em forma de pilha e é deslocado para a direita. Então o conteúdo do elemento `p` se espalha em torno da `div nav`. Também incluí uma linha cujo comentário pode ser removido para acrescentar uma borda visível em torno do elemento flutuante. Experimente fazer isso!

Embora simplesmente fazer com que um elemento flutue para a direita normalmente funcione bem para imagens ou para outros elementos inline que precisem de texto ao redor deles, com frequência vamos querer criar um layout do tipo grade com duas colunas. Isso nos coloca diante de um desafio um pouco mais interessante. Nesse caso, queremos fazer com que o elemento `p` flutue para a esquerda e garantir que o tamanho total dos dois elementos não ultrapasse o tamanho do contêiner. Aqui está um trecho de CSS que executará essa tarefa:

```
main {  
  width: 500px;  
  margin: auto;  
  background: gray;
```

```

}
nav {
  width: 100px;
  float: right;
}
/* remove os defaults dos elementos p que estão em nav */
nav p {
  margin: 0;
  padding: 0;
}
p.content {
  margin: 0; /* remove a margem default de p */
  float: left;
  width: 400px;
}

```

Agora se dermos uma olhada nisso no navegador, veremos que as duas colunas se alinham corretamente, porém nossa cor de fundo cinza se foi. Isso ocorre porque, quando todos os elementos contidos em um elemento flutuam, a altura do elemento que os contém torna-se igual a 0. Há uma correção simples para esse caso – podemos definir a propriedade `overflow` da `div` que contém o elemento como `auto`:

```

main {
  width: 500px;
  margin: auto;
  background: gray;
  overflow: auto;
}

```

Isso resulta em um layout semelhante àquele mostrado na figura 3.9.

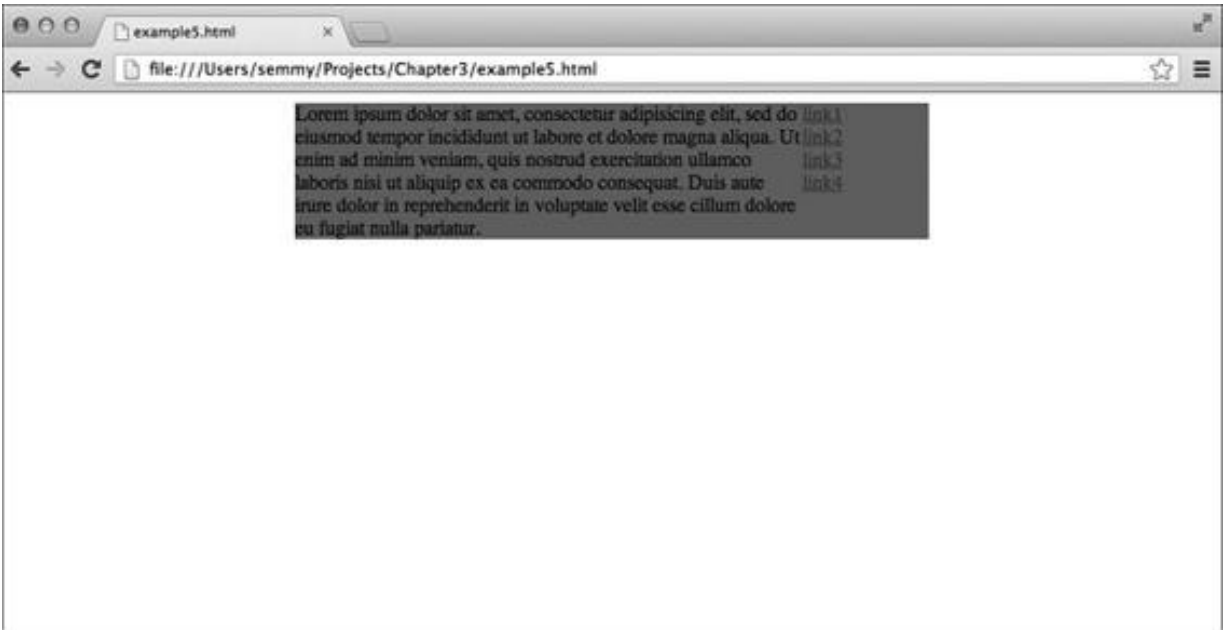


Figura 3.9 – Um layout simples de duas colunas usando floats.

Observe que definimos a soma das larguras dos elementos da esquerda e da direita nesse exemplo para 500 pixels, que é exatamente a largura da div que os contém. Isso pode causar problemas se adicionarmos padding, margem ou borda a qualquer um dos elementos. Por exemplo, podemos querer deslocar o texto do elemento `p` para longe das bordas do elemento. Isso exigirá a adição de padding. Porém, se adicionarmos 20 pixels de padding ao elemento, o resultado será algo semelhante ao que está sendo mostrado na figura 3.10.

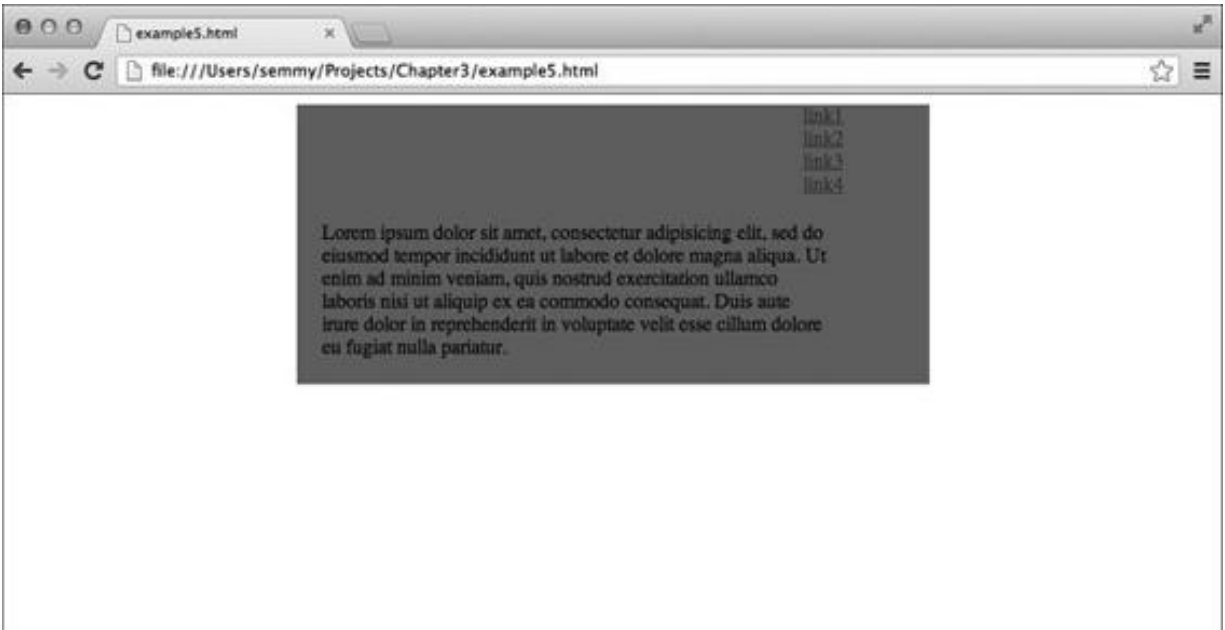


Figura 3.10 – A adição de padding desestruturou o nosso layout.

Isso ocorre porque a soma dos pixels do elemento `main` é maior que a sua largura. Podemos corrigir isso subtraindo o padding duas vezes (pois há um padding de 10 pixels tanto à direita quanto à esquerda) da largura do elemento `p`. O CSS final terá um aspecto semelhante a:

```
main {  
  width: 500px;  
  margin: auto;  
  background: gray;  
  overflow: auto;  
}  
nav {  
  width: 100px;  
  float: right;  
}  
p.content {  
  margin: 0; /* remove a margem default de p */  
  padding: 10px;  
  float: left;  
  width: 380px; /* 400 - 2*10 = 380 */  
}
```

Esse mesmo truque também deve ser aplicado sempre que

adicionarmos uma borda ou uma margem diferente de zero aos elementos.

Propriedade clear

Um problema interessante pode surgir quando criamos layouts que utilizam elementos flutuantes. Aqui está um documento HTML um pouco diferente. Dessa vez, nosso objetivo é definir a navegação do lado esquerdo com um rodapé abaixo de ambas as colunas:

```
<body>
  <nav>
    <p><a href="link1">link1</a></p>
    <p><a href="link2">link2</a></p>
    <p><a href="link3">link3</a></p>
    <p><a href="link4">link4</a></p>
  </nav>
  <main>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
      eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
      ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
      aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit
      in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
    </p>
  </main>
  <footer>
    <p>This is the footer</p>
  </footer>
</body>
```

O CSS a seguir faz o elemento `nav` flutuar para a esquerda e zera as margens e os paddings default usando o seletor *universal*, que seleciona todos os elementos do DOM. A cor de fundo dos elementos também está sendo definida com vários tons distintos de cinza:

```
* {
  margin: 0;
  padding: 0;
}
```

```

nav {
  float: left;
  background:darkgray;
}

main {
  background:lightgray;
}

footer {
  background:gray;
}

```

Quando o documento for renderizado usando essa folha de estilo, veremos algo semelhante à figura 3.11.

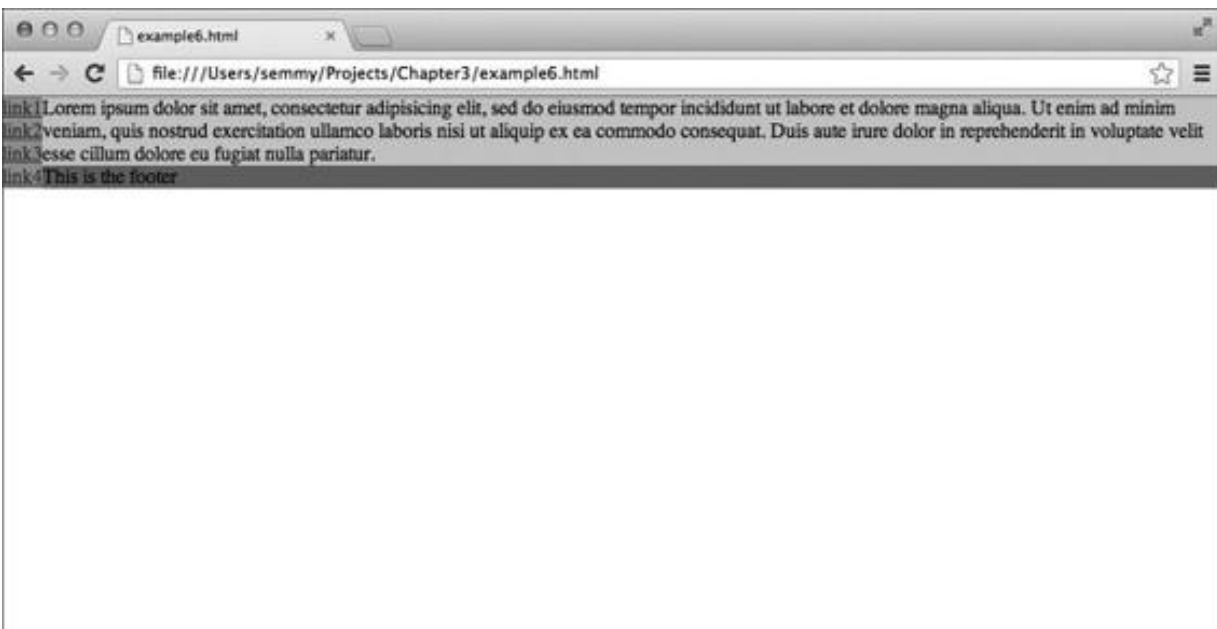


Figura 3.11 – Observe que o rodapé está abaixo da seção principal, em vez de estar abaixo de tudo.

Note que o rodapé, na realidade, está à direita do layout, abaixo da seção principal. Queremos que o rodapé fique abaixo de ambos os elementos. É isso o que a propriedade `clear` faz por nós. Podemos forçar o elemento referente ao rodapé para que fique abaixo de um elemento flutuante à direita ou à esquerda, ou abaixo de ambos os elementos flutuantes, à direita ou à esquerda, por meio da especificação da propriedade `clear`. Desse modo, podemos alterar o CSS para o rodapé da seguinte maneira:

```
footer {  
    background:gray;  
    clear: both; /* ou podemos simplesmente usar 'clear: left', nesse caso */  
}
```

E isso será renderizado conforme mostrado na figura 3.12.

Agora o rodapé está abaixo do elemento flutuante, como desejado.

A flutuação tende a ser um dos aspectos mais confusos do CSS para os iniciantes, portanto sugiro que você invista algum tempo realizando experiências e criando layouts.

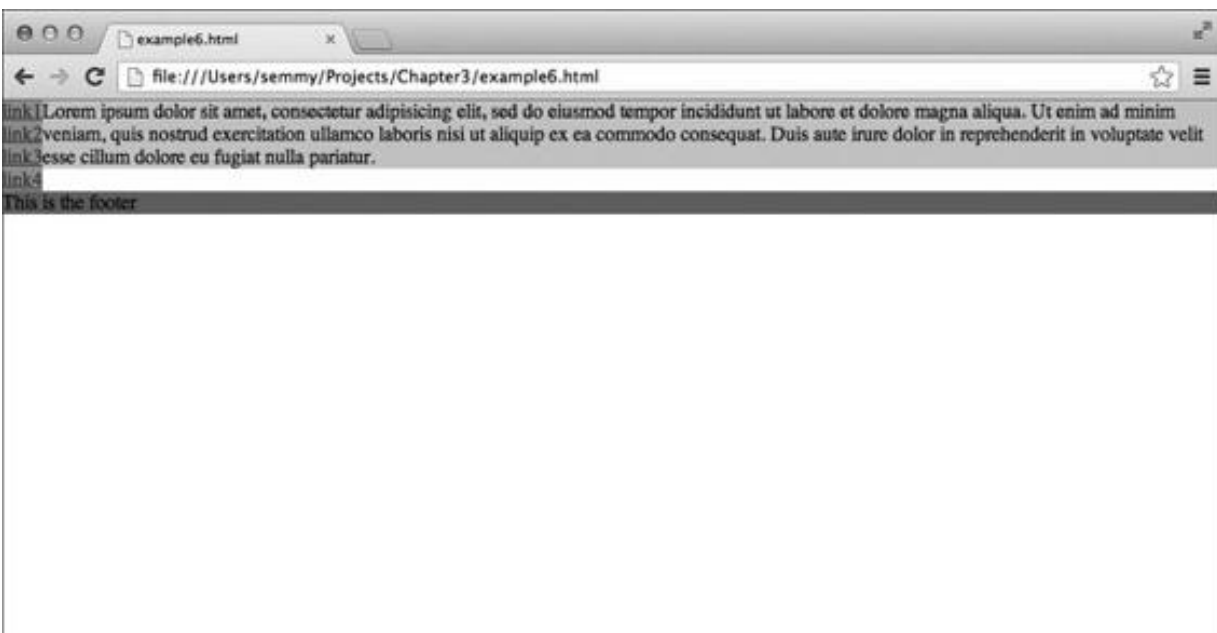


Figura 3.12 – Corrigimos o layout ao definir a propriedade clear no rodapé.

Trabalhando com fontes

No passado, trabalhar com fontes personalizadas em sites era um problema porque ficávamos limitados às fontes que estivessem instaladas no computador de quem visualizasse o nosso site. As fontes web facilitaram bastante a situação, pois você pode disponibilizá-las pela Internet quando forem necessárias. O Google atualmente hospeda diversas fontes personalizadas e faz com que a sua utilização seja muito fácil.

Podemos começar acessando a página inicial do Google Fonts (<http://www.google.com/fonts>), como mostrado na figura 3.13. Nesse local, podemos ver uma lista de fontes pelas quais podemos efetuar rolagens.

Nesse exemplo, usaremos uma fonte chamada *Denk One*, que era a primeira fonte da lista quando acessei a página inicial do Google Fonts. Para usar essa fonte, clique no botão Quick-use, que corresponde a um dos dois botões imediatamente à esquerda do botão grande chamado Add to Collection (Adicionar à coleção).

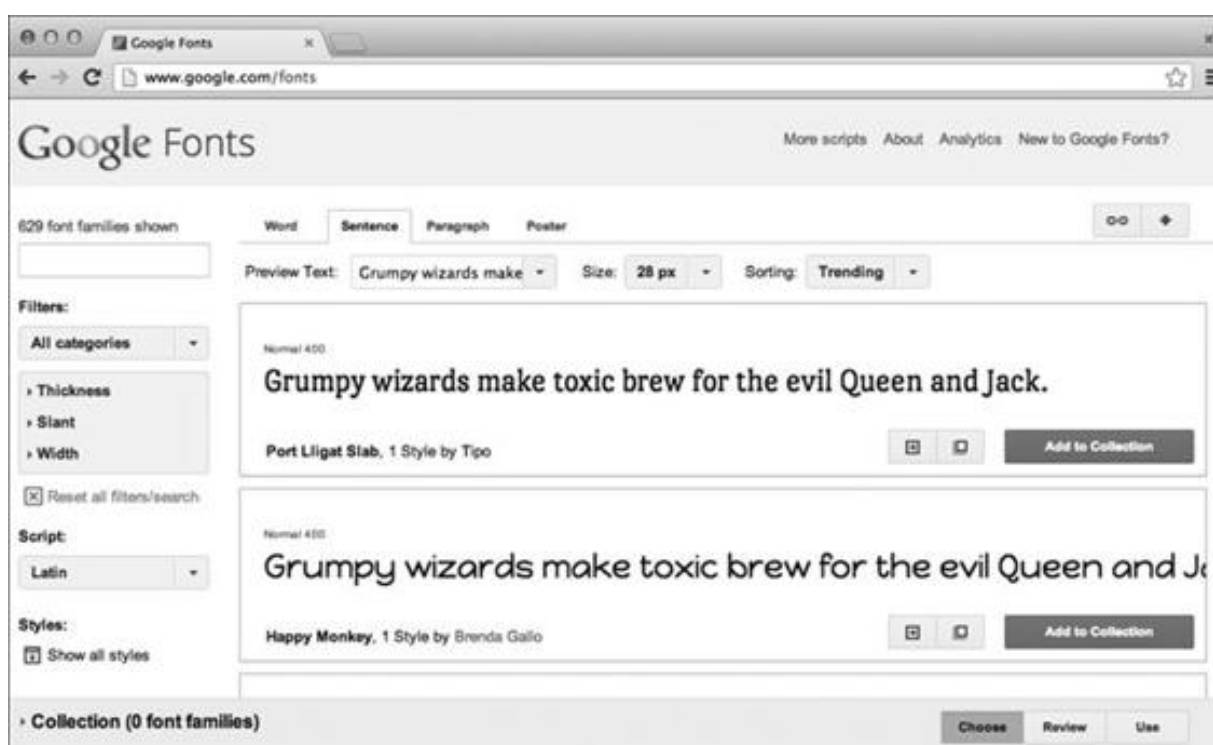


Figura 3.13 – A página inicial do Google Fonts.

Isso fará com que instruções sobre como usar a fonte em nossa página sejam apresentadas. Devemos adicionar dois itens: inicialmente, é necessário acrescentar uma tag `<link>` na seção head de nosso HTML. Portanto, se estivermos utilizando o mesmo exemplo anterior, podemos recortar e colar a tag `<link>` especificada na página acessada por meio do botão Quick-use, ou podemos simplesmente adicionar uma linha semelhante a esta:

`<head>`

```
<title>Amazeriffic</title>
<!-- Há uma quebra de linha abaixo para melhorar a legibilidade -->
<link href="http://fonts.googleapis.com/css?family=Denk+One"
      rel="stylesheet" type="text/css">
<link href="style.css" rel="stylesheet" type="text/css">
</head>
```



O HTML permite tanto aspas simples (') quanto aspas duplas (") para delimitar strings. Procurei usar aspas duplas de forma consistente ao longo deste livro, porém o código copiado e colado do Google utiliza aspas simples. Como as aspas são intercambiáveis, não importa se você as deixar como estão ou se você alterá-las para aspas duplas.

Isso fará com que a fonte web se torne disponível em nossa página. A seguir, podemos usá-la em nosso arquivo *style.css*. Por exemplo, se quisermos estilizar o nosso elemento `h1` com a fonte Denk One, devemos adicionar uma propriedade `font-family` ao nosso conjunto de regras:

```
h1 {
  font-family: 'Denk One', sans-serif;
}
```

A primeira parte da propriedade especifica a fonte (que deve ser carregada por meio da tag `<link>`), e a segunda especifica a fonte a ser utilizada, caso a primeira não esteja disponível. Por exemplo, essa linha diz: “use a fonte Denk One se ela estiver disponível; porém, se não estiver, use a fonte sans serif default instalada no sistema do usuário”.

Agora que podemos apresentar fontes facilmente e alterar suas cores (por meio da propriedade `color` do elemento associado), como podemos alterar o seu tamanho? No passado, havia diversas maneiras de fazer isso, porém há uma que normalmente é considerada como a melhor prática.

Em primeiro lugar, devemos considerar que o usuário pode alterar o tamanho default de seu texto no navegador, e nossas páginas devem respeitar isso. Sendo assim, no conjunto de regras associadas a `body` para o documento, devemos definir um tamanho básico de fonte e então fazer a escala de todas as demais fontes em

relação a essa. Felizmente, o CSS torna essa tarefa bem simples ao utilizar unidades *em* para especificar o tamanho:

```
body {  
  font-size: 100%; /* isso define o tamanho base da fonte para tudo */  
}  
h1 {  
  font-size: xx-large; /* faz a definição em relação ao tamanho base da  
    fonte */  
}  
h2 {  
  font-size: x-large;  
}  
.important {  
  font-size: larger; /* faz com que o tamanho seja um pouco maior que o do pai */  
}  
.onePointTwo {  
  font-size: 1.2em; /* configura para 1,2 vezes o tamanho da base */  
}
```

Nesse exemplo, definimos o tamanho-base da fonte para 100% do tamanho da fonte utilizada pelo usuário em seu navegador e, em seguida, fazemos a escala de todos os tamanhos de fonte em relação a ele. Podemos usar os valores absolutos de tamanho do CSS, `xxlarge` e `x-large` (ou, de modo semelhante, `x-small` ou `xx-small`), que farão com que o tamanho seja escalado de acordo com essas configurações. Também podemos usar tamanhos relativos como `larger` ou `smaller` para fazer com que o tamanho da fonte seja relativo ao contexto atual.

Se precisarmos de um controle mais preciso sobre os tamanhos das fontes, devemos usar *ems*, que são metaunidades que representam um multiplicador para o tamanho atual da fonte. Por exemplo, se definirmos o tamanho-base de nossa fonte no corpo com um valor específico, por exemplo, 12pt, então definir o tamanho da fonte de outro elemento com 1.5em fará com que o seu tamanho seja de 18pt. Isso é útil porque os tamanhos da fonte irão escalar de acordo com o tamanho-base da fonte, o que significa que, para deixar o texto

maior ou menor na página, basta alterar o tamanho-base. Isso também é prático para os usuários com deficiência visual: normalmente, eles irão definir um tamanho-base de fonte maior para o navegador, portanto usar em no lugar de valores explícitos fará com que a página seja escalada apropriadamente.

No exemplo anterior, definimos o tamanho da fonte do parágrafo para 1,2 vez o tamanho-base. A figura 3.14 ilustra de que modo os conjuntos anteriores de regras estilizam um documento.

h1: xx-large

h2: x-large

p.important larger

p: 1.2 em

p: base size

Figura 3.14 – Alguns exemplos de fontes baseados na folha de estilo anterior.

Eliminando as inconsistências dos navegadores

Uma ferramenta controversa que você verá ser usada com frequência é o CSS reset.

Você se lembra de nosso exemplo de flutuação, em que removemos a margem e o padding default das tags de parágrafo? Acontece que vários navegadores implementam suas configurações básicas de CSS de maneira um pouco diferente, o que pode resultar na renderização de seu CSS de maneira levemente distinta em navegadores diferentes. Um CSS *reset* foi concebido para remover todos os defaults dos navegadores. O reset mais famoso é o de Eric Meyer (<http://meyerweb.com/eric/tools/css/reset/>).

Por que os CSS resets são controversos? Existem vários motivos,

porém eles podem ser classificados em relação a três pontos principais. Um deles refere-se à acessibilidade – por exemplo, os resets podem causar problemas às pessoas que estiverem navegando por meio do teclado. Outra crítica refere-se ao desempenho. Como o seletor universal (*) normalmente é utilizado, os CSS resets podem ser relativamente ineficientes. E, por fim, os CSS resets criam um volume enorme de redundância porque os defaults do navegador normalmente têm exatamente os valores que você deseja.

Por outro lado, acho que os resets são excelentes do ponto de vista pedagógico e é por isso que estou incluindo-os aqui. Eles forçam os iniciantes a definir explicitamente a maneira como eles querem que determinados aspectos da página sejam apresentados, em vez de contar com os defaults do navegador. Como, supostamente, você está lendo este livro porque está começando a trabalhar com o CSS, sugiro que você utilize um.

Para tornar a situação mais explícita, incluí o meu reset como uma folha de estilo totalmente separada e adicionei um elemento `link` a mais em meu HTML. Podemos copiar o reset de Eric Meyer em outro arquivo CSS chamado *reset.css* e efetuar o seu link a partir do HTML:

```
<head>
  <title>Amazeriffic</title>
  <link rel="stylesheet" type="text/css" href="reset.css">
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
```

É muito fácil perceber o efeito dramático que um reset pode provocar em uma página. Considere o HTML a seguir:

```
<body>
  <h1>This is a header</h1>
  <h3>This is a slightly less important header</h3>
  <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
    tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
    veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
    commodo consequat.</p>
```

```
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod  
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim  
veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea  
commodo consequat.</p>
```

```
</body>
```

A figura 3.15 mostra a página renderizada no Chrome, somente com a estilização default do navegador sendo aplicada.

Observe como os elementos `h1` e `h3` já possuem fontes e margens estilizadas. Da mesma maneira, os elementos `p` também possuem margens.

Agora acrescentaremos o reset. Eu copiei o arquivo de reset da página de Eric Meyer e o salvei como *reset.css* no mesmo diretório em que está o meu HTML. Modifiquei a tag `<head>` em meu HTML para que se parecesse com:

```
<head>  
  <title>Example 8 -- Using a CSS Reset</title>  
  <link href="reset.css" rel="stylesheet" type="text/css">  
</head>
```

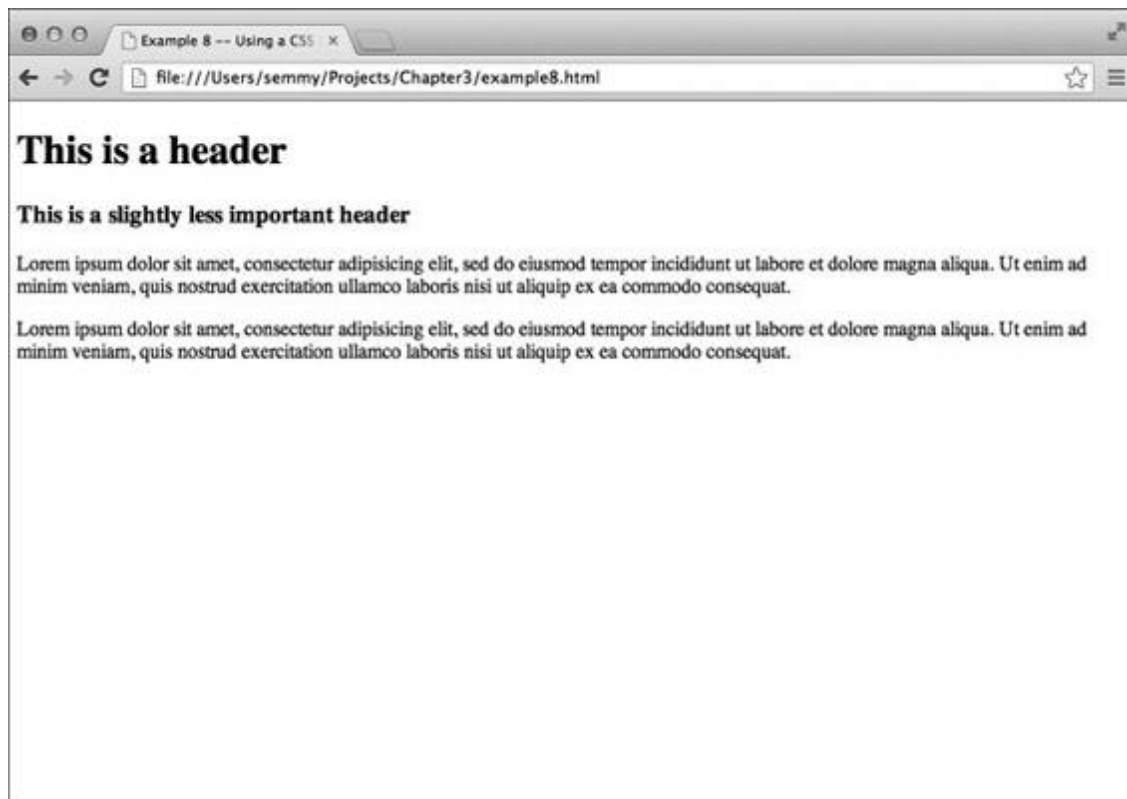


Figura 3.15 – Uma página simples com um pouco de estilização default.

Agora, quando eu recarregar a página, ela terá o aspecto apresentado na figura 3.16.

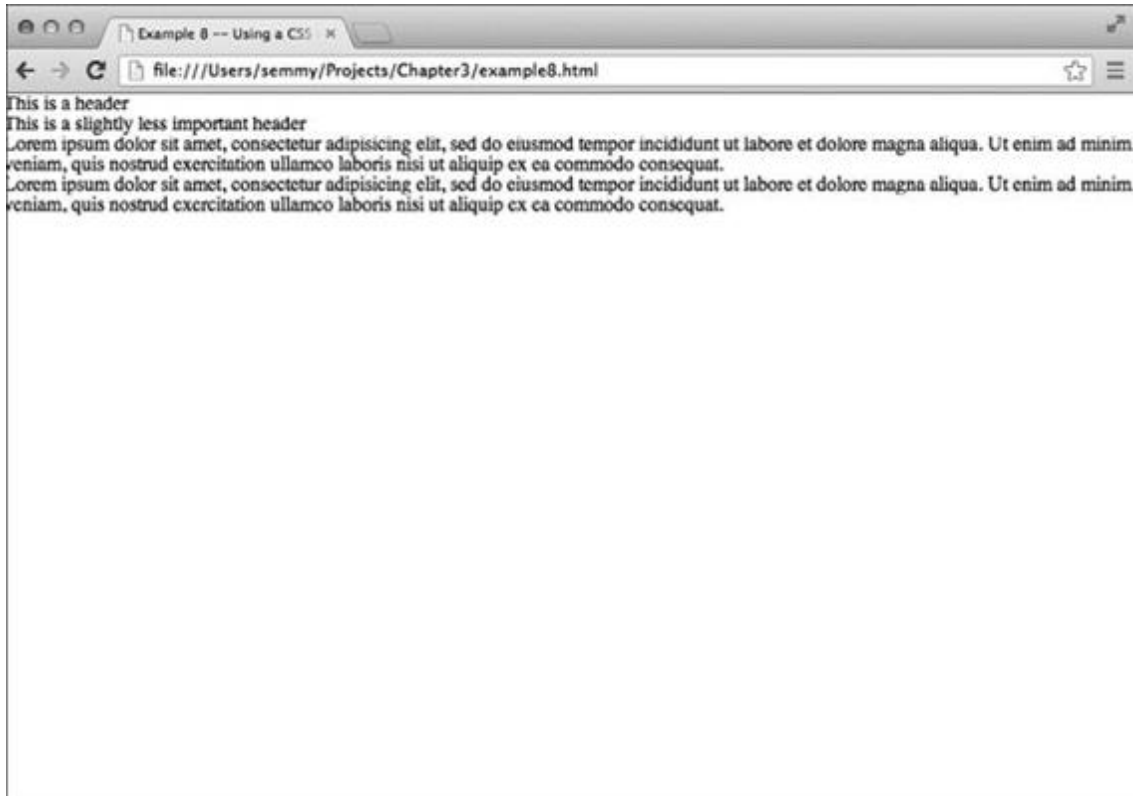


Figura 3.16 – Uma página simples com toda a estilização default removida por meio de um reset.

As margens, o padding e os tamanhos default das fontes foram totalmente removidos da página. É claro que não queremos que o nosso site tenha essa aparência, porém a questão é que incluir um arquivo de CSS reset que remova a estilização default do navegador permite que as regras que fornecermos em nosso próprio arquivo CSS tenham sempre o mesmo efeito, independentemente do navegador (e de sua opção correspondente de regras default) que o usuário possa estar utilizando.

Usando o CSS Lint para identificar

problemas em potencial

Assim como ocorre com o HTML, às vezes é útil ter uma ferramenta que nos ajude a identificar possíveis problemas em nosso CSS. Por exemplo, você consegue encontrar o erro nesse CSS de exemplo?

```
body {  
  background: lightblue;  
  width: 855px;  
  margin: auto;  
  
h1 {  
  background: black;  
  color: white;  
}  
  
p {  
  color: red;  
  margin: 10px  
  padding: 20px;  
}
```

Se você encontrou um erro, então você está enganado! Você deveria ter encontrado dois erros! Acho que o erro mais evidente é a falta de uma chave de fechamento no conjunto de regras para `body` na parte inferior, antes do início do conjunto de regras para `h1`. Mas existe ainda outro erro.

Você provavelmente deve tê-lo percebido após ter observado as regras pela segunda vez. Mas, se isso não aconteceu, o erro está no conjunto de regras para o elemento `p`. Está faltando ponto e vírgula após a propriedade `margin`.

Semelhante a uma ferramenta de validação de HTML, o CSS Lint é uma ferramenta que nos ajudará a identificar possíveis problemas em nosso código. E, assim como ocorre com a ferramenta de validação de HTML online do W3C, não é necessário instalar nenhum software para usá-lo; basta acessar <http://csslint.net> (como mostrado na figura 3.17) e recortar e colar o nosso CSS.

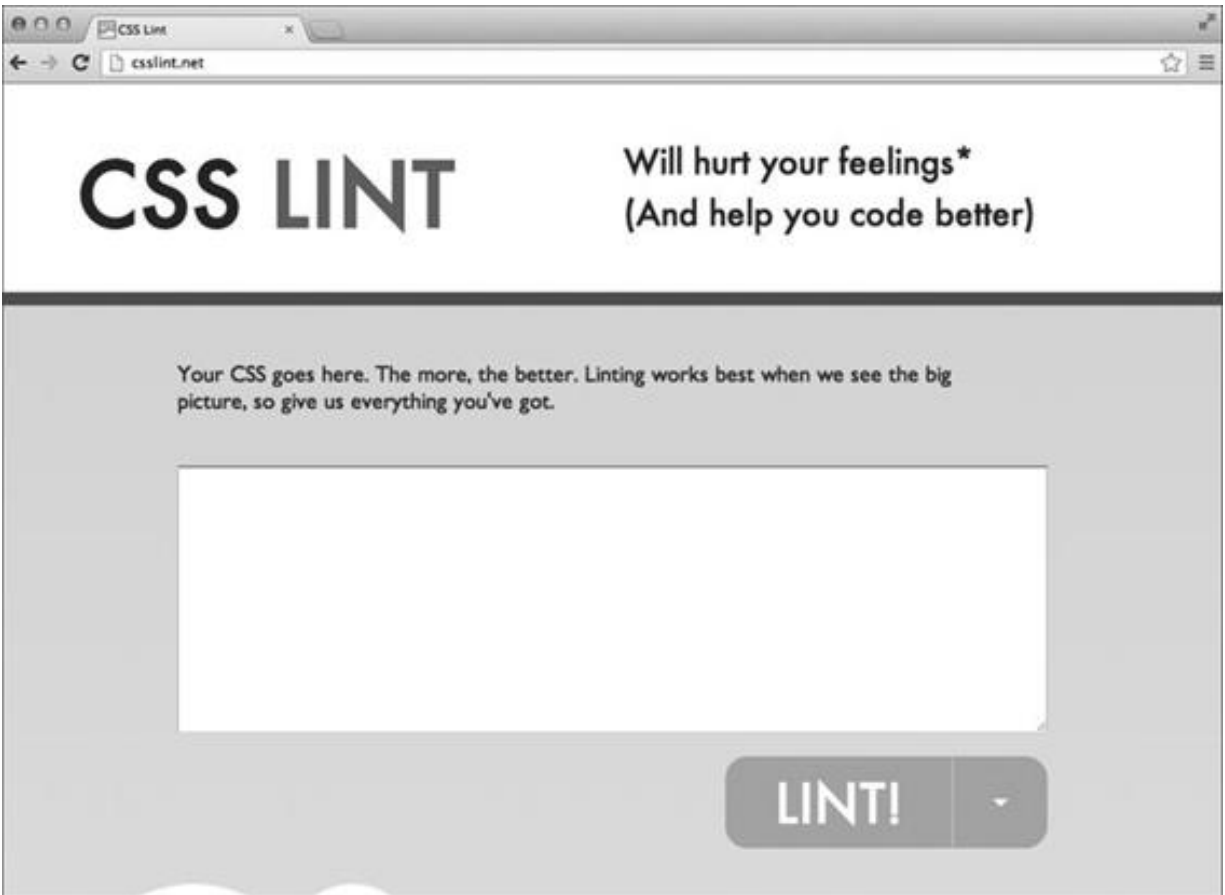


Figura 3.17 – A página inicial do CSS Lint.

O CSS Lint apresenta uma quantidade muito maior de warnings (avisos) do que a ferramenta de validação de HTML que estudamos no capítulo 2, porém há opções que permitem personalizar o nível dos warnings. Essas opções podem ser vistas ao clicar na seta para baixo, bem ao lado do botão Lint! na página principal. As opções são classificadas de acordo com os motivos dos warnings. Por exemplo, “Disallow universal selector” (Não permitir o seletor universal) está na categoria de Performance (Desempenho). Isso significa que, se esse recurso CSS em particular for utilizado, sua página poderá ser renderizada mais lentamente, portanto o CSS Lint avisará você a esse respeito.

Normalmente, deixo todas as opções selecionadas no CSS Lint, mas você pode querer ser um pouco mais flexível. Sugiro deixar todas as opções selecionadas e, à medida que você receber os warnings, reserve um tempo para fazer algumas pesquisas no

Google e descobrir por que o CSS Lint está gerando essas advertências. É quase como ter um especialista em CSS sentado logo ali ao seu lado!

Interagindo e resolvendo problemas com as ferramentas para desenvolvedores do Chrome

Como ocorre com qualquer tipo de desenvolvimento de software, às vezes, nem sempre tudo funcionará do modo como você gostaria. Com o CSS, geralmente significa que a sua página terá um aspecto diferente do que você esperava no navegador, e o motivo para isso acontecer não estará bem claro. Felizmente, o navegador Chrome inclui um bom conjunto de ferramentas que pode ajudar na resolução de problemas de seu CSS e de seu HTML.

Vamos começar abrindo o arquivo *margin_border_padding.html* no navegador Chrome. A seguir, acesse o menu View (Visualizar) no Chrome, selecione o submenu Developer (Desenvolvedor) e clique na opção Developer Tools (Ferramentas do desenvolvedor). Uma janela Developer Tools do Chrome será aberta na parte inferior da janela de seu navegador. Caso ainda não esteja selecionada, clique na aba Elements (Elementos) na parte superior da janela Developer Tools e você verá algo semelhante ao que está sendo apresentado na figura 3.18.

Do lado esquerdo, você verá uma porção de código HTML, juntamente com setas que podem mostrar/ocultar os elementos descendentes. Se a seta estiver apontando para a direita, isso significa que você poderá clicá-la para que o HTML contido nesse elemento seja mostrado. De modo semelhante, se a seta estiver apontando para baixo, ela pode ser clicada para ocultar o HTML contido.

À medida que você passar o cursor sobre os elementos HTML, os elementos poderão ser vistos em destaque na parte superior da

janela, na seção de apresentação. Um elemento HTML específico pode ser selecionado ao clicar em sua tag de abertura e sua linha associada aparecerá então destacada.

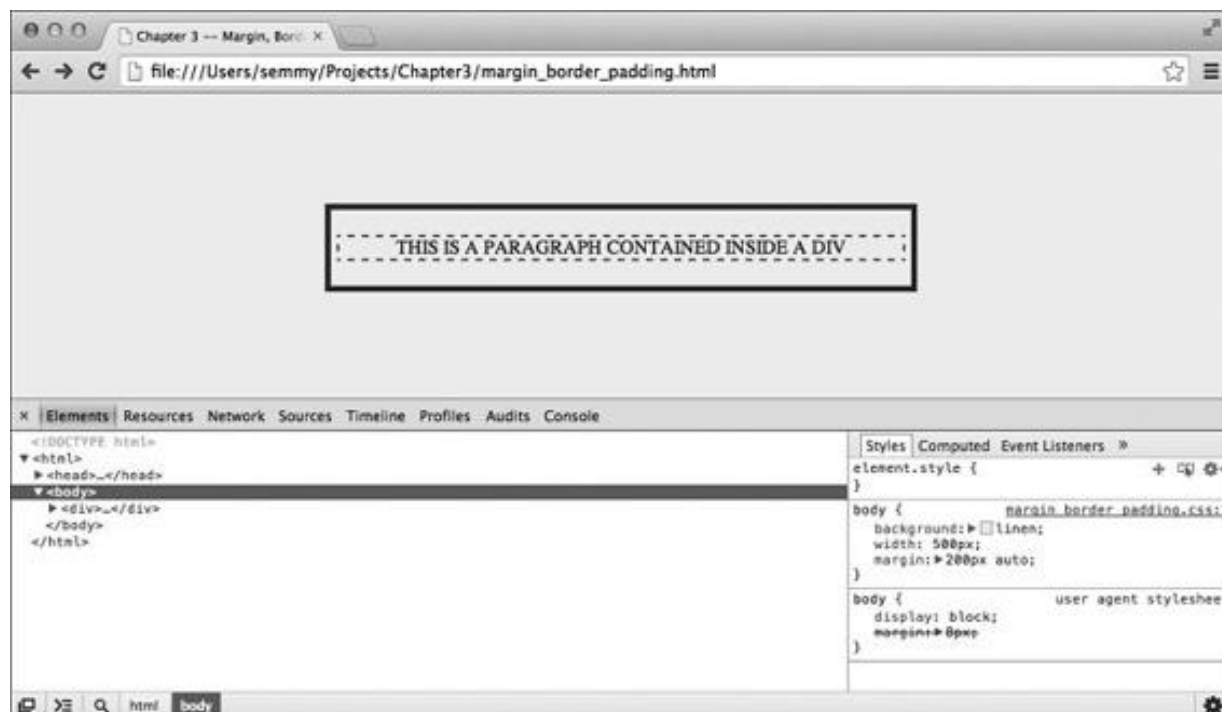


Figura 3.18 – O Developer Tools (Ferramentas do desenvolvedor) do Chrome aberto com o exemplo `margin_border_padding.html`.

O HTML visto na aba Elements pode ou não ser exatamente igual ao código HTML que está em seu arquivo HTML, mas, mesmo que não seja, é o HTML que o Chrome estará apresentando no momento. Como já mencionei no capítulo 2, um navegador, com frequência, irá fazer inferências a respeito de seu código (como, por exemplo, supor as tags de fechamento), e o HTML apresentado na aba Elements inclui todas as suposições feitas pelo Chrome. Por exemplo, se você criar um documento HTML que contenha somente uma tag `<p>` (e não uma tag `<html>`, `<header>` ou `<body>`), o Chrome irá inserir essas tags por você, e elas poderão ser vistas na aba Elements, como mostrado na figura 3.19. Essa pode ser uma maneira útil de reduzir a distância às vezes confusa entre o que você realmente pretende e o que o Chrome acha que seja a sua intenção.

Do lado direito da janela, você pode ver o CSS associado ao elemento HTML selecionado. Vá em frente e selecione o elemento `div` que contém o parágrafo. À direita, você verá o conjunto de regras associado a ele na janela Matched CSS Rules (Regras CSS correspondentes) e, quando você passar o cursor sobre ele, caixas de seleção surgirão à esquerda de cada regra. Você pode desmarcar a seleção de uma regra para removê-la. Se você se deparar com problemas em seu CSS, normalmente dar uma olhada nesse local para garantir que a regra de estilo esperada está realmente sendo aplicada a um determinado elemento pode ser útil. As regras de estilo podem ser vistas no Developer Tools do Chrome, como mostrado na figura 3.19.

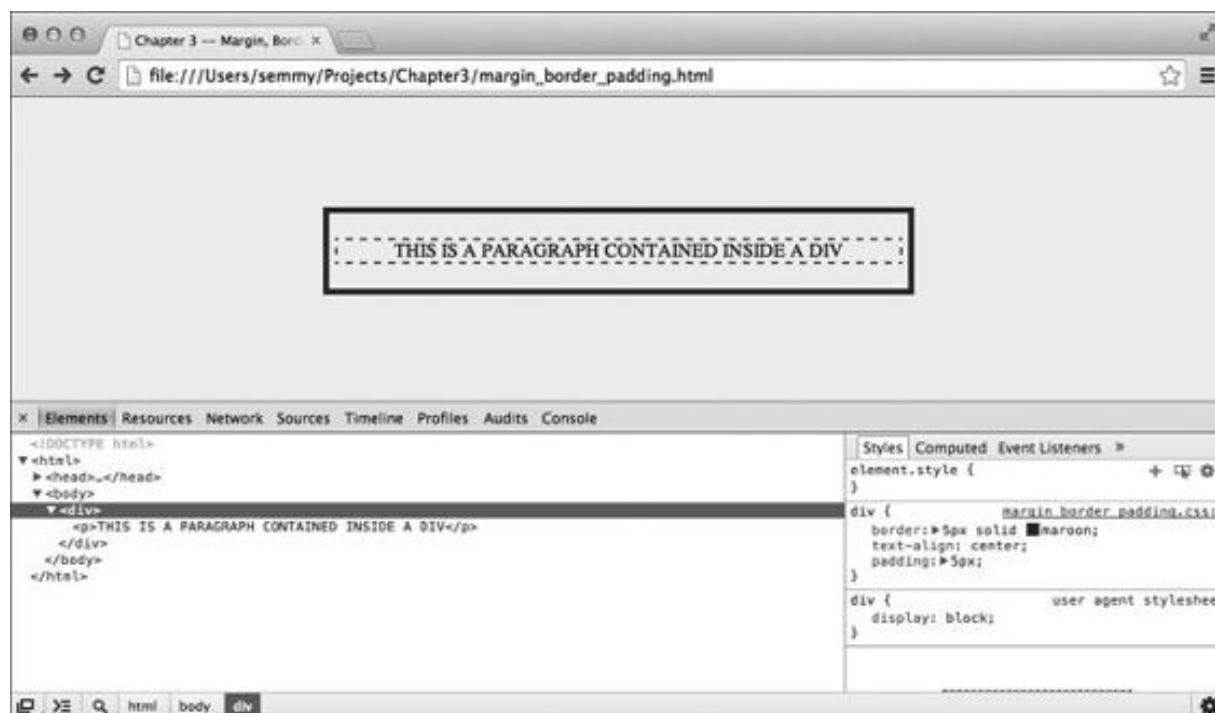


Figura 3.19 – A div principal está selecionada e você pode ver o conjunto de regras de estilo associado à direita.

Porém, melhor ainda, você pode manipular diretamente uma regra ao clicar nela e digitar um novo valor para essa regra. Você pode até mesmo adicionar regras novas aqui, mas tenha em mente que isso não se refletirá em seu arquivo CSS. Portanto, se você encontrar uma solução para um problema de estilização nessa janela, será

necessário retornar ao Sublime para incluir o estilo em seu arquivo CSS.

Você já deve ter feito experimentos com as configurações de padding, borda e margem, modificando o seu código. Experimente fazer manipulações semelhantes, porém, dessa vez, na aba Elements da janela Developer Tools do Chrome. Você verá que é muito mais fácil fazer experimentos e obter feedback imediatamente. Também é importante observar que, ao recarregar a página, todas as suas alterações serão perdidas e o Chrome retornará para a página definida em seus arquivos HTML e CSS.

Estilizando o Amazeriffic

Agora vamos reunir tudo em um só exemplo. No capítulo anterior, desenvolvemos a estrutura para a página inicial de uma aplicação chamada Amazeriffic. O HTML resultante que define a estrutura de nossa página tem o seguinte aspecto:

```
<!doctype html>
<html>
  <head>
    <title>Amazeriffic</title>
  </head>
  <body>
    <header>
      <h1>Amazeriffic</h1>
      <nav>
        <a href="#">Sign Up!</a> |
        <a href="#">FAQ</a> |
        <a href="#">Support</a>
      </nav>
    </header>
    <main>
      <h2>Amazeriffic will change your life!</h2>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
        eiusmod tempor incididunt ut labore et dolore magna aliqua.</p>
      <h3>Here's why you need Amazeriffic</h3>
      <ul>
```

```

    <li>It fits your lifestyle</li>
    <li>It's awesome</li>
    <li>It rocks your world</li>
</ul>


</main>

<footer>
  <div class="contact">
    <h5>Contact Us</h5>
    <p>Amazeriffic!</p>
    <p>555 Fiftieth Street</p>
    <p>Asheville, NC 28801</p>
  </div>

  <div class="sitemap">
    <h5>Sitemap</h5>
    <ul>
      <li><a href="#">Home</a></li>
      <li><a href="#">About Us</a></li>
      <li><a href="#">Privacy</a></li>
      <li><a href="#">Support</a></li>
      <li><a href="#">FAQ</a></li>
      <li><a href="#">Careers</a></li>
    </ul>
  </div>
</footer>
</body>
</html>

```

Inicialmente, entre no diretório que contém o nosso projeto Git para o Amazeriffic. Se estivermos em nosso diretório home, podemos ir diretamente para o diretório *Amazeriffic* usando `cd` e fornecendo o path completo como argumento:

```
hostname $ cd Projects/Amazeriffic
```

Podemos usar o comando `ls` para ver o conteúdo do diretório e, em seguida, executar o comando `git log` para verificar o nosso histórico de commits do capítulo anterior. Os meus comandos apresentam um aspecto deste tipo:

```
hostname $ ls
```

index.html

hostname \$ **git log**

commit efeb5a9a5f80d861119f5761df789f6bde0cda4f

Author: Semmy Purewal <semmy@semmy.me>

Date: Thu May 23 1:41:52 2013 -0400

Add content and structure to footer

commit 09a6ea9730521ed1effd135a243723a2745d3dc5

Author: Semmy Purewal <semmy@semmy.me>

Date: Thu May 23 12:32:17 2013 -0400

Add content to main section

commit f90c9a6bd896d1a303f6c3647a7475d6de9c4f9e

Author: Semmy Purewal <semmy@semmy.me>

Date: Thu May 23 11:45:21 2013 -0400

Add content to header section

commit 1c808e2752d824d815929cb7c170a04267416c04

Author: Semmy Purewal <semmy@semmy.me>

Date: Thu May 23 10:36:47 2013 -0400

Add skeleton of structure to Amazeriffic

commit 147deb5dbb3c935525f351a1154b35cb5b2af824

Author: Semmy Purewal <semmy@semmy.me>

Date: Thu May 23 10:35:43 2013 -0400

Initial commit

A seguir, podemos abrir o diretório no Sublime e criar dois arquivos CSS novos. Inicialmente, criaremos um arquivo *reset.css* com base no reset de Eric Meyer. Você pode digitar o código (conforme descrito anteriormente) ou pode recortar e colar o código a partir do site de Meyer (<http://meyerweb.com/eric/tools/css/reset>). Também criaremos um arquivo *style.css* que incluirá o nosso CSS personalizado.

Antes de efetuarmos o commit, vamos prosseguir e efetuar o link de nossos arquivos CSS a partir de nosso arquivo HTML adicionando duas tags de link na seção head:

```
<head>
```

```
  <title>Amazeriffic</title>
```

```
  <link rel="stylesheet" type="text/css" href="reset.css">
```

```
  <link rel="stylesheet" type="text/css" href="style.css">
```

</head>

Agora podemos abrir o arquivo no Chrome. Ele deverá parecer totalmente sem estilização por causa do arquivo de reset. O meu arquivo tem a aparência mostrada na figura 3.20.

Agora faremos o commit das alterações em nosso repositório Git:

```
hostname $ git add reset.css
hostname $ git add style.css
hostname $ git add index.html
hostname $ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# modified:   index.html
# new file:   reset.css
# new file:   style.css
#
hostname $ git commit -m "Add reset.css and style.css, link in index.html"
[master b9d4bc9] Add reset.css and style.css, link in index.html
3 files changed, 142 insertions(+), 2 deletions(-)
create mode 100644 reset.css
create mode 100644 style.css
```

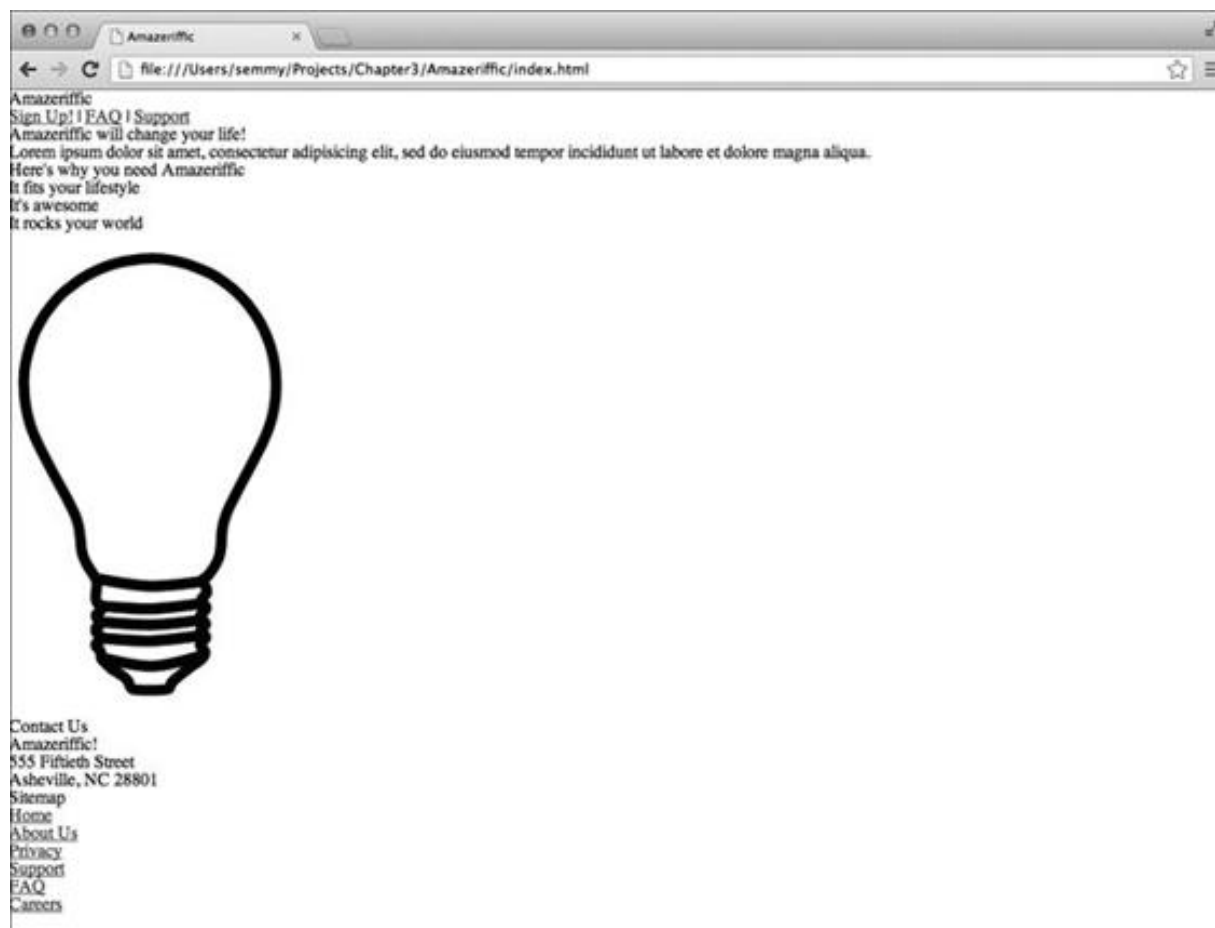


Figura 3.20 – O HTML final do Amazeriffic com um reset incluído.

A grade

Agora vamos analisar o estilo para o Amazeriffic. Observe que o conteúdo está alinhado horizontalmente em duas colunas. A primeira coluna ocupa cerca de dois terços da área do conteúdo, enquanto a segunda ocupa aproximadamente um terço dessa área. A seguir, observe que o conteúdo está alinhado verticalmente em cerca de três linhas. Isso define uma grade de estilização para o conteúdo, que corresponde a uma abordagem básica para efetuar o design dos layouts.

Outro aspecto relativo ao design que é difícil de ser encontrado em um livro refere-se ao fato de o design ter uma largura fixa. Isso significa que, à medida que redimensionarmos a janela de nosso navegador, o conteúdo da página permanecerá centralizado com a

mesma largura. As figuras 3.21 e 3.22 mostram o Amazeriffic em duas janelas de navegador de tamanhos diferentes.

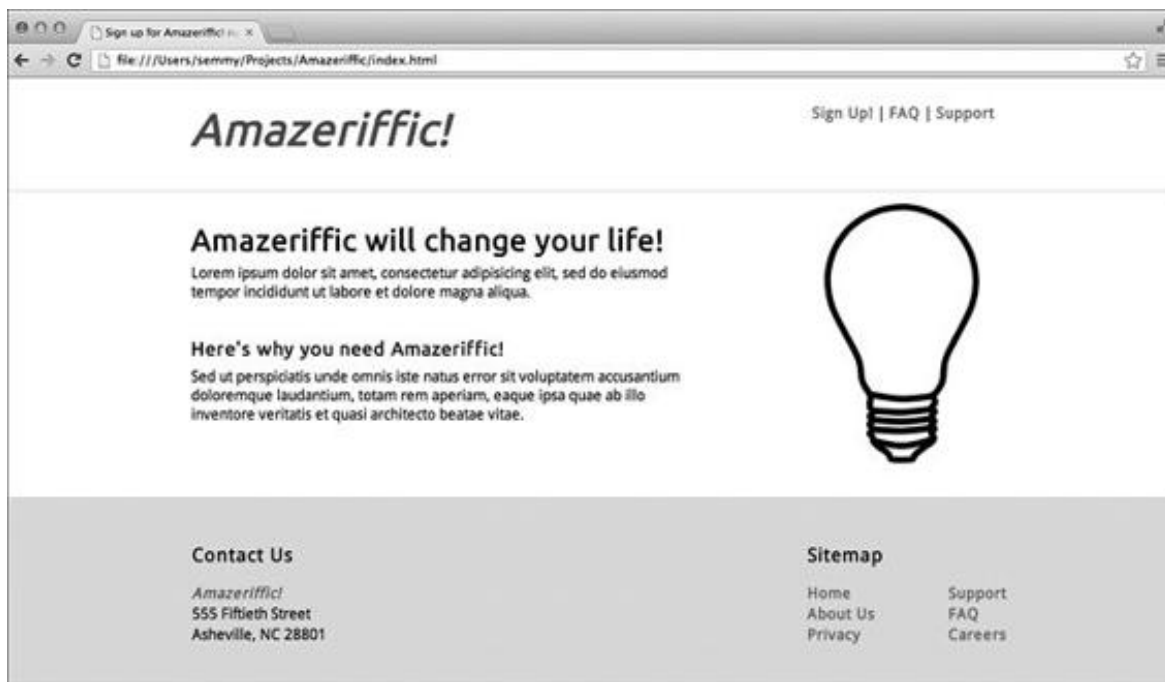


Figura 3.21 – O Amazeriffic em uma janela de navegador de 1.250 × 650.

Isso significa que o design não é *responsivo* – uma abordagem moderna para layouts usando CSS que permite a reorganização do layout por conta própria de acordo com a largura do navegador. Isso é importante ao efetuar um design para a web, pois muitas pessoas visualizarão esse design em um telefone celular ou em um tablet. Não iremos investir muito tempo em design responsivo neste livro, porém você terá a oportunidade de experimentar algumas ferramentas que simplificam o processo de criar um design responsivo nos exercícios que estão no final deste capítulo.

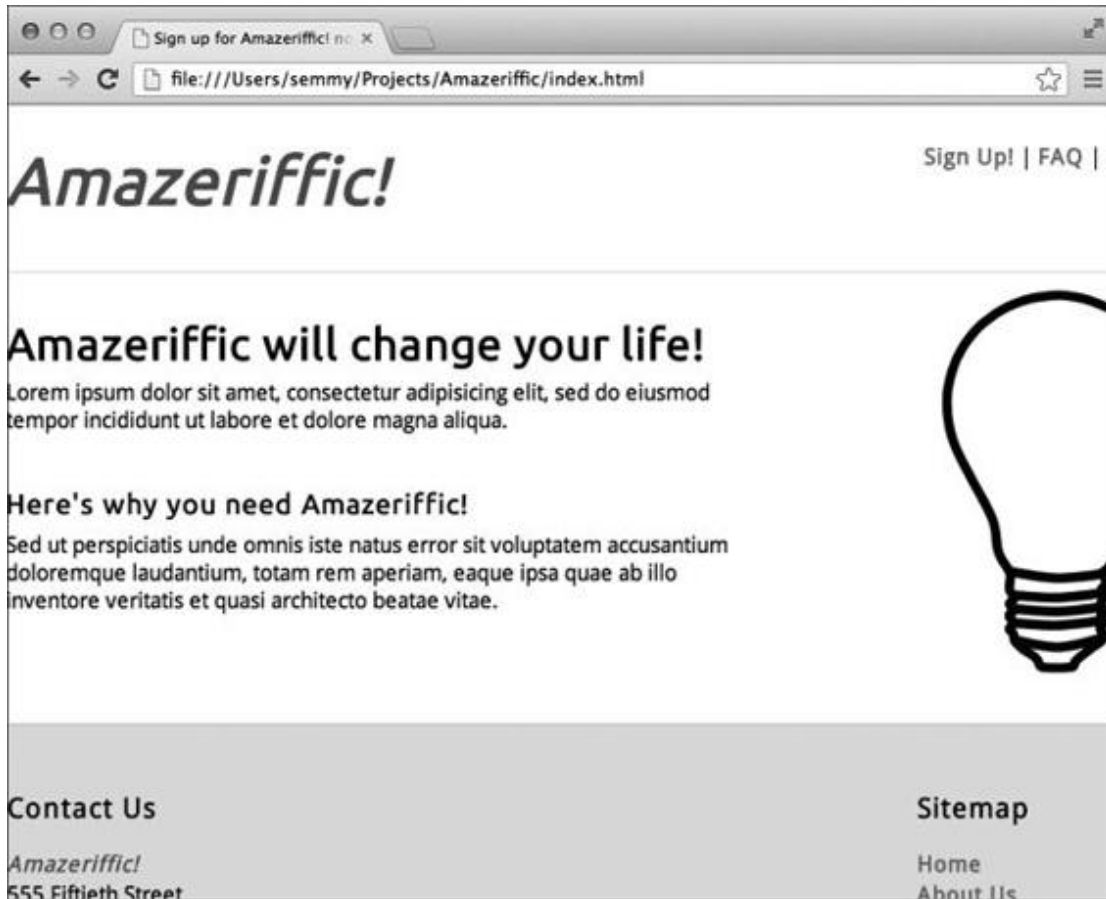


Figura 3.22 – O Amazeriffic em uma janela de navegador de 800 × 575.

Nosso objetivo é criar esse design de largura fixa com duas colunas usando o CSS. Para começar, precisamos criá-lo de modo que as linhas (que, nesse caso, correspondem aos nossos elementos header, main e footer) tenham a propriedade de largura fixa. A primeira tarefa que podemos fazer para nos ajudar nesse caso é criar um conjunto de regras para todos esses elementos. Em nosso arquivo *style.css*, adicionaremos um conjunto de regras como:

```
header, main, footer {  
  min-width: 855px;  
}
```

A propriedade `min-width` faz com que, se o navegador for redimensionado para uma viewport menor do que 855 px, a parte restante da página fique fora do campo de visão. Experimente fazer isso.

A seguir, observe que o rodapé possui um box que deve preencher a parte inferior da página à medida que a janela do navegador for expandida verticalmente. Isso significa que a cor do box deverá ser a cor de fundo de toda a página e que devemos alterar a cor dos elementos `header` e `main` para branco, posteriormente.

Podemos acrescentar uma regra para o corpo que torne a cor de fundo da imagem igual à cor amarela do conjunto de regras associado a `body`. Observe que, apesar de ter adicionado o conjunto anterior de regras, vou adicionar o conjunto de regras para `body` acima dele no arquivo *style.css*. Isso ocorre porque o corpo vem antes dos elementos `header`, `main` e `footer` no arquivo HTML. Manter as localizações (relativamente) consistentes entre esses dois arquivos permite descobrir erros e fazer alterações:

```
body {  
    background: #f9e933;  
}  
  
header, main, footer {  
    min-width: 855px;  
}
```

Agora podemos ver que a página toda apresenta a cor amarela, portanto mudaremos as cores de fundo dos elementos `header` e `main` para branco. Para isso, adicionaremos dois novos conjuntos de regras para os elementos `header` e `main`, além do conjunto que contém a regra `min-width`. Iremos separá-los porque queremos acrescentar regras diferentes de estilização a cada um à medida que seguirmos adiante:

```
header {  
    background: white;  
}  
  
main {  
    background: white;  
}
```

Agora a aparência será semelhante à da figura 3.23.

Na época desta publicação, a tag <code><main></code> não estava sendo renderizada



corretamente no navegador Safari. Se você estiver usando esse navegador e estiver tendo problemas, experimente adicionar um conjunto de regras para main que inclua a regra `display: block;`.

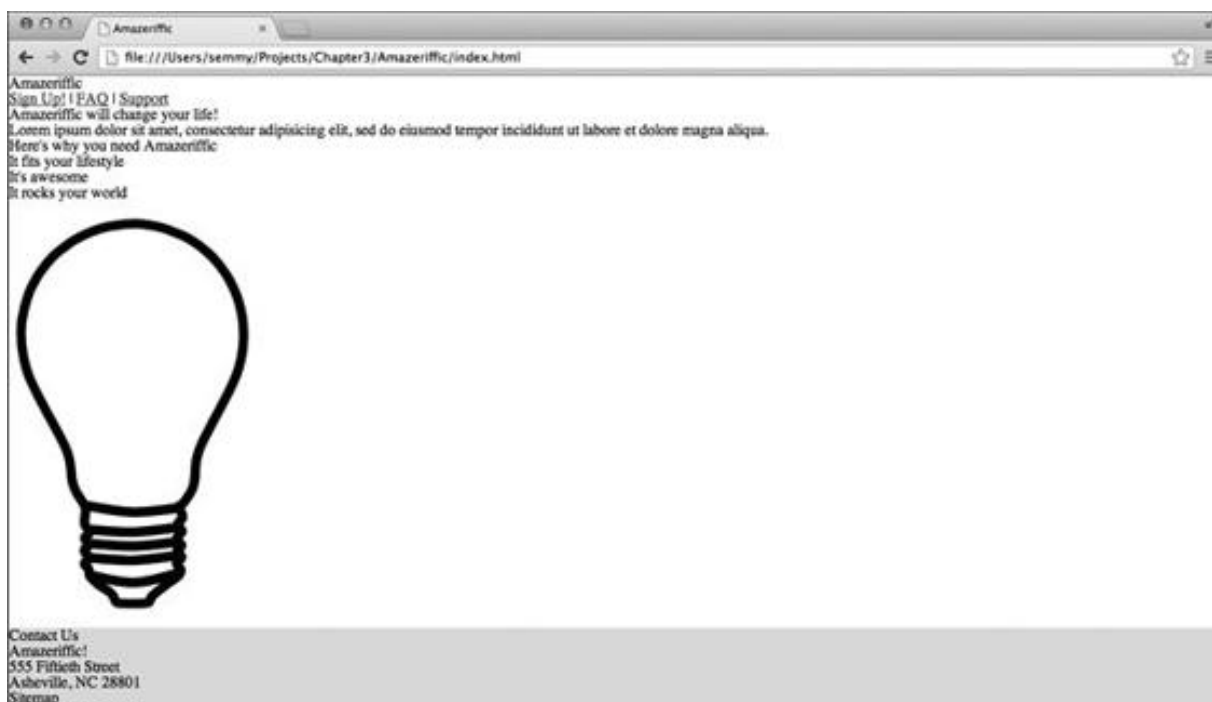


Figura 3.23 – O Amazeriffic após um pouco de estilização (bem básica).

Agora que já temos definida uma estilização básica, provavelmente é uma boa ideia efetuar um commit das alterações em seu repositório Git. Comece verificando o status. O único arquivo que deve ter sido alterado é `style.css`. Coloque esse arquivo no stage (por meio do comando `add`) e, em seguida, efetue o commit da alteração com uma mensagem significativa de commit.

A seguir, queremos que aconteça o seguinte. O conteúdo deve estar centralizado e fixo com 855 px de largura. Estamos optando por 855 px porque queremos que o número de pixels seja divisível por 3 (pois as colunas têm largura de cerca de dois terços e um terço). Como $855/3$ é igual a 285, isso nos permite trabalhar com números inteiros.

Aqui está uma possível abordagem para solucionar esse problema. Podemos adicionar uma regra `max-width` ao conjunto de regras que estiliza `header`, `main` e `footer`, definir o `width` de `body` para 855px e, em

seguida, definir `margin` em `body` para `auto`. Antes de prosseguir, experimente fazer isso para ver o que acontece.

Se você realmente experimentou fazê-lo, é provável que tenha percebido que as áreas de cabeçalho e de conteúdo principal agora não estão mais sendo expandidas de modo a preencher o tamanho total da página e, desse modo, a cor de fundo está preenchendo os espaços à esquerda e à direita. Não é isso que queremos.

Como corrigir esse problema? Esse é um caso em que devemos adicionar um elemento `div` ao DOM para que possamos estilizar os elementos de maneira independente. Queremos que os elementos `header` e `main` ocupem toda a largura da página, mas o conteúdo permaneça com 855 px. Portanto criaremos uma classe contêiner nesses dois elementos para armazenar o conteúdo. Vá em frente e modifique o HTML para que os elementos `header`, `main` e `footer` contenham uma classe chamada `container`, que contém os elementos referentes ao conteúdo. Por exemplo, aqui está o que faremos com a `div` de `header`:

```
<header>
  <div class="container">
    <h1>Amazeriffic</h1>
    <nav>
      <a href="#">Sign Up!</a> |
      <a href="#">FAQ</a> |
      <a href="#">Support</a>
    </nav>
  </div>
</header>
```

Agora podemos fazer com que o conteúdo do contêiner tenha uma largura máxima de 855 px e uma margem automática. Quando concluirmos, nossos conjuntos de regras deverão ter um aspecto semelhante a:

```
body {
  background: #f9e933;
}

header, main, footer {
```

```
    min-width: 855px;
}
header .container, main .container, footer .container {
    max-width: 855px;
    margin: auto;
}
header {
    background: white;
}
main {
    background: white;
}
```

Agora, quando redimensionarmos o nosso navegador, o conteúdo permanecerá com uma largura fixa! Este é um bom momento para ir até a janela do terminal e efetuar o commit de suas alterações no repositório Git. Dessa vez, dois arquivos foram modificados, portanto será necessário adicionar ambos ao repositório Git antes de efetuar o commit.

Criando as colunas

A próxima tarefa a ser feita é criar as colunas para os conteúdos. Para isso, faremos o conteúdo que está na coluna da direita flutuar para a direita e especificaremos as larguras da coluna da direita para 285 px e a largura da coluna da esquerda para 570 px. Também definiremos a propriedade `overflow` com `auto` para que o elemento não seja reduzido a zero quando fizermos a flutuação de ambos os elementos filhos.

Ao finalizarmos os conjuntos de regras para `header`, eles terão o seguinte aspecto:

```
header {
    background: white;
    overflow: auto;
}
header h1 {
    float: left;
```

```
width: 570px;  
}  
header nav {  
  float: right;  
  width: 285px;  
}
```

O cabeçalho é um pouco diferente porque ele contém somente dois elementos filhos – o elemento `h1` e o elemento `nav`. Isso não vale para os elementos `main` e `footer`. Se você estiver pensando nisso da forma correta, é bem provável que seja necessário acrescentar elementos `div` adicionais a ambos a fim de empacotar o conteúdo flutuante da esquerda e da direita em um único elemento. E, do lado direito do rodapé, será preciso criar um layout à esquerda e à direita dentro do layout do rodapé a fim de estilizar o mapa do site. Isso é definido de acordo com o design – você deve ser capaz de fazer isso com as habilidades adquiridas neste capítulo.

Sugiro que você trabalhe em uma seção de cada vez: faça com que o cabeçalho pareça correto; em seguida, faça com que a área de conteúdo principal pareça correta e, por fim, faça o rodapé parecer correto. Realize commits intermediários em seu repositório Git. Sugiro também que você passe o seu código periodicamente pelo CSS Lint (e pela ferramenta de validação de HTML, se você o estiver modificando) para garantir que não haja nenhum problema em potencial.

Ao terminarmos esta seção, teremos algo semelhante ao que está sendo apresentado na figura 3.24 (com exceção do mapa do site, que ainda não está estilizado nessa figura).

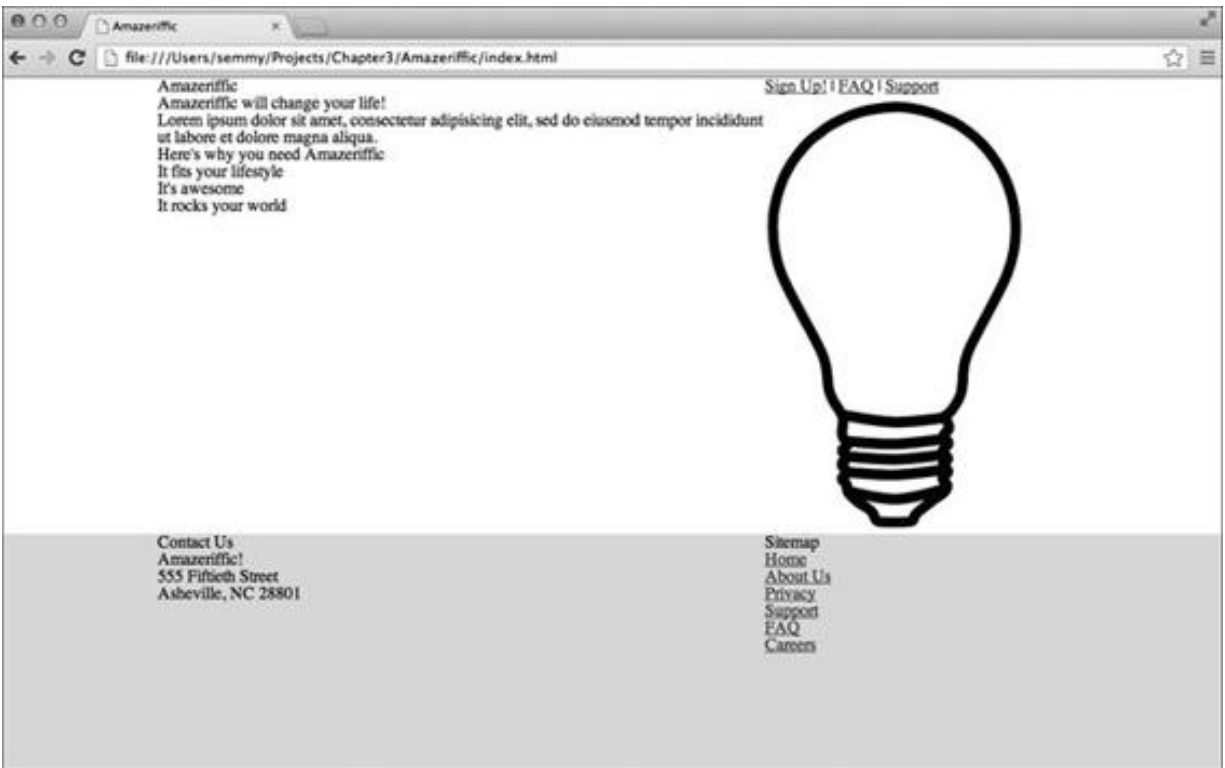


Figura 3.24 – O Amazeriffic após acrescentarmos um pouco mais de estilização.

Depois que a sua página tiver essa aparência e você tiver identificado qualquer possível problema passando o seu CSS pelo CSS Lint, é uma boa hora para fazer outro commit em seu repositório Git.

Adicionando e manipulando fontes

A seguir, adicionaremos duas fontes do Google. Nesse exemplo, utilizei as fontes *Ubuntu* e *Droid Sans*. As tags de cabeçalho (<h1>, <h2>, <h3> e <h4>) estão todas estilizadas com a fonte Ubuntu, enquanto o restante do conteúdo da página está estilizado com Droid Sans. Isso exigirá a adição de duas tags de link em nosso arquivo HTML e também de regras no CSS.

Após ter definido corretamente as fontes, efetue o commit dos arquivos alterados em seu repositório Git. Feito isso, defina um tamanho-base de fonte igual a 100% e modifique os tamanhos das fontes dos elementos em uma tentativa de torná-los o mais parecido

possível com a imagem do mockup (esboço) original.

Depois de fazer com que os tamanhos das fontes pareçam adequados, valide o seu HTML e passe o seu CSS pelo Lint para tentar identificar os problemas. Depois disso, efetue o commit das alterações em seu repositório Git.

Algumas modificações adicionais

Por fim, será necessário adicionar padding e margens para que tudo fique alinhado e espaçado corretamente. Isso exigirá efetuar experimentos com as margens e os paddings em vários elementos. Sugiro que você faça isso no Developer Tools do Chrome. Também será necessário estilizar os links da página. Todos os links são vermelhos e não estão sublinhados, a menos que o cursor seja passado sobre eles. Isso exigirá a manipulação da pseudoclasse `hover` dos elementos `a`.

Resumo

Neste capítulo, utilizamos o CSS para acrescentar estilo à estrutura de nossa interface de usuário e ao conteúdo. A ideia básica por trás do CSS é associar *conjuntos de regras* para estilização aos elementos do DOM. A maioria desses conjuntos de regras é *herdada*, o que significa que, quando as regras são aplicadas a um determinado elemento do DOM, elas também são aplicadas a todos os elementos descendentes do DOM, a menos que sejam sobrescritas por um conjunto específico de regras associado a esse elemento. Nos casos em que houver ambiguidade, regras *em cascata* serão aplicadas.

Podemos restringir os nossos conjuntos de regras CSS a um subconjunto dos elementos do DOM usando *seletores CSS*. Descobrir qual é o seletor apropriado para um elemento ou um conjunto de elementos do DOM requer experiência.

Podemos definir layouts básicos de documentos na forma de uma

grade usando as propriedades de flutuação do CSS. Os elementos CSS podem flutuar para a esquerda ou para a direita, de modo a desviá-los do fluxo normal do documento. Se um elemento tiver dois elementos filhos e nós flutuarmos um elemento filho para a esquerda e o outro para a direita, eles aparecerão lado a lado, supondo que eles não se sobreponham. Podemos garantir que eles não irão se sobrepor se especificarmos suas larguras.

O CSS Lint é uma ferramenta que nos ajuda a evitar armadilhas e erros comuns de CSS, enquanto o Chrome possui um conjunto variado de ferramentas CSS interativas.

Práticas e leituras adicionais

Ainda existem alguns problemas a serem corrigidos no Amazeriffic. Sugiro que você os corrija e faça com que a página se torne a mais parecida possível com o mock-up.

Memorização

Começaremos adicionando alguns passos a mais em nosso exercício. Lembre-se de que o seu objetivo é *memorizar* a maneira de realizar as tarefas, juntamente com os passos já listados nos capítulos anteriores:

1. Crie um arquivo *style.css* simples que utilize o seletor universal para zerar a margem e o padding de todos os elementos e alterar a cor de fundo da página para uma cor próxima ao branco.
2. Faça o link do arquivo *style.css* em seu documento HTML na tag `<head>` e recarregue a página para confirmar se o funcionamento está correto.
3. Faça o commit de *style.css* e de seu *index.html* em seu repositório Git.
4. Acrescente um pouco de estilização básica aos elementos header, main e footer.

5. Faça o commit das alterações em seu repositório.

Exercícios com os seletores CSS

Digite o documento HTML a seguir e salve-o em um arquivo chamado *selectorpractice.html*:

```
<!doctype html>
<html>
  <head>
  </head>

  <body>
    <h1>Hi</h1>
    <h2 class="important">Hi again</h2>
    <p class="a">Random unattached paragraph</p>

    <div class="relevant">
      <p class="a">first</p>
      <p class="a">second</p>
      <p>third</p>
      <p>fourth</p>
      <p class="a">fifth</p>
      <p class="a">sixth</p>
      <p>seventh</p>
    </div>
  </body>
</html>
```

Em seguida, associe um arquivo CSS por meio de um link. No CSS, adicione um único seletor:

```
* {
  color: red;
}
```

Essa regra altera a cor do texto de todos os elementos para vermelho. Utilizaremos esse arquivo para exercitar o uso de seletores CSS. Substitua o * por um seletor que selecione o(s) seguinte(s) elemento(s) e confirme se somente o(s) elemento(s) selecionado(s) é/são alterado(s) para vermelho.

1. Selecione o elemento <h1>.

2. Selecione o elemento `<h2>` pela classe.
3. Selecione todos os parágrafos relevantes (relevant).
4. Selecione o primeiro parágrafo entre os parágrafos relevantes.
5. Selecione o terceiro parágrafo entre os parágrafos relevantes.
6. Selecione o sétimo parágrafo relevante.
7. Selecione todos os parágrafos da página.
8. Selecione somente o parágrafo aleatório, sem associação (random unattached paragraph).
9. Selecione todos os parágrafos com a classe `a`.
10. Selecione somente os parágrafos relevantes com a classe `a`.
11. Selecione o segundo e o quarto parágrafos (DICA: use uma vírgula).

A documentação para desenvolvedores do Mozilla contém um ótimo tutorial sobre seletores (https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Getting_started/Selectors). Esse tutorial inclui muito mais informações do que vimos aqui. Depois que você se sentir à vontade com os seletores apresentados, sugiro que você leia esse tutorial.

Estilize a página de FAQ do Amazeriffic

Se você concluiu os exercícios no final do capítulo 2, uma página de FAQ deve ter sido criada para o Amazeriffic. Faça o link do arquivo `style.css` com `faq.html` e acrescente regras adicionais para estilizar a parte referente ao conteúdo da página. Se a estrutura do cabeçalho e do rodapé forem iguais, você não deverá modificar essas regras. Basta acrescentar novas regras à seção `main` para estilizar a página de FAQ.

Regras em cascata

Neste capítulo, discutimos as regras em cascata do CSS de modo bem genérico. Acho-as bem intuitivas em sua maior parte. Por outro

lado, conhecer as regras em detalhes normalmente ajuda na resolução de problemas de CSS. O W3C apresenta as regras claramente especificadas (<http://www.w3.org/TR/CSS2/cascade.html#cascade>). Sugiro que você as leia e guarde-as na memória, particularmente se você se vir fazendo vários trabalhos com CSS.

Capacidade de ser responsivo e bibliotecas responsivas

Um assunto importante relacionado ao CSS, sobre o qual não aprendemos, refere-se ao *design responsivo*. Dizemos que um design é *responsivo* quando ele altera o seu layout de acordo com a altura e a largura do navegador no qual ele está sendo apresentado. Isso é importante nos dias de hoje por causa da quantidade de pessoas que visualizam as aplicações e as páginas web em telefones celulares e em tablets.

É possível incluir a capacidade de ser responsivo em seu CSS ao usar *media queries*. Porém esse assunto está um pouco além do escopo deste livro. Você pode ler sobre isso no site para desenvolvedores do Mozilla (https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Media_queries). Sugiro que você o faça.

Além do mais, vários frameworks para CSS que permitem criar designs responsivos com um mínimo de esforço estão disponíveis. Tenho usado tanto o Twitter Bootstrap (<http://getbootstrap.com/>) quanto o Foundation da Zurb (<http://foundation.zurb.com/>). Se você gostou de trabalhar com layouts e com CSS neste capítulo, leia a documentação de ambos os frameworks e tente recriar o layout do Amazeriffic de modo a ter um design responsivo usando um deles. Esses dois frameworks também são ótimos para ter em mente como pontos de partida para projetos futuros.

CAPÍTULO 4

Interatividade

Até agora, estudamos três aspectos muito importantes do desenvolvimento de aplicações web. Infelizmente, porém, não criamos nenhuma aplicação que realmente *faça* algo. Pudemos somente estruturar o conteúdo e estilizá-lo para que tivesse uma aparência interessante.

Neste capítulo, começaremos a explorar a interatividade. Isso nos permitirá ir além do desenvolvimento de páginas web que não se alteram. Começaremos a desenvolver aplicativos dinâmicos que respondem às entradas dos usuários.

Para isso, aprenderemos um pouco sobre o JavaScript – a linguagem de scripting executada em todo navegador web – e a jQuery – a biblioteca JavaScript que é útil para a manipulação do DOM (entre outras coisas). Além do mais, investiremos algum tempo aprendendo o básico sobre o JavaScript como linguagem, com ênfase no uso e na manipulação de arrays.

Hello, JavaScript!

Vamos começar com um exemplo. Como antes, criaremos um diretório chamado *Chapter4* em *Projects*. Nesse diretório, começaremos pela criação de um subdiretório chamado *Example1* e o abriremos no Sublime. A seguir, criaremos três arquivos: *index.html*, *style.css* e *app.js*. Vamos começar pelo *index.html*:

```
<!doctype html>
<html>
  <head>
    <title>App Name</title>
```

```
<link href="style.css" rel="stylesheet" type="text/css">
</head>
<body>
  <h1>App Name</h1>
  <p>App Description</p>
  <script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

Tudo em *index.html* deve ser familiar a você, com exceção das tags `<script>` na parte final do elemento `body`. Assim como o elemento `link`, os elementos `script` permitem associar arquivos externos ao seu HTML, porém esses elementos normalmente apontam para arquivos JavaScript. Outro aspecto importante a ser observado é que, de modo diferente da tag `<link>`, o `<script>` sempre exige uma tag de fechamento.

Optamos por posicionar as tags `<script>` no elemento `body` em vez de colocar em `head` por um motivo de caráter técnico: o navegador apresenta a página de cima para baixo, criando os elementos do DOM à medida que os encontra no documento HTML. Ao posicionar as tags `<script>` no final, os arquivos JavaScript serão um dos últimos itens da página a serem carregados. Como os arquivos JavaScript normalmente são bem grandes e exigem um pouco de tempo para a carga, preferimos fazer isso por último para que o usuário obtenha feedback visual dos outros elementos o mais rápido possível.

A primeira tag `<script>` inclui uma biblioteca chamada *jQuery*, comumente utilizada do lado do cliente, sobre a qual aprenderemos bastante nas próximas seções. A *jQuery* é carregada a partir de *http://code.jquery.com*, que é uma *CDN* (Content Delivery Network, ou Rede de fornecimento de conteúdo). Isso significa que não estamos mantendo uma cópia da *jQuery* em nosso computador local. Em vez disso, informamos ao navegador para que efetue o download dessa biblioteca a partir de um site externo, semelhante à maneira pela qual trabalhamos com o Google Fonts no capítulo

anterior. Portanto nosso computador deverá estar conectado à Internet para que esse exemplo funcione adequadamente.

O segundo elemento `script` inclui o script *app.js*, que contém o código JavaScript a ser implementado. Criaremos esse arquivo a seguir.

Também temos um link para um arquivo CSS, que configuramos para remover a margem e o padding default de todos os elementos da página. O uso do seletor universal fará com que o CSS Lint gere um warning (aviso), porém vamos ignorá-lo por enquanto:

```
* {  
  margin: 0;  
  padding: 0;  
}
```

O arquivo no qual estamos mais interessados é o *app.js*, e seu conteúdo está sendo mostrado no código a seguir. Antes de discutirmos a seu respeito, vá em frente e digite-o exatamente como você o vê aqui; em seguida, abra *index.html* em seu navegador web:

```
var main = function () {  
  "use strict";  
  window.alert("hello world!");  
};  
$(document).ready(main);
```

Se tudo for digitado corretamente, você deverá ver uma caixa de diálogo de alerta contendo *hello world!* quando a página for carregada, conforme mostrado na figura 4.1.

Esse é um esqueleto básico que usaremos em todos os nossos programas JavaScript. Ele executa duas tarefas principais que podem ser generalizadas para os demais códigos das aplicações presentes neste livro, referentes ao lado do cliente: (1) define uma função global chamada *main*, que corresponde ao ponto de entrada para a execução de nosso programa e (2) usa a jQuery para configurar a execução da função *main* após o documento HTML estar totalmente carregado e pronto.

O código também está definido para executar em modo *strict*

(estrito), o que desabilita certos aspectos ruins do JavaScript que causaram problemas aos programadores no passado. Essa sempre será a primeira linha de nossa função `main`. O script também cria o diálogo “hello world!”, mas isso é somente para esse exemplo.

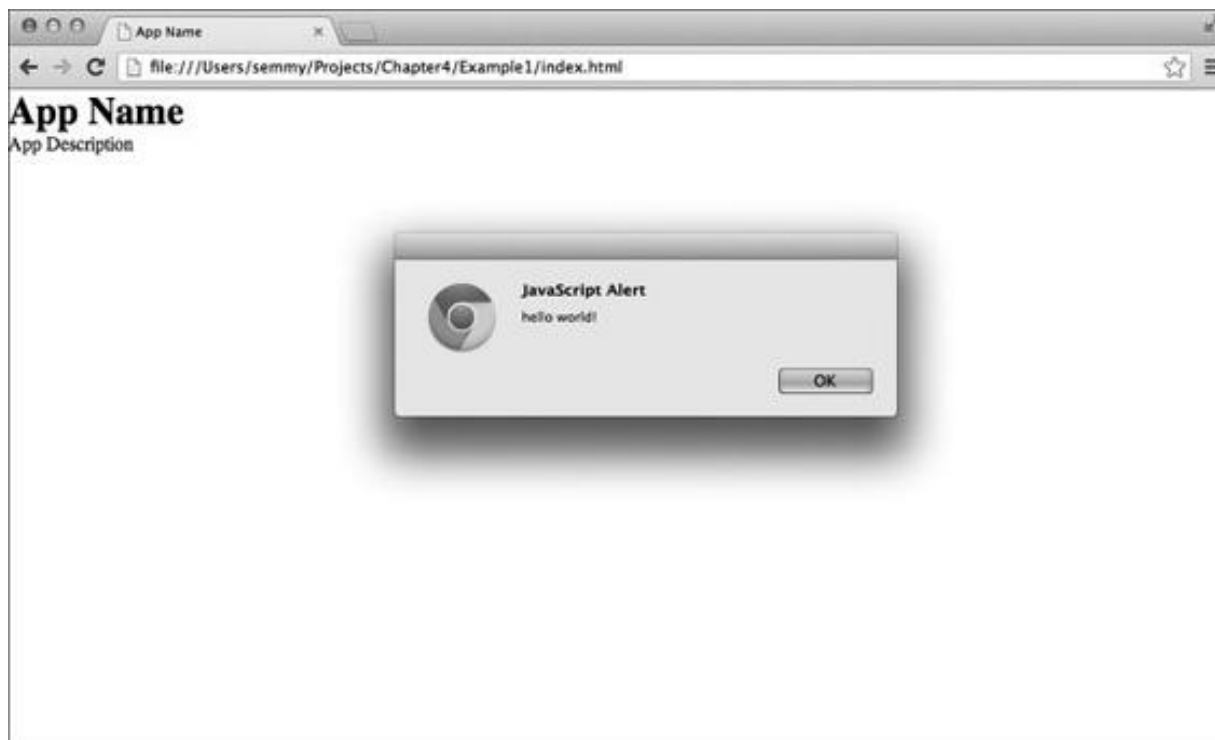


Figura 4.1 – Nossa primeira aplicação interativa (ou quase).

Nossa primeira aplicação interativa

Agora usaremos a jQuery para criar algo um pouco mais interessante. Nossa meta é criar uma aplicação simples que permita ao usuário inserir um pouco de texto que, posteriormente, será adicionado à página em um local diferente. Esse exemplo demonstrará que é muito fácil fazer coisas interessantes acontecerem usando somente algumas linhas de código.

Vamos começar pela criação de um novo diretório chamado *Example2* em nosso diretório *Chapter4*. Inicialize um repositório Git vazio e abra o diretório no Sublime. Recrie os três arquivos de exemplo da seção anterior (*index.html*, *style.css* e *app.js*). Conforme discutimos nos capítulos anteriores, é uma boa ideia memorizar

essa estrutura básica para que você possa criá-la a partir do zero, sem a necessidade de copiar nenhum código.

Depois que o esqueleto do projeto estiver definido, faça o commit inicial em seu repositório Git.

A estrutura

Vamos dar uma olhada rápida na interface do usuário antes de começar. Ela está sendo mostrada na figura 4.2.

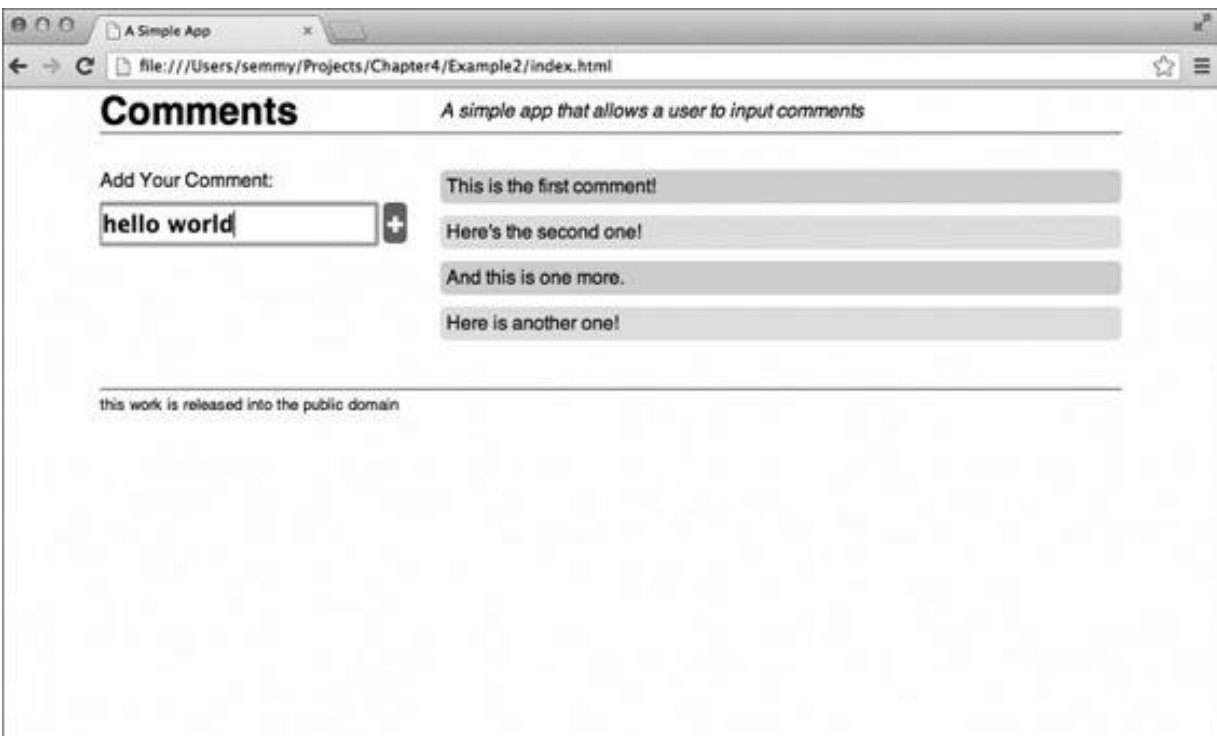


Figura 4.2 – A interface do usuário de nossa primeira aplicação interativa.

Você verá que a estrutura é constituída de três elementos principais: o header, o footer e o main. Com isso em mente, podemos modificar o nosso HTML para que represente a nossa interface de usuário:

```
<!doctype html>
<html>
  <head>
    <title>A Simple App</title>
    <link href="style.css" rel="stylesheet" type="text/css">
```



```

</head>
<body>
  <header>
  </header>

  <main>
    <!-- observe que usamos um hífen aqui no nome de nossa classe CSS
         que contém várias palavras: essa é uma convenção do CSS -->
    <section class="comment-input">
    </section>

    <section class="comments">
    </section>
  </main>

  <footer>
  </footer>

  <script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
  <script src="app.js"></script>
</body>
</html>

```

Após ter modificado o seu HTML para que se pareça com o código anterior, faça o commit desse arquivo em seu repositório Git. A seguir, preencha os elementos `header` e `footer`, juntamente com a parte principal do conteúdo de `main` que está à esquerda para que corresponda ao design mostrado na figura 4.2. Espero que você possa perceber que algumas das seções terão uma tag `<h1>`, `<h2>` ou `<h3>`, juntamente com algum conteúdo. Adicione-os agora.

Na seção `comment-input`, usaremos uma nova tag HTML: `<input>`. O elemento `input` representa uma caixa de texto de entrada e não exige uma tag de fechamento. Acrescente uma tag `<input>` vazia à seção `comment-input`. Você também pode adicionar o parágrafo com o nosso prompt “Add Your Comment:” (Adicione o seu comentário:)

```

<section class="comment-input">
  <p>Add Your Comment:</p>
  <input type="text">
</section>

```

Na seção `.comments`, adicione alguns exemplos de comentário que corresponderão simplesmente a tags de parágrafo. Isso nos ajudará

na estilização. Por exemplo, podemos modificar nossa seção de comentários para que tenha este aspecto:

```
<section class="comments">
  <p>This is the first comment!</p>
  <p>Here's the second one!</p>
  <p>And this is one more.</p>
  <p>Here is another one</p>
</section>
```

Por fim, adicionaremos um elemento `button` cujo label (rótulo) será o caractere `+`. Embora ainda não tenhamos visto o elemento `button`, não haverá nada de particularmente especial nele até que lhe associemos um comportamento JavaScript. Para adicionar um botão, acrescentamos a tag `<button>` imediatamente após a tag `<input>`. O código de minha seção `comment-input` se parece com:

```
<section class="comment-input">
  <p>Add Your Comment:</p>
  <input type="text"><button>+</button>
</section>
```

Após termos a estrutura definida, vamos efetuar o seu commit em nosso repositório Git e prosseguir para a estilização da página.

O estilo

O estilo é constituído de alguns elementos simples. Em primeiro lugar, pode ser interessante utilizar um reset, porém não se sinta na obrigação de fazer isso neste exemplo. No mínimo, você deve manter o ponto de partida representado pelo CSS simples, em que zeramos a margem e o padding defaults.

A largura do corpo tem 855 pixels nesse exemplo, com a coluna da esquerda ocupando um terço desse espaço. Os outros dois terços do corpo são constituídos pelos comentários.

Um aspecto interessante da seção de comentários é que alternamos as suas cores. Em outras palavras, os comentários de número par são de uma cor e os comentários de número ímpar são de outra cor. Podemos obter esse efeito utilizando somente o CSS – basta usar a

pseudoclasse `nth-child`, juntamente com o valor `even` ou `odd`. Por exemplo, se quisermos alternar as cores `lavender` e `gainsboro`, podemos criar conjuntos de regras CSS desta maneira:

```
.comments p:nth-child(even) {  
  background: lavender;  
}  
.comments p:nth-child(odd) {  
  background: gainsboro;  
}
```

Se alguns comentários de exemplo forem adicionados, você deverá ver as cores de fundo se alternarem após esse conjunto de regras ter sido adicionado ao CSS. Observe que também definimos um pequeno `border-radius` em nossos comentários para que tivessem bordas arredondadas.

Com exceção desses itens, o restante desse layout deve ser muito semelhante ao dos exemplos anteriores. Termine de estilizar a interface da melhor maneira possível e faça o commit de seu código no repositório Git para que possamos prosseguir e torná-lo interativo. Se tiver problemas, você poderá consultar o código em nosso repositório do GitHub (<http://bit.ly/1fY92Fe>).

Interatividade

Agora estamos prontos para trabalhar com a interatividade. Implementaremos esse exemplo passo a passo, efetuando commits no Git durante o processo.

Lidando com eventos clique

Vamos começar fazendo o seguinte: quando o usuário clicar no botão `+`, iremos inserir um novo comentário na seção de comentários. Para isso, começaremos pela modificação do código de nosso *app.js* para que tenha o seguinte aspecto:

```
var main = function () {  
  "use strict";  
  $(".comment-input button").on("click", function (event) {
```

```
        console.log("Hello World!");
    });
};

$(document).ready(main);
```

Abra a sua página no navegador e abra o Developer Tools do Chrome, conforme descrito no capítulo anterior. Estando nessa ferramenta, clique na aba Console na parte superior do Developer Tools. Agora, ao clicar no botão + da página, você deverá ver “Hello World!” aparecer no console! Se tudo estiver definido e funcionando corretamente, é uma boa hora para efetuar o commit de seu progresso em seu repositório Git.

O que está acontecendo aqui? Esse código associa um *listener de evento* ao elemento do DOM referenciado na chamada à função \$. Observe que o conteúdo de \$ se parece bastante com um seletor CSS – e isso não é uma coincidência! É exatamente como deve ser! A jQuery permite facilmente selecionar elementos do DOM como alvos usando seletores CSS, e esses elementos são manipulados por meio do JavaScript.

O evento que estamos ouvindo no elemento `button` é o evento “click” (clique). O listener propriamente dito é uma função que simplesmente exibe “Hello World!” no console. Desse modo, podemos traduzir o código para o português da seguinte maneira: *quando o usuário clicar em +, exiba “Hello World!” no console.*

É claro que não queremos realmente exibir “Hello World!” no console – queremos adicionar um comentário na seção de comentários. Para seguir nessa direção, modificaremos um pouco o nosso código substituindo o `console.log` por outra linha da jQuery que adiciona um elemento do DOM à seção de comentários:

```
var main = function () {
    "use strict";

    $(".comment-input button").on("click", function (event) {
        $(".comments").append("<p>this is a new comment</p>");
    });
};
```

```
$(document).ready(main);
```

Atualize a janela de seu navegador e, se tudo correr bem, você deverá ser capaz de clicar em + e ver a frase “this is a new comment” (este é um novo comentário) ser adicionada à seção de comentários. Você perceberá que o código da jQuery (que começa com o \$) seleciona a seção de comentários da mesma maneira que nós a selecionamos em nosso arquivo CSS e, em seguida, adiciona uma porção de código HTML a ela. Se tudo estiver funcionando, faça o commit de seu código usando uma mensagem significativa. Se houver problemas, volte e certifique-se de que tudo tenha sido digitado corretamente.

Manipulando dinamicamente o DOM

Fizemos um pouco de progresso em direção ao nosso objetivo, porém, sempre que clicamos no botão, o mesmo conteúdo é adicionado. Queremos alterar o texto do parágrafo de acordo com o conteúdo que estiver na caixa de entrada. Começaremos pela criação de uma variável que armazenará o elemento do DOM que iremos adicionar:

```
var main = function () {  
  "use strict";  
  
  $(".comment-input button").on("click", function (event) {  
    var $new_comment = $("<p>");  
  
    $new_comment.text("this is a new comment");  
    $(".comments").append($new_comment);  
  });  
};  
  
$(document).ready(main);
```

A modificação, na realidade, não alterou nada em relação ao que o nosso código está fazendo: ela simplesmente efetuou uma *refatoração* para que pudéssemos personalizar mais facilmente o conteúdo textual da tag de parágrafo. Especificamente, adicionamos uma declaração de *variável* e uma atribuição. A variável `$new_comment` pode ter o nome que quisermos, porém, se ela for armazenar um

objeto jQuery, às vezes é interessante distingui-la usando \$ como o primeiro caractere.

A primeira linha do código novo cria um novo elemento do tipo parágrafo na forma de um objeto jQuery, e a segunda linha altera o conteúdo textual do novo parágrafo para “this is a new comment” (este é um comentário novo). Como a jQuery permite *encadear* chamadas de funções, podemos executar ambas as linhas em uma só, se preferirmos:

```
var $new_comment = $("

").text("this is a new comment");


```

Mesmo que o recurso de encadeamento da jQuery seja utilizado, é importante lembrar-se de que dois fatos estão ocorrendo aqui – um novo elemento do tipo parágrafo está sendo criado e o conteúdo textual desse elemento está sendo alterado para “this is a new comment”.

A seguir, gostaríamos de obter o conteúdo da caixa de entrada para armazená-lo na variável que criamos. Antes de prosseguir, porém, pare um instante e procure descobrir como fazer a seleção do elemento `input` contido na seção `.comment-input` usando a jQuery.

Espero que você tenha tentado usar seletores CSS! A resposta é a seguinte:

```
$(".comment-input input");
```

Assim como no CSS, essa linha seleciona os elementos que têm a classe `.comment-input` e depois seleciona os elementos `input` que são descendentes das seções `.comment-input`. Como somente um elemento atende a esse critérios, ele será o único elemento a ser selecionado.

Agora que sabemos disso, podemos obter o conteúdo da caixa de entrada. Acontece que a jQuery tem uma função que retorna o conteúdo de uma caixa de entrada, e essa função chama-se `val`, que é a abreviatura de `value` (valor). Podemos acessar o conteúdo da caixa de entrada usando o código a seguir:

```
$(".comment-input input").val();
```

Agora só precisamos fazer algo com esse valor! Nesse caso,

faremos com que o conteúdo da caixa de entrada seja o valor do texto de nosso novo elemento parágrafo. Portanto podemos refatorar o nosso código para que tenha o seguinte aspecto:

```
var main = function () {  
  "use strict";  
  $(".comment-input button").on("click", function (event) {  
    var $new_comment = $("<p>"),  
    comment_text = $(".comment-input input").val();  
    $new_comment.text(comment_text);  
    $(".comments").append($new_comment);  
  });  
};  
$(document).ready(main);
```

E, se quisermos, podemos executar boa parte desse código em uma única linha, sem utilizar uma variável intermediária para armazenar o resultado da chamada à função `val`:

```
var $new_comment = $("<p>").text($(".comment-input input").val());
```

Agora abra a aplicação em seu navegador. Se tudo estiver funcionando corretamente, você deverá poder digitar um conteúdo na caixa de texto e, em seguida, clicar no botão e ver o conteúdo ser adicionado à seção de comentários. Se algo não estiver funcionando, verifique novamente para garantir que todos os pontos e vírgulas e parênteses estejam nos lugares corretos e então abra o Developer Tools do Chrome para conferir se há algum erro no console. O erro pode oferecer pistas sobre o que foi digitado incorretamente.

Depois que tudo estiver funcionando de modo que você fique satisfeito, faça o commit das alterações em seu repositório Git usando uma mensagem significativa.

A essa altura, de modo geral, tudo deve estar funcionando. Para os nossos usuários, porém, a experiência deixa muito a desejar, e podemos fazer pequenas alterações que proporcionarão melhorias substanciais.

Eliminando um bug

Provavelmente, você ainda não percebeu que o nosso código contém um bug, mas o fato é que ele está lá! É interessante reservar um momento para pensar no comportamento esperado da aplicação e ver se você é capaz de encontrar esse bug.

Caso ainda não o tenha encontrado, aqui está ele: quando clicamos no botão de adição e não há nenhum conteúdo em nossa caixa de entrada, o nosso programa jQuery adiciona um elemento `p` vazio ao DOM. De que modo esse bug se apresenta? Na realidade, ele aparece como um problema no esquema de cores pares e ímpares que aplicamos em nosso CSS.

Para ver esse bug dar sinal de vida, comece recarregando a página. A seguir, digite um comentário usando a caixa de entrada, limpe a caixa, clique no botão de adição e, em seguida, adicione outro comentário. Se essas instruções forem seguidas, você verá que os dois comentários que aparecem têm a mesma cor! Isso ocorre porque o comentário vazio está assumindo a cor ímpar e ele não é mostrado.

Também podemos conferir se o bug está sendo causado pelo elemento `p` vazio usando a aba Elements (Elementos) da janela Developer Tools do Chrome. Comece abrindo o Developer Tools e clique na aba Elements. A seguir, detalhe o elemento `main` e entre na seção de comentários. Após essa seção ter sido aberta, clique no botão sem que haja conteúdo na caixa de entrada. Você verá o elemento `p` vazio sendo adicionado.

Como podemos corrigir esse problema? Basicamente, iremos inserir uma verificação para conferir se o conteúdo da caixa de entrada está vazio antes de fazer algo com ele. Podemos fazer isso por meio de uma instrução `if`:

```
$(".comment-input button").on("click", function (event) {  
    var $new_comment;  
    if ($(".comment-input input").val() !== "") {  
        $new_comment = $("

").text($(".comment-input input").val());  
        $(".comments").append($new_comment);  
    }  
})


```



```
});
```

O `!==` confirma se o conteúdo da caixa de entrada *não* é igual a uma string vazia, que é basicamente equivalente a verificar se a caixa de entrada está vazia. Portanto a instrução `if` somente executará o código se a caixa de entrada não estiver vazia. Essa alteração simples deve corrigir o bug e, feito isso, é uma boa ideia efetuar um commit em seu repositório Git.



Observe que mudamos a declaração da variável para que ficasse antes da instrução `if`. É sempre uma boa ideia manter suas variáveis declaradas no início de suas definições de função.

Limpendo a caixa de entrada

O próximo problema importante relacionado à experiência do usuário é o fato de a caixa de entrada não estar sendo limpa quando os usuários clicam no botão. Se os usuários quiserem inserir um segundo comentário, eles deverão apagar manualmente o conteúdo que estiver lá anteriormente. Limpar essa caixa de entrada é bem fácil: se chamarmos o método `val` do objeto jQuery com um valor explícito, a caixa será preenchida com esse valor. Em outras palavras, podemos limpar o conteúdo atual da caixa enviando uma string vazia ao método `val`:

```
$(".comment-input input").val("");
```

Desse modo, uma linha de código a mais fará esse recurso ser acrescentado:

```
$(".comment-input button").on("click", function (event) {  
    var $new_comment;  
  
    if ($(".comment-input input").val() !== "") {  
        $new_comment = $("<p>").text($(".comment-input input").val());  
        $(".comments").append($new_comment);  
        $(".comment-input input").val("");  
    }  
});
```

Fazendo a tecla Enter funcionar conforme esperado

Outro recurso esperado pelos usuários é que a tecla Enter deve

efetuar a submissão do comentário. Isso normalmente ocorre quando estamos interagindo com um programa de bate-papo, por exemplo.

Como podemos fazer isso acontecer? Podemos acrescentar um event handler adicional que preste atenção no evento keypress (pressionamento de tecla) no próprio elemento `input`. Isso pode ser acrescentado diretamente após o nosso event handler para cliques:

```
$(".comment-input input").on("keypress", function (event) {  
    console.log("hello world!");  
});
```

Observe que há duas diferenças principais entre esse listener e o anterior. A primeira diferença é que esse listener está configurado para prestar atenção no evento keypress no lugar do evento clique. A segunda é que estamos ouvindo um evento em um elemento diferente: nesse caso, estamos prestando atenção na caixa de entrada, enquanto no exemplo anterior estávamos ouvindo no elemento `button`.

Se você tentar executar esse código, verá que “Hello World!” será exibido no console do desenvolvedor no Chrome sempre que digitarmos uma tecla. Queremos ignorar a maior parte das teclas digitadas na caixa de entrada e reagir somente quando o usuário teclar Enter. Para isso, podemos usar a variável local `event` que ignoramos no handler anterior – ela armazena o valor da tecla digitada. Como podemos ver esse valor? Vamos modificar um pouco o nosso código:

```
$(".comment-input input").on("keypress", function (event) {  
    console.log("this is the keyCode " + event.keyCode);  
});
```

Observe que o `C` em `keyCode` é maiúsculo. Esse é um exemplo de *camelCase*: quando temos um nome de variável composto de várias palavras, utilizamos letras maiúsculas no início de cada uma delas após a primeira.

Na saída, estamos usando `+` para concatenar o valor de `keyCode` à

string que começa com “this is the keyCode” (este é o keyCode). Quando o seu código estiver executando, você verá o valor de keyCode na saída.

Agora, quando atualizarmos o navegador e começarmos a digitar na caixa de entrada, veremos os keyCodes fazendo rolagens na tela. Podemos usar isso para descobrir qual é o keyCode correspondente à tecla Enter. Feito isso, podemos encapsular o nosso código em uma instrução if para que ele responda somente à tecla Enter:

```
$(".comment-input input").on("keypress", function (event) {  
    if (event.keyCode === 13) {  
        console.log("this is the keyCode " + event.keyCode);  
    }  
});
```

Esse código exibe o keyCode somente quando a tecla Enter for pressionada. Por fim, podemos copiar o código que adiciona um novo comentário de nosso outro listener de evento:

```
$(".comment-input input").on("keypress", function (event) {  
    var $new_comment;  
    if (event.keyCode === 13) {  
        if ($(".comment-input input").val() !== "") {  
            var $new_comment = $("<p>").text($(".comment-input input").val());  
            $(".comments").append($new_comment);  
            $(".comment-input input").val("");  
        }  
    }  
});
```

Fazendo o fade in do novo comentário

Agora todos os nossos recursos importantes devem estar funcionando. Entretanto vamos adicionar mais um aspecto à experiência: em vez de o novo comentário simplesmente aparecer imediatamente, vamos fazê-lo aparecer gradualmente (fade in). Felizmente, a jQuery facilita bastante essa tarefa porque todo elemento jQuery possui um método `fadeIn` incluído. Porém, para efetuar o fade in do elemento, é necessário garantir que ele estará

inicialmente oculto. Para isso, chamaremos o método `hide` do elemento antes de adicioná-lo ao DOM. O código a seguir faz exatamente isso:

```
$(".comment-input button").on("click", function (event) {  
    var $new_comment;  
  
    if ($("#comment-input input").val() !== "") {  
        $new_comment = $("

").text($("#comment-input input").val());  
        $new_comment.hide();  
        $(".comments").append($new_comment);  
        $new_comment.fadeIn();  
        $("#comment-input input").val("");  
    }  
});


```

Agora, quando adicionarmos um comentário com o botão, veremos que ele aparecerá gradualmente. Também devemos modificar o nosso evento `keypress` para fazer o mesmo. Depois que isso estiver funcionando, faça o commit de seu código.

Refatorando para simplificar

A essa altura, o meu arquivo *app.js* tem o aspecto apresentado a seguir, e o seu deverá ser semelhante:

```
var main = function () {  
    "use strict";  
  
    $(".comment-input button").on("click", function (event) {  
        var $new_comment;  
  
        if ($("#comment-input input").val() !== "") {  
            $new_comment = $("

").text($("#comment-input input").val());  
            $new_comment.hide();  
            $(".comments").append($new_comment);  
            $new_comment.fadeIn();  
            $("#comment-input input").val("");  
        }  
    });  
  
    $(".comment-input input").on("keypress", function (event) {  
        var $new_comment;  
  
        if (event.keyCode === 13) {  
            if ($("#comment-input input").val() !== "") {


```

```

        $new_comment = $("<p>").text($(".comment-input input").val());
        $new_comment.hide();
        $(".comments").append($new_comment);
        $new_comment.fadeIn();
        $(".comment-input input").val("");
    }
}
});
};
$(document).ready(main);

```

Um aspecto que você perceberá imediatamente é que temos código duplicado. Em particular, o nosso código que adiciona um comentário está duplicado em ambos os event handlers e, quando alteramos um deles para fazer o fade in do texto, tivemos de alterar o outro também. Isso viola um princípio de desenvolvimento de software conhecido como princípio *DRY* (Don't Repeat Yourself, ou Não seja repetitivo). Sempre que recortarmos e colarmos um código, uma pequeno alarme deverá disparar em nossa mente para nos avisar que, provavelmente, deve haver uma maneira melhor de fazer o que pretendemos implementar. Nesse caso, podemos reorganizar o nosso código para que tenha o seguinte aspecto:

```

var main = function () {
    "use strict";
    var addCommentFromInputBox = function () {
        var $new_comment;

        if($(".comment-input input").val() !== "") {
            $new_comment = $("<p>").text($(".comment-input input").val());
            $new_comment.hide();
            $(".comments").append($new_comment);
            $new_comment.fadeIn();
            $(".comment-input input").val("");
        }
    };
    $(".comment-input button").on("click", function (event) {
        addCommentFromInputBox();
    });
    $(".comment-input input").on("keypress", function (event) {
        if (event.keyCode === 13) {

```

```
        addCommentFromInputBox();  
    }  
    });  
};  
$(document).ready(main);
```

Nesse exemplo, abstraímos o código duplicado na forma de uma função reutilizável e então chamamos a função em cada um dos listeners de evento. Fazemos isso por meio da declaração de uma variável para armazenar a função e, em seguida, definimos a função. Isso resulta em um código cuja manutenção é muito mais simples: agora, quando for necessário fazer uma alteração no comportamento referente à adição de comentários, isso precisará ser feito somente em um único local!

Generalização da jQuery

Uau! Esse exemplo foi bem intenso, mas espero que você tenha conseguido sobreviver a ele! Olhando pelo lado positivo, conseguimos ver vários dos tipos de tarefas que a jQuery pode executar. Agora vamos dar um passo para trás e analisar alguns aspectos que podem ser generalizados a partir desse exemplo. Essencialmente, a jQuery nos oferece três recursos:

- uma abordagem simplificada e expressiva para a manipulação do DOM;
- uma abordagem consistente para o tratamento de eventos do DOM;
- uma abordagem simplificada para o AJAX.

Estudaremos os dois primeiros recursos neste capítulo e o terceiro, no próximo. Também vale a pena mencionar que a jQuery tem uma quantidade incrível de plug-ins de terceiros que podem acrescentar melhorias aos sites de forma rápida e fácil.

Criando um projeto

Antes de começar, vamos discutir um pouco mais sobre o nosso fluxo de trabalho. Em geral, a parte de nossa aplicação relativa ao lado do cliente pode dar origem a vários arquivos HTML, CSS e JavaScript. Sendo assim, normalmente é útil manter esses arquivos organizados de modo que faça sentido.

A partir de agora, iremos manter todos os nossos arquivos HTML no diretório raiz de nosso projeto e teremos três subdiretórios chamados *stylesheets*, *images* e *javascripts*. Iremos garantir que os conteúdos desses diretórios sejam, adequadamente, um reflexo de seus nomes.

Isso muda um pouco a situação quando efetuarmos o link com as folhas de estilo ou quando importarmos scripts. Aqui está um exemplo de como essa alteração afeta o nosso HTML:

```
<!doctype html>
<html>
  <head>
    <link href="stylesheets/reset.css" rel="stylesheet">
    <link href="stylesheets/style.css" rel="stylesheet">
  </head>
  <body>
    <script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
    <script src="javascripts/app.js"></script>
  </body>
</html>
```

Observe que tivemos de especificar explicitamente a localização de nosso script e dos arquivos CSS, prefixando-os com o nome do diretório e uma barra.

Comentários

Antes de começar a digitar código de verdade, provavelmente é interessante iniciar uma discussão sobre os comentários em JavaScript. Assim como no HTML e no CSS, podemos inserir comentários em nosso JavaScript a fim de inserir anotações destinadas aos leitores humanos em nosso código. Há dois tipos de

comentário: um comentário de uma linha e um comentário de várias linhas. Comentários de uma única linha utilizam duas barras, uma ao lado da outra, enquanto comentários que ocupam várias linhas são semelhantes aos comentários do CSS:

```
// este é um comentário de uma só linha, que avança até o final da linha  
var a = 5; // este é um comentário de uma só linha após um código  
/* Este é um comentário de várias linhas que se prolongará até  
que nós o fechemos. É semelhante a um comentário no CSS */
```

Seletores

Como vimos no exemplo anterior, os seletores jQuery são muito semelhantes aos seletores CSS. Com efeito, podemos usar qualquer seletor CSS como um seletor jQuery. Por exemplo, os seletores jQuery a seguir selecionam exatamente aquilo que esperamos que seja selecionado:

```
$("*"); // seleciona todos os elementos do documento  
$("h1"); // seleciona todos os elementos h1  
$("p"); // seleciona todos os elementos p  
$("p .first"); // seleciona todos os elementos do tipo parágrafo que  
                // tenham a classe 'first'  
$(".first"); // seleciona todos os elementos que tenham a classe 'first'  
$("p:nth-child(3)"); // seleciona todos os elementos do tipo parágrafo que  
                    // correspondem ao terceiro filho
```

No entanto os seletores jQuery não são idênticos aos seletores CSS em todos os casos. Com efeito, a jQuery acrescenta um conjunto rico de pseudoclasses e de pseudoelementos que, atualmente, não estão disponíveis no CSS. Além do mais, certos identificadores válidos no CSS devem ser representados de modo diferente quando usados na jQuery (caracteres especiais como ponto (.) devem ser escapados com duas barras invertidas). No entanto, em nosso caso, vamos pensar nos seletores jQuery como seletores CSS que retornam elementos do DOM, os quais podem ser manipulados em nosso JavaScript. Se você estiver interessado em aprender mais, dê uma olhada na documentação sobre os seletores jQuery

(<http://api.jquery.com/category/selectors/>).

Manipulação do DOM

Após termos usado os seletores para a adição de elementos ao DOM com sucesso, é provável que queiramos manipular esses elementos de alguma maneira. A jQuery facilita bastante essa tarefa.

Adicionando elementos ao DOM

Para começar, é interessante nos lembrarmos do nosso modelo mental em forma de árvore para o DOM. Por exemplo, considere este trecho de HTML:

```
<body>
  <h1>This is an example!</h1>
  <main>
</main>
  <footer>
</footer>
</body>
```

Observe que os nossos elementos `main` e `footer` estão vazios. Podemos desenhar um diagrama de árvore, conforme mostrado na figura 4.3.

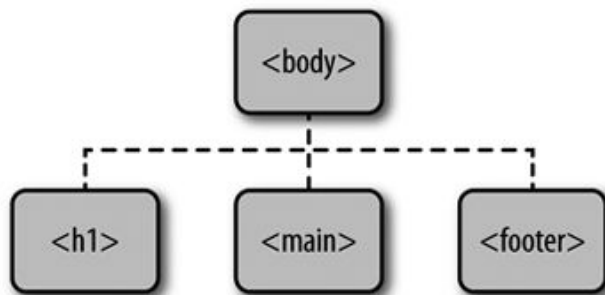


Figura 4.3 – Um diagrama de árvore do DOM antes de qualquer manipulação ter sido feita com a jQuery.

Para criar um elemento, usamos o mesmo operador `$` utilizado para selecionar um elemento. Em vez de enviar um seletor, porém, enviamos a tag que representa o elemento que queremos criar:

```
// Por convenção, início o nome de variáveis que representam objetos jQuery
// com um $
var $newUL = $("<ul>"); // cria um novo elemento ul
var $newParagraphElement = $("<p>"); // cria um novo elemento p
```

Após termos criado um elemento, podemos adicionar itens a ele, inclusive outros elemento HTML. Por exemplo, podemos adicionar texto ao nosso elemento p desta maneira:

```
$newParagraphElement.text("this is a paragraph");
```

Depois que esse código for executado, o nosso elemento do DOM referente ao parágrafo será equivalente ao HTML a seguir:

```
<p>this is a paragraph</p>
```

Os elementos ul e p que criamos ainda não fazem parte do DOM que aparece na página, portanto o nosso diagrama de árvore teria esses novos elementos flutuando no espaço, porém referenciados pelas suas variáveis jQuery associadas. Isso está sendo mostrado na figura 4.4.

Podemos adicioná-los à página ao selecionar o elemento com o qual queremos associá-los e, em seguida, chamando a função append do objeto jQuery. Por exemplo, se quisermos adicionar esse elemento como filho do elemento footer, podemos fazer o seguinte:

```
$("footer").append($newParagraphElement);
```

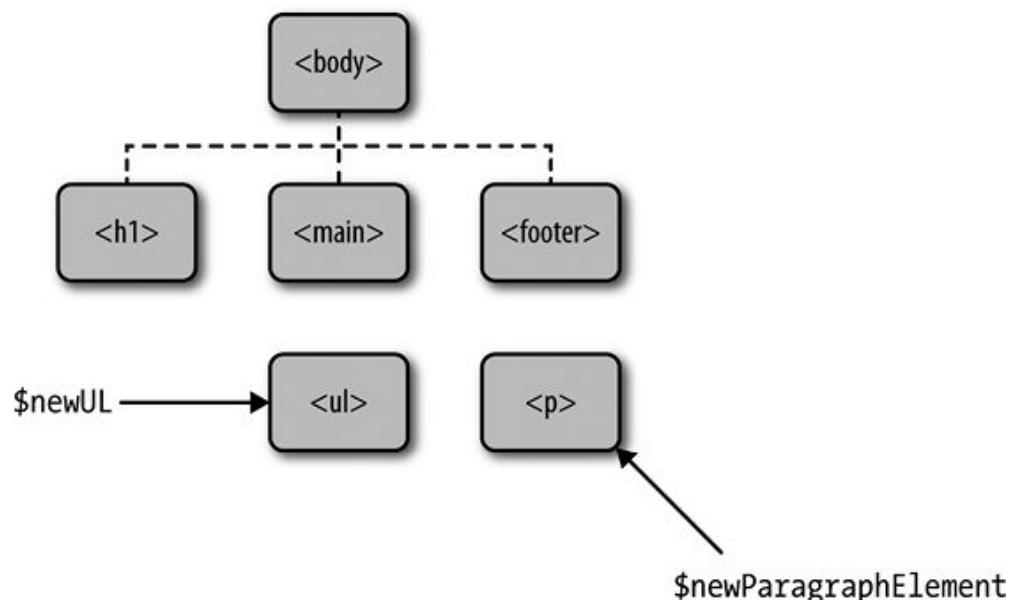


Figura 4.4 – Após a criação dos elementos `ul` e `p`, porém sem a associação ao DOM.

Agora o nosso conteúdo deve aparecer no rodapé porque o elemento está conectado no diagrama de árvore! A aparência será semelhante àquela mostrada na figura 4.5.

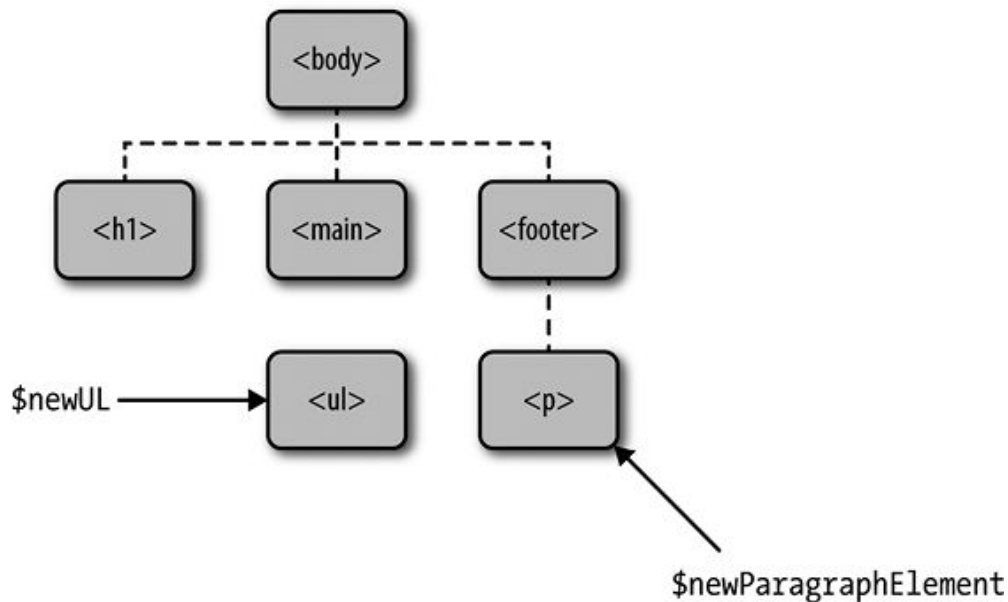


Figura 4.5 – O DOM após o elemento `p` ter sido adicionado ao rodapé.

Podemos criar estruturas DOM muito mais complicadas antes de fazê-las aparecer na página. Por exemplo, podemos adicionar elementos `li` ao elemento `ul` criado anteriormente:

```
// podemos encadear a criação e a chamada para a adição do texto
var $listItemOne = $("<li>").text("this is the first list item");
var $listItemTwo = $("<li>").text("second list item");
var $listItemThree = $("<li>").text("OMG third list item");

// agora iremos concatenar esses elementos ao elemento ul criado anteriormente
$newUL.append($listItemOne);
$newUL.append($listItemTwo);
$newUL.append($listItemThree);
```

A essa altura, temos uma nova *subárvore* desconectada do restante da árvore. A aparência será semelhante àquela mostrada na figura 4.6.

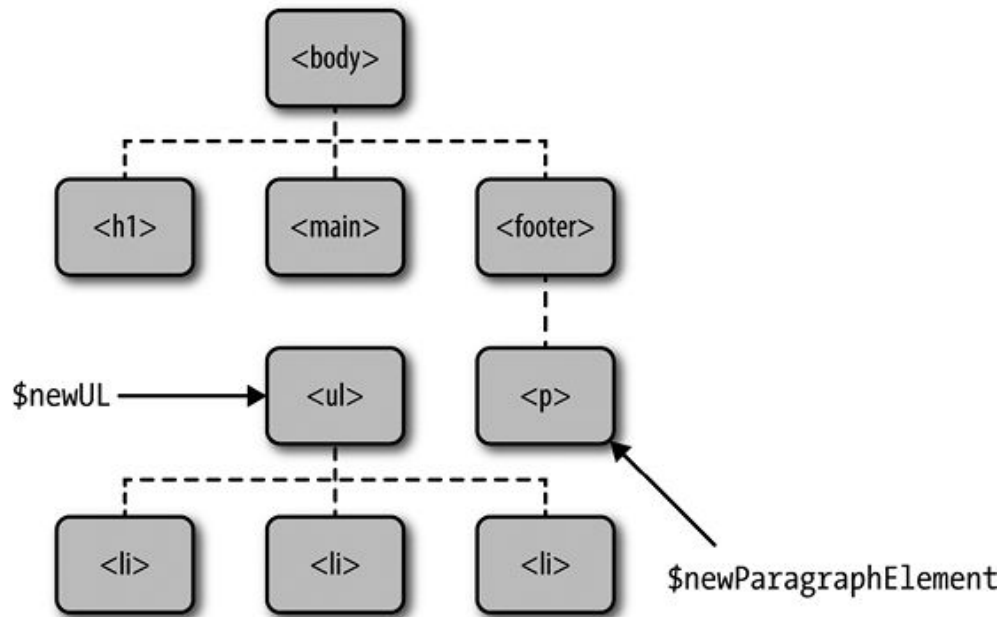


Figura 4.6 – O DOM com uma subárvore desconectada que representa um elemento `ul`.

Agora suponha que desejamos associar essa subárvore ao corpo como parte do elemento `main`. Seguimos o mesmo padrão usado antes, porém devemos associar somente a *raiz* à nossa subárvore!

```
$("main").append($newUL);
```

Agora a nossa árvore do DOM tem o aspecto apresentado na figura 4.7.

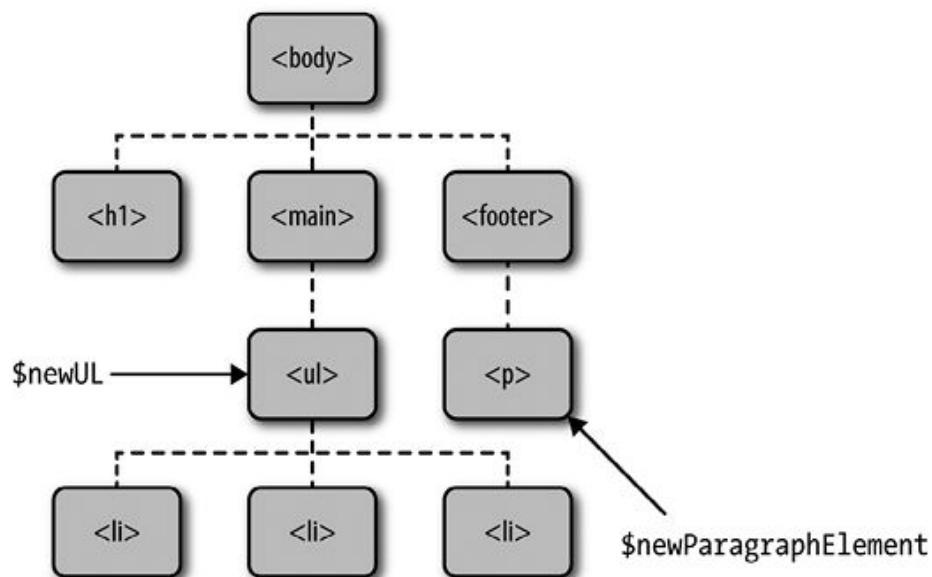


Figura 4.7 – O DOM, com a subárvore ul agora conectada.

A jQuery provê certa flexibilidade na adição de elementos quando um elemento do DOM está selecionado. Por exemplo, podemos inserir elementos na frente, em vez de concatená-los, o que os torna o primeiro filho de um nó pai:

```
var $footerFirstChild = $("<p>").text("I'm the first child of the footer!");
$("footer").prepend($footerFirstChild);
```

Além disso, podemos usar `appendTo` ou `prependTo` para mudar a maneira pela qual o nosso código é lido:

```
// isto é equivalente à linha anterior
$footerFirstChild.appendTo($("footer"));
```

Não se esqueça de que também podemos usar a aba Elements do Developer Tools do Chrome para confirmar se o nosso DOM está atualizado, conforme descrito na seção anterior.

Removendo elementos do DOM

Remover elementos é quase tão fácil quanto adicioná-los, e a jQuery disponibiliza algumas maneiras de fazer isso. Uma opção é simplesmente usar a função `remove` em um seletor, o que removerá o(s) elemento(s) selecionado(s) do DOM:

```
// remove o primeiro item da lista
// criada anteriormente
$("ul li:first-child").remove();
```

Também podemos apagar todos os filhos de um elemento usando a função `empty` em um objeto jQuery. Por exemplo, para limpar os itens da lista criada no exemplo anterior, podemos fazer o seguinte:

```
// remove todos os filhos da
// lista criada anteriormente
$newUL.empty();
```

No primeiro exemplo, aprendemos que podemos usar `fadeOut` para fazer com que um elemento invisível apareça gradualmente. Além disso, podemos fazer os elementos desaparecerem usando `fadeOut` ou `slideUp` antes de removê-los do DOM!

```
// isto removerá o parágrafo do rodapé contido no DOM.
```

```
$("#footer p").fadeOut();
```

O fato é que `fadeOut` não remove realmente o elemento do DOM – ele simplesmente o oculta. Para removê-lo, devemos chamar `remove` no elemento, porém isso deve ser feito *após* o processo de fade out ter sido concluído. Isso significa que devemos agendar para que essa tarefa aconteça assincronamente, assunto sobre o qual aprenderemos na próxima seção.

Eventos e programação assíncrona

No primeiro exemplo deste capítulo, vimos como processar tanto o evento *click* (clique) quanto o evento *keypress* (pressionamento de tecla). Em geral, podemos processar qualquer evento utilizando o padrão `on`. Por exemplo, podemos responder a um evento `dblclick`, que é disparado quando um usuário efetua um clique duplo:

```
$("#.button").on("dblclick", function () {  
    alert("Hey! You double-clicked!");  
});
```

Em geral, esse estilo de programação chama-se programação *orientada a eventos* (event-driven) ou *assíncrona* (asynchronous). De que maneira ela difere da programação tradicional? Normalmente, estamos acostumados com o nosso código sendo executado da seguinte maneira:

```
console.log("this will print first");  
console.log("this will print second");  
console.log("this will print third");
```

Na programação assíncrona, determinadas funções relacionadas a eventos aceitam *callbacks* como argumentos, ou seja, funções que serão executadas posteriormente. Isso faz com que a ordem de execução nem sempre seja tão clara:

```
console.log("this will print first");  
$("#button").on("click", function () {  
    console.log("this will only print when someone clicks");  
});  
console.log("this will print second");
```

A seguir, temos outro exemplo que não depende de entradas do usuário para que as callbacks sejam executadas. Já vimos a função `ready`, que espera o DOM estar pronto para executar a callback. A função `setTimeout` se comporta de modo semelhante, porém executa a callback após o número especificado de milissegundos ter passado:

```
// Este é um evento jQuery que executa a callback
// quando o DOM estiver pronto. Neste exemplo, estamos usando
// uma função anônima em vez de enviar a função
// principal como argumento.
$(document).ready(function () {
    console.log("this will print when the document is ready");
});

// Esta é uma função pronta do JavaScript que executa
// após o número especificada de milissegundos ter passado.
setTimeout(function () {
    console.log("this will print after 3 seconds");
}, 3000);

// Isto será exibido antes de tudo o mais, embora esteja no final.
console.log("this will print first");
```

A essa altura, a programação orientada a eventos baseada na interação com o usuário provavelmente faz sentido, e os exemplos envolvendo `setTimeout` e a função `ready` devem ser relativamente fáceis de compreender. Os problemas começam a surgir quando queremos sequenciar os eventos. Por exemplo, considere uma situação em que estamos usando a função `slideDown` da jQuery para efetuar a animação de uma porção de texto que deslize para baixo:

```
var main = function () {
    "use strict";

    // cria e oculta o nosso conteúdo na forma de uma div
    var $content = $("

Hello World!</div>").hide();

    // adiciona o conteúdo ao elemento body
    $("body").append($content);

    // desliza o conteúdo para baixo durante 2 segundos
    $content.slideDown(2000);
};

$(document).ready(main);


```

Agora suponha que queremos fazer o fade in de uma segunda mensagem *após* o conteúdo ter deslizado para baixo. Podemos, de imediato, tentar fazer algo como:

```
var main = function () {  
    "use strict";  
  
    // cria e oculta o nosso conteúdo na forma de uma div  
    var $content = $("<div>Hello World!</div>").hide();  
    var $moreContent = $("<div>Goodbye World!</div>").hide();  
  
    // adiciona o conteúdo ao elemento body  
    $("body").append($content);  
  
    // desliza o conteúdo para baixo durante 2 segundos  
    $content.slideDown(2000);  
  
    // adiciona o segundo conteúdo a body  
    $("body").append($moreContent);  
  
    // faz o fade in do segundo conteúdo  
    $moreContent.fadeIn();  
}  
  
$(document).ready(main);
```

Digite esse código e execute-o. Você verá que a div “Goodbye World!” faz o fade in *enquanto* a div “Hello World!” está deslizando para baixo. Não é isso que queríamos. Pare e pense no motivo pelo qual isso está acontecendo.

Provavelmente, você deve ter percebido que a função `slideDown` está sendo executada assincronamente. Isso significa que o código que se segue está sendo executado *enquanto* o deslizamento para baixo está ocorrendo! Felizmente, a jQuery nos oferece uma solução alternativa – a maioria das funções assíncronas aceita um parâmetro opcional de callback como o seu último argumento, o que nos permite sequenciar eventos assíncronos. Desse modo, podemos conseguir o efeito pretendido ao modificar o nosso código da seguinte maneira:

```
var main = function () {  
    "use strict";  
  
    // cria e oculta o nosso conteúdo na forma de uma div
```



```

var $content = $("<div>Hello World!</div>").hide();
var $moreContent = $("<div>Goodbye World!</div>").hide();

// adiciona o conteúdo ao elemento body
$("body").append($content);

// desliza o conteúdo para baixo durante 2 segundos
// e então execute a callback que
// contém o segundo conteúdo
$content.slideDown(2000, function () {
    // adiciona o segundo conteúdo a body
    $("body").append($moreContent);

    // faz o fade in do segundo conteúdo
    $moreContent.fadeIn();
});
};

$(document).ready(main);

```

Usaremos a mesma abordagem para finalizar o nosso exemplo da seção anterior. Para remover o elemento `p` de `footer` quando o `fade out` terminar, faremos algo como:

```

$("footer p").fadeOut(1000, function () {
    // isto acontecerá quando o elemento p
    // terminar de desaparecer (fade out)
    $("footer p").remove();
});

```

Mais adiante neste livro, veremos outros exemplos de programação assíncrona com o Node.js, mas, por enquanto, é bom ter domínio sobre essas funções e compreender os padrões.

Generalizações para o JavaScript

Vários desenvolvedores de frontend sabem muito sobre HTML e CSS, além de conhecerem o suficiente de jQuery para fazer os plug-ins funcionarem e realizarem algumas manipulações básicas com o DOM. Um conhecimento desses três assuntos pode levar você bem longe no desenvolvimento de frontends web. Por outro lado, é uma boa ideia aprender o máximo possível de JavaScript para ser um desenvolvedor altamente eficaz de frontends e criar aplicações web

mais complexas.

Nesta seção, analisaremos alguns elementos básicos do JavaScript. Meu plano é manter tudo resumido e focar nos aspectos mais importantes da linguagem. Veremos um exemplo que reunirá todas essas ideias no final do capítulo.

Interagindo com o JavaScript no JavaScript Console do Chrome

O fato é que não é necessário criar todo um projeto para começar a trabalhar com o JavaScript. O Chrome contém um interpretador JavaScript interativo muito bom incluído em seu Developer Tools (Ferramentas do desenvolvedor)!

Vamos começar abrindo uma nova janela ou aba em nosso navegador e abrindo o Developer Tools, do Chrome, conforme descrito no capítulo anterior [acesse View → Developer (Visualizar → Desenvolvedor); depois clique em Developer Tools (Ferramentas do desenvolvedor)]. A seguir, clique na aba Console na parte superior. Você deverá ver um prompt semelhante àquele mostrado na figura 4.8.

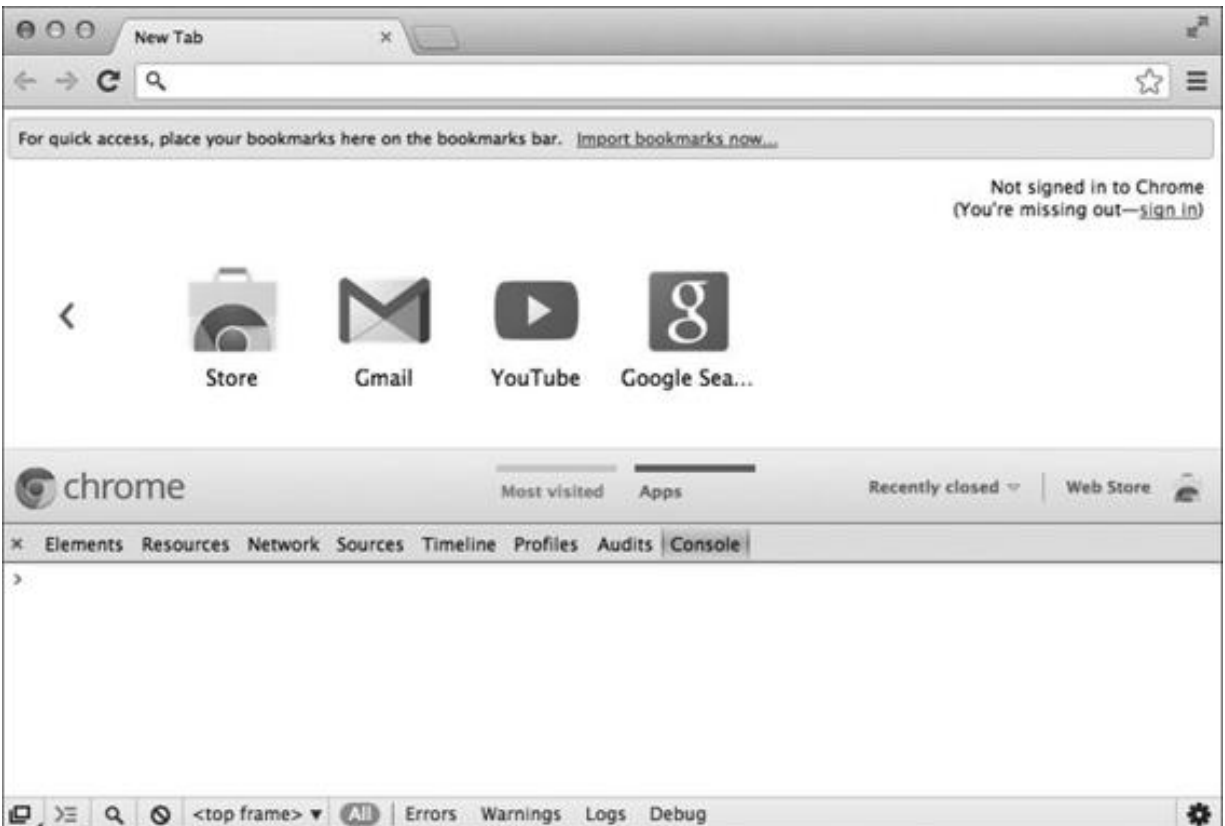


Figura 4.8 – O JavaScript Console do Chrome.

Agora podemos interagir com o JavaScript Console. Por exemplo, digite o código a seguir, teclando Enter ao final de cada linha:

```
5+2;  
//=> 7  
  
Math.random();  
//=> 0.3896130360662937  
  
console.log("hello world!");  
//=> hello world!
```

A primeira linha deve exibir 7, pois a expressão numérica é avaliada como 7. A segunda linha deve exibir um número decimal aleatório entre 0 e 1.

A última linha deve exibir “hello world!”, seguido de uma linha contendo “undefined”. Isso não é um problema – o Chrome sempre mostra o resultado da avaliação da última linha. A instrução `console.log` realiza uma tarefa, porém não é realmente *avaliada* como nada, motivo pelo qual o Chrome exibe `undefined`.

No restante desta seção, você poderá digitar os exemplos diretamente no JavaScript Console, porém com uma pequena ressalva. Se for necessário digitar algo que ocupe várias linhas (por exemplo, uma instrução `if` ou um laço `for`), você deverá teclar Shift-Enter no lugar de teclar somente Enter no final da linha. Faça uma experiência digitando o código a seguir:

```
var number = 5;
if (number >= 3) {
  console.log("The number is bigger than or equal to 3!");
}
//=> The number is bigger than or equal to 3!
```

Observe também que você pode pressionar as teclas correspondentes às setas para cima e para baixo em seu teclado para navegar pelo trechos de código anteriormente digitados (da mesma maneira que você pode navegar pelo histórico de comandos em sua janela do terminal).

Variáveis e tipos

O JavaScript não é uma linguagem com tipagem forte. Isso significa que as variáveis podem armazenar dados de qualquer tipo (como inteiros, strings e números com casas decimais). O JavaScript é diferente de linguagens como o Java ou o C++, em que as variáveis devem ser declaradas para armazenar tipos específicos:

```
// Esta variável armazena uma string
var message = "hello world!";

// Estas variáveis armazenam números
var count = 1;
var pi = 3.1415;

// Esta variável armazena um booleano
var isFun = true;

console.log(message);
//=> hello world!

console.log(pi);
//=> 3.1415
```

```
console.log(isFun);  
//=> true
```

As declarações/definições de variáveis podem ser feitas como uma única instrução, separadas por vírgulas. Conforme mencionado anteriormente, é uma boa ideia colocá-las no início da definição de sua função. A convenção gera consistência que, por sua vez, contribui para a legibilidade de seu código:

```
var main = function () {  
  "use strict";  
  
  var message = "hello world!",  
      count = 1,  
      pi = 3.1415,  
      isFun = true;  
  
  console.log(message);  
};
```

Funções

Já vimos vários exemplos de funções até agora. De modo diferente do C++ e do Java, as funções são cidadãos de *primeira classe*. Isso significa que podemos atribuir as funções a variáveis, enviar funções como parâmetros para outras funções e definir funções *anônimas* (que são simplesmente funções sem nome):

```
// define uma função e a armazena em uma variável chamada sayHello  
var sayHello = function () {  
  console.log("hello world!");  
}  
// executa a função que está na variável sayHello  
sayHello();  
//=> "hello world!"
```

De modo semelhante às demais linguagens, as funções JavaScript podem ter *entradas* e uma *saída*. As entradas normalmente são chamadas de *argumentos* ou *parâmetros* e são especificadas entre parênteses na definição da função. A saída geralmente é chamada de *valor de retorno* e é sempre precedida da palavra-chave `return`:

```
// define uma função chamada add que
```

```
// aceita duas entradas: num1 e num2
// e tem uma saída: a soma dos
// dois números
var add = function (num1, num2) {
    // soma as entradas e armazena o resultado em sum
    var sum = num1 + num2;

    // retorna a soma
    return sum;
}
// executa a função add com 5 e 2 como entradas
add(5,2);
//=> 7
```

Uma consequência interessante do fato de as funções serem objetos de primeira classe em JavaScript é que podemos usar outras funções como entradas de funções. Já utilizamos esse padrão ao enviar funções anônimas como callbacks, porém funções nomeadas também podem ser utilizadas. Por exemplo, enviamos uma função chamada `main` à função `ready` na jQuery:

```
// o ponto de entrada principal de nosso programa
var main = function () {
    "use strict";
    console.log("hello world!");
};

// configura o main para ser executado depois que o DOM estiver pronto
$(document).ready(main);
```

Embora tenha ponteiros de função, o C++ não admite funções anônimas facilmente, de modo que padrões de código como esses que acabamos de ver não são usados com frequência. De modo semelhante, ao menos na época desta publicação, o Java não tinha nenhum mecanismo simples para criar funções que aceitassem outras funções como parâmetros.

Nos próximos capítulos, teremos a oportunidade de criar funções que aceitem outras funções como parâmetros. Isso será útil quando estivermos trabalhando com programações assíncronas mais complexas.

Seleção

Uma das primeiras estruturas de controle que aprendemos em qualquer linguagem é a instrução `if`. Essa instrução nos permite dizer ao interpretador que um bloco de código deve ser executado somente se alguma condição for verdadeira:

```
var count = 101;
if (count > 100) {
  console.log("the count is bigger than 100");
}
//=> the count is bigger than 100

count = 99;
if (count > 100) {
  console.log("the count is bigger than 100");
}
//=> não mostra nada
```

Uma instrução `else` nos permite fazer algo diferente caso a condição seja falsa:

```
var count = 99;
if (count > 100) {
  console.log("the count is bigger than 100");
} else {
  console.log("the count is less than 100");
}
//=> the count is less than 100
```

Às vezes, perceberemos que é necessário realizar várias tarefas mutuamente exclusivas de acordo com diferentes condições. Quando isso ocorrer, o padrão `if-else-if` normalmente será bastante apropriado:

```
var count = 150;
if (count < 100) {
  console.log("the count is less than 100");
} else if (count <= 200) {
  console.log("the count is between 100 and 200 inclusive");
} else {
  console.log("the count is bigger than 200");
}
```

```
}  
//=> the count is between 100 and 200 inclusive
```

O melhor de tudo é que as condições não precisam ser simples. Podemos utilizar os operadores `&&` (e), `||` (ou) e `!` (de negação) para criar condições mais complexas. Conforme mencionado no capítulo 2, a tecla `|` encontra-se junto à tecla de barra invertida em seu teclado – é preciso pressionar Shift para utilizá-la:

```
// verifica se *alguma* das condições é verdadeira  
if (cardRank === "king" || cardRank === "queen" || cardRank === "jack") {  
    console.log("that's a high ranking card!");  
} else {  
    console.log("not quite royalty!");  
}  
  
// verifica se a carta é um ás de espadas  
if (cardRank === "ace" && cardSuit === "spades") {  
    console.log("THAT'S THE ACE OF SPADES!");  
} else {  
    console.log("Sadly, that's not the ace of spades");  
}  
  
{// verifica se a carta *não é* um ás de espadas  
// invertendo o resultado com o operador !  
if (!(cardRank === "ace" && cardSuit === "spades")) {  
    console.log("That card is not the ace of spades!");  
}  
}
```



Em JavaScript, um único sinal de igual representa uma instrução de atribuição, enquanto três sinais de igual representam uma comparação que retorna true ou false se os lados esquerdo e direito da expressão forem equivalentes. Discutiremos mais a respeito desse assunto posteriormente.

Iteração

Com frequência, precisamos executar certas tarefas diversas vezes. Por exemplo, suponha que desejamos exibir os cem primeiros números. Obviamente, podemos fazer isso de uma maneira bastante ingênua:

```
console.log(0);  
console.log(1);  
console.log(2);
```



```
// ... ugh  
console.log(99);  
console.log(100);
```

É muito código! É muito mais fácil utilizar uma estrutura de laço para exibir todos os cem números:

```
var num; // esse será o número a ser exibido  
// exibe 101 números, começando pelo 0  
for (num = 0; num <= 100; num = num + 1) {  
    console.log(num);  
}
```

Esse código apresenta o mesmo resultado que o código anterior, porém de maneira muito mais concisa, e constitui um exemplo de um laço `for`.

Um laço `for` é composto de quatro itens. Três deles estão entre parênteses que vêm depois da palavra `for`, e eu me refiro a eles como *instrução de inicialização* (initialization statement), *condição de continuidade* (continuation condition) e *instrução de atualização* (update statement). O quarto item é o *corpo do laço* (loop body), que é o código que está entre chaves. A instrução de inicialização ocorre imediatamente antes de o corpo do laço ser executado pela primeira vez. A instrução de atualização é executada sempre que o corpo do laço é concluído. E a condição de continuidade é verificada imediatamente antes de o corpo do laço ser executado (até mesmo da primeira vez).

Aqui estão dois laços `for` com comentários, que apresentam o mesmo resultado – exibem os números pares menores do que cem:

```
var i;  
// inicialização: i é definido com 0  
// continuidade: continue enquanto i for menor que 100  
// atualização: some 2 a i  
// corpo: exibe i  
// em outras palavras, exibe somente os números pares, começando com 0 e  
// terminando com 98  
for (i = 0; i < 100; i = i + 2) {  
    console.log(i);  
}
```

```

}
// inicialização: i é definido com 0
// continuidade: continue enquanto i for menor que 100
// atualização: some 1 a i
// corpo: exibe i somente se o resto da divisão de i por 2 for igual a 0
// exibe somente os números pares, começando com 0 e terminando com 98
for (i = 0; i < 100; i = i + 1) {
  if (i%2 === 0) {
    console.log(i);
  }
}

```

No segundo exemplo, utilizamos o operador de resto (%), que fornece o resto de uma divisão de inteiros. Em outras palavras, $5\%2$ é avaliado como 1 porque $5/2$ é 2, com um resto igual a 1. Nesse exemplo, usamos esse operador para verificar se o número é divisível por 2 (o que quer dizer que o número é par). Apesar de esse operador poder *parecer* exótico, na realidade, ele é extremamente útil. Com efeito, ele é tão útil para os programadores experientes que perguntas envolvendo esse assunto, com frequência, são feitas em entrevistas de emprego para cargos de programadores (veja o exercício com o problema FizzBuzz no final deste capítulo).

Se você estiver familiarizado com os laços `while` e `do...while` do Java ou do C++, ficará feliz em saber que o JavaScript também os suporta. Gosto de manter a simplicidade, portanto usarei somente laços `for` (e laços `forEach` em arrays) no restante deste livro.

Arrays

Os laços em geral são interessantes, porém são incrivelmente úteis no contexto dos *arrays*. Os arrays são simplesmente coleções indexadas de entidades JavaScript. Uma maneira de pensar nos arrays é imaginá-los como uma única variável que pode armazenar diversos valores. Aqui está um exemplo simples de criação de um array:

```

var greetings = ["hello", "namaste", "hola", "salut", "aloha"];

```

É fácil generalizar esse exemplo: podemos criar um array literal usando colchetes e, em seguida, listar os elementos, separando-os com vírgulas. Esse array contém cinco elementos, em que cada um deles é uma string. Aqui está um exemplo de um array de inteiros:

```
var primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
```

Podemos indexar os arrays para acessar determinados elementos usando o nome da variável, seguido de colchetes. A indexação de arrays sempre tem início em 0 e termina com o tamanho do array menos um.

```
console.log(greetings[1]);  
//=> 'namaste'  
  
console.log(greetings[0]);  
//=> 'hello'  
  
console.log(primes[4]);  
//=> 11  
  
console.log(greetings[4]);  
//=> 'aloha'
```

De modo semelhante, podemos definir elementos individuais do array usando o mesmo truque da indexação:

```
greetings[3] = "bonjour"; // altera 'salut' para 'bonjour'
```

E se quiséssemos apresentar todos os elementos do array? Uma abordagem consiste em usar a propriedade `length` do array para criar uma condição de continuidade em um laço `for`:

```
var index;  
  
for (index = 0; index < greetings.length; index = index + 1) {  
    console.log(greetings[index]);  
}
```

Essa é uma abordagem perfeitamente válida, usada com frequência no JavaScript, além de ser usada em outras linguagens, por exemplo, no Java. No entanto o JavaScript tem uma maneira *peculiar*, um pouco mais interessante, de obter o mesmo resultado. Todo array tem um laço `forEach` associado, o qual recebe uma função que opera sobre cada elemento:

```
// o laço forEach recebe uma função como argumento
// e a chama para cada elemento do array
greetings.forEach(function (element) {
    console.log(element);
});
```

Essa solução é mais elegante porque elimina a necessidade de manter uma variável extra, como `index` no exemplo anterior. Normalmente, ao remover uma declaração de variável, o nosso código estará menos suscetível a erros.

Além de incluir a função `forEach`, os arrays JavaScript apresentam algumas vantagens adicionais em relação aos arrays puros do C++ ou do Java. Em primeiro lugar, eles podem aumentar e diminuir de tamanho dinamicamente. Em segundo, eles têm várias funções incluídas que nos permitem executar algumas operações comuns. Por exemplo, com frequência, vamos querer adicionar elementos no final de nosso array. Podemos utilizar a função `push` para isso:

```
// cria um array vazio
var cardSuits = [];

cardSuits.push("clubs");
console.log(cardSuits);
//=> ["clubs"]

cardSuits.push("diamonds");
console.log(cardSuits);
//=> ["clubs", "diamonds"]

cardSuits.push("hearts");
console.log(cardSuits);
//=> ["clubs", "diamonds", "hearts"]

cardSuits.push("spades");
console.log(cardSuits);
//=> ["clubs", "diamonds", "hearts", "spades"]
```

Há várias outras funções prontas nos arrays JavaScript, porém ficaremos somente com a função `push` por enquanto. Aprenderemos sobre mais algumas funções à medida que prosseguirmos.

Os arrays e os laços são ferramentas muito importantes, independentemente da linguagem de programação com a qual você

estiver trabalhando. A única maneira de dominá-los é exercitar a sua criação e o seu uso. Vários problemas que servem como exercício no final deste capítulo farão você criar e manipular arrays utilizando laços.

Usando o JSLint para identificar possíveis problemas

Assim como o HTML e o CSS, é bem fácil escrever um código JavaScript que *funcione*, mas que não esteja em conformidade com as melhores práticas. Com efeito, isso é realmente bem fácil de fazer ao começar a trabalhar com a linguagem. Por exemplo, considere o código a seguir:

```
cardRank = "king";  
if (cardRank = "king") {  
    console.log("the card is a king!");  
} else {  
    console.log("the card is not a king!");  
}  
//=> the card is a king!
```

À primeira vista, esse código passa a impressão de ser adequado e *parece* funcionar! Digite-o no JavaScript Console do Chrome e você verá o resultado esperado! Entretanto vamos alterar um pouco o código – vamos definir `cardRank` com “queen”:

```
cardRank = "queen";  
if (cardRank = "king") {  
    console.log("the card is a king!");  
} else {  
    console.log("the card is not a king!");  
}  
//=> the card is a king!
```

Agora o erro provavelmente está um pouco mais evidente. O problema é que a nossa instrução `if` contém uma instrução de atribuição em vez de ter uma comparação, e a instrução de atribuição está sendo avaliada com um valor *verdadeiro* (truthy).

Não é essencial compreender o que é um valor *truthy* a esta altura ou por que ele faz a instrução `if` executar o seu bloco de código, porém devemos entender que o erro resulta do fato de termos utilizado acidentalmente `=` (atribuição) no lugar de `===` (comparação).

Podemos corrigir esse problema ao trocar a atribuição por uma comparação:

```
if (cardRank === "king") {  
    console.log("the card is a king!");  
} else {  
    console.log("the card is not a king!");  
}
```

Na realidade, há outro problema sutil nesse código. Felizmente, há uma ferramenta JavaScript semelhante ao CSS Lint, que se chama JSLint (o que não é de surpreender). A página inicial do JSLint está sendo mostrada na figura 4.9.

Vamos recortar e colar o nosso código original e inseri-lo no JSLint (<http://www.jslint.com/>) para ver o que ele nos diz. Ele deve apresentar uma resposta conforme a mostrada na figura 4.10.

Os primeiros erros que vemos estão relacionados ao erro sutil mencionado anteriormente. O JavaScript não exige, de forma alguma, que declaremos as variáveis antes de utilizá-las, porém, com frequência, isso pode resultar em consequências indesejadas. É melhor garantir que todas as variáveis sejam declaradas com a palavra-chave `var`. Portanto vamos modificar o nosso código de modo que a variável seja declarada, além de ser definida:

```
var cardRank = "king";
```



Figura 4.9 – Página inicial do JSLint.



Figura 4.10 – Nossos primeiros erros no JSLint!

Após ter feito isso e submeter o nossa código novamente ao Lint,

veremos que os dois primeiros erros desaparecerão. O próximo erro refere-se ao problema da atribuição mencionado antes. Depois que esse erro for corrigido, restarão os dois erros finais.

Os dois últimos erros relacionam-se ao fato de a variável global `console` estar sendo usada antes de ser definida. Não podemos simplesmente adicionar a palavra `var` na frente dessa variável porque nós não a criamos. Em vez disso, podemos acessar as opções do JSLint e habilitar as opções globais `console`, `alert`, ...; desse modo esses erros desaparecerão.

Como ocorre com o CSS Lint, o JSLint é muito semelhante a ter algum programador profissional de JavaScript observando por cima de seus ombros enquanto você codifica. Em muitas ocasiões, os seus warnings (avisos) serão estranhos e será necessário certo trabalho para descobrir como corrigi-los (o Google é um ótimo ponto de partida para isso), porém vale muito a pena fazer isso, e você aprenderá a escrever um código JavaScript melhor, de modo muito mais eficiente.

Acrescentando interatividade ao Amazeriffic

Vamos reunir algumas dessas ideias em um exemplo. Expandiremos a nossa ideia para o Amazeriffic transformando-a em uma aplicação que controla uma lista de tarefas a serem feitas (todo list). Para isso, criaremos uma interface com três abas – a primeira irá mostrar a nossa lista de tarefas, com os itens mais recentes em primeiro lugar, conforme mostrado na figura 4.11.

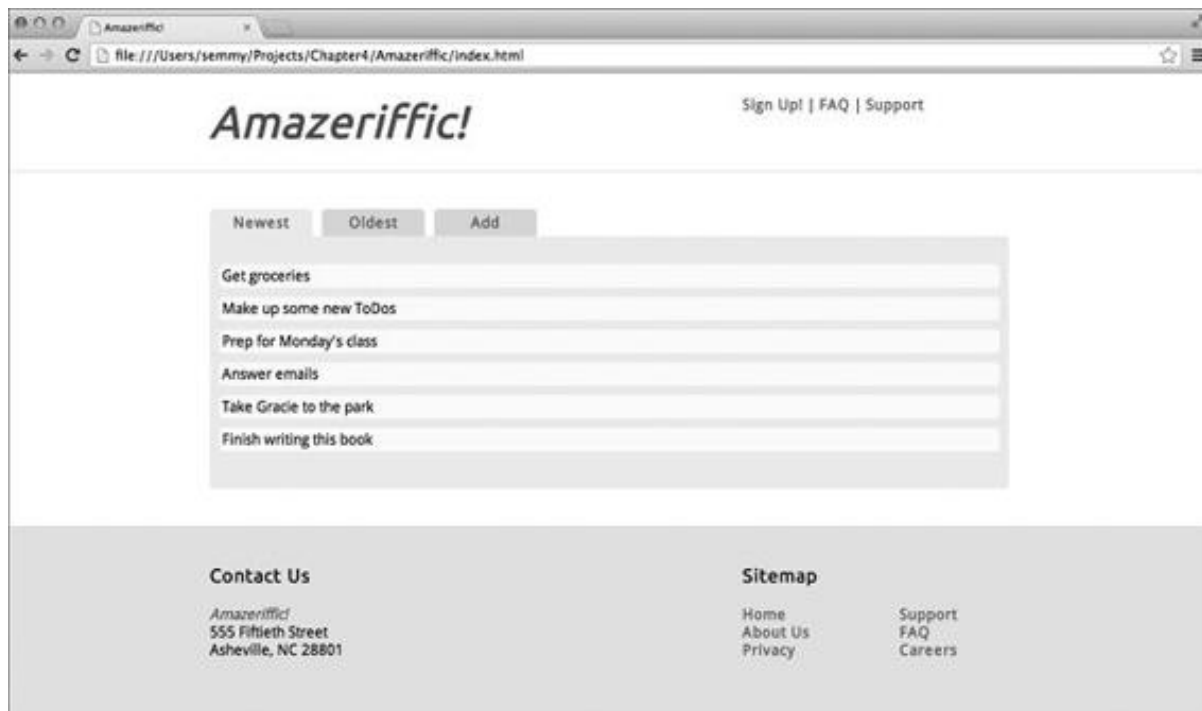


Figura 4.11 – Nossa primeira aba, com os itens da lista de tarefas ordenados do mais recente para o mais antigo.

A segunda aba mostrará a mesma lista, porém ela será apresentada com os itens mais antigos em primeiro lugar, como mostrado na figura 4.12.

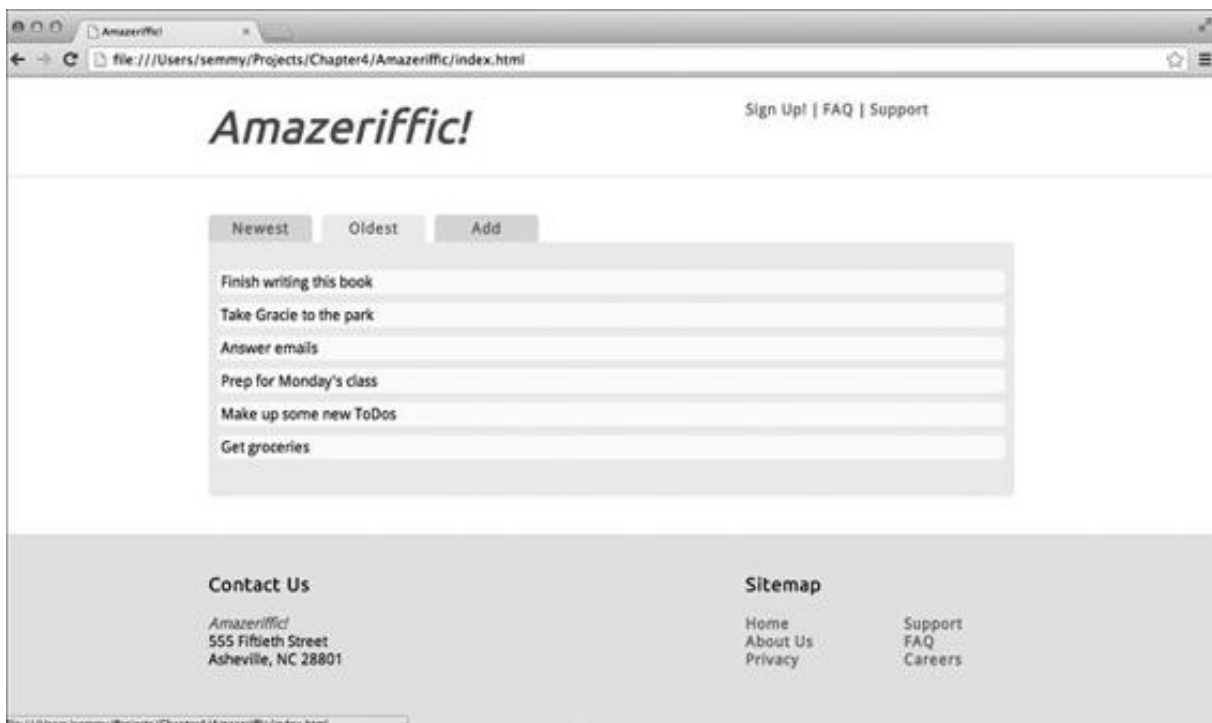


Figura 4.12 – Nossa segunda aba, com os itens da lista de tarefas ordenados do mais antigo para o mais recente.

E a última aba terá uma caixa de entrada por meio da qual poderemos adicionar novos itens na lista de tarefas, conforme mostrado na figura 4.13.

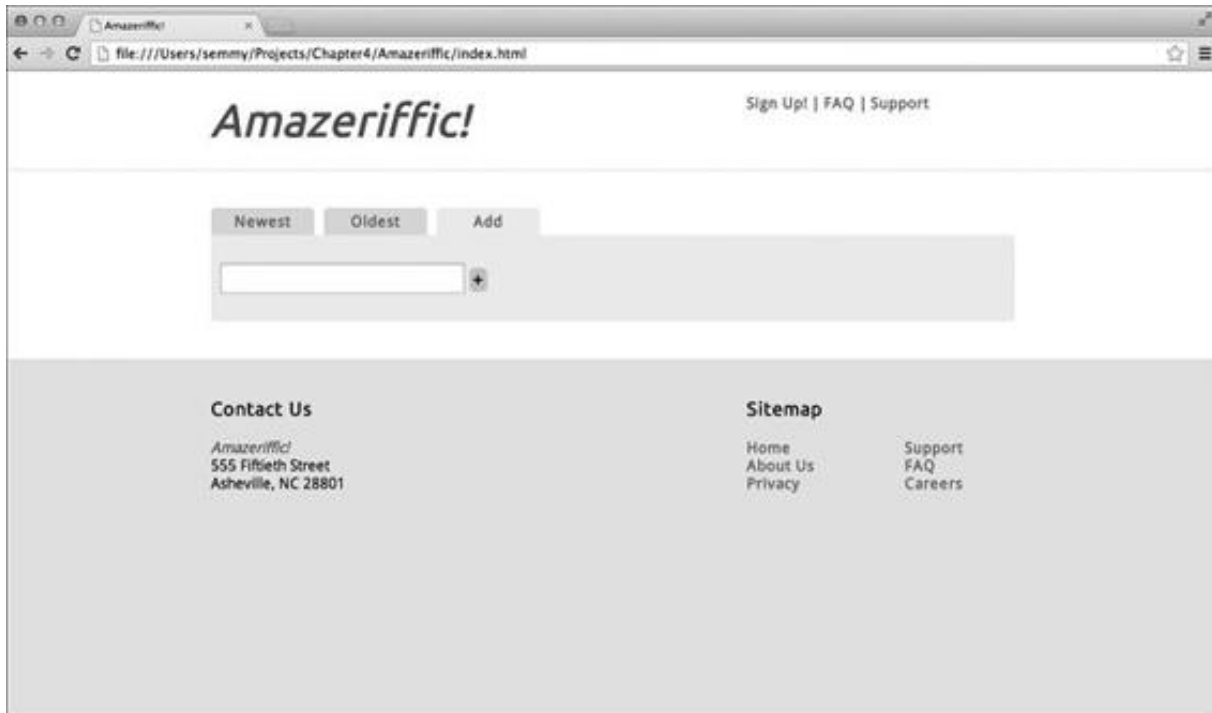


Figura 4.13 – A aba que nos permite adicionar outro item à lista de tarefas.

Iniciando

Como já vimos vários exemplos com o Git até agora, deixarei que você decida os momentos e os lugares apropriados para atualizar o seu repositório no caso desse exemplo. Só não se esqueça de realmente criar um repositório Git e atualizá-lo regularmente à medida que seguirmos adiante com o exemplo.

Você também verá que eu reutilizei boa parte do design do último exemplo que está no capítulo anterior. Desse modo, você poderá copiar o seu trabalho (ou seja, o HTML e o CSS) do capítulo anterior. Nesse caso, porém, queremos criar os diretórios *javascripts*

e *stylesheets* para manter o nosso código organizado.

A estrutura e o estilo

Podemos começar pela modificação do elemento `main` que contém a nossa UI. O meu HTML para o elemento `main` apresenta o seguinte aspecto:

```
<main>
  <div class="container">
    <div class="tabs">
      <a href="#"><span class="active">Newest</span></a>
      <a href="#"><span>Oldest</span></a>
      <a href="#"><span>Add</span></a>
    </div>
    <div class="content">
      <ul>
        <li>Get Groceries</li>
        <li>Make up some new Todos</li>
        <li>Prep for Monday's class</li>
        <li>Answer recruiter emails on LinkedIn</li>
        <li>Take Gracie to the park</li>
        <li>Finish writing book</li>
      </ul>
    </div>
  </div>
</main>
```

Essa estrutura básica pode ser utilizada para o elemento `main`, juntamente com o trabalho desenvolvido no capítulo 3, para estilizar a página. Aqui estão as regras de estilo para as minhas abas:

```
.tabs a span {
  display: inline-block;
  border-radius: 5px 5px 0 0;
  width: 100px;
  margin-right: 10px;
  text-align: center;
  background: #ddd;
  padding: 5px;
}
```

Você verá uma novidade aqui: eu defini a propriedade `display` com `inline-block`. Lembre-se de que os elementos `inline` como o `span` não possuem uma propriedade `width`, enquanto os elementos `block` têm, mas eles aparecem em uma linha nova. A configuração `inline-block` cria um elemento híbrido, fornecendo uma propriedade `width` ao `span`, a qual defini com `100px`. Isso faz com que todas as abas tenham a mesma largura.

É claro que também tenho um conjunto de regras extras para a aba ativa. O código a seguir faz a aba ativa ter a mesma cor que a cor de fundo do conteúdo, o que dá à interface um pouco de profundidade visual:

```
.tabs a span.active {  
  background: #eee;  
}
```

A interatividade

Vamos prosseguir para a parte realmente interessante: o JavaScript! É claro que, a essa altura, já devemos ter incluído tanto a jQuery quanto `/javascripts/app.js` por meio de um elemento `script` na parte final de nosso elemento `body` no HTML. Portanto podemos dar início ao `app.js` com o nosso programa que contém o esqueleto básico:

```
var main = function () {  
  "use strict";  
  console.log("hello world!");  
};  
$(document).ready(main);
```

Lembre-se de que, se abrirmos esse arquivo na janela de nosso navegador, devemos ver *hello world!* no JavaScript Console, se tudo estiver corretamente definido! Vá em frente e faça isso.

Criando a funcionalidade de abas

Se tudo estiver funcionando corretamente, vamos fazer nossas abas realizarem a sua função. Daremos início com uma porção de código que responda aos eventos clique em cada aba:

```

var main = function () {
    "use strict";

    $(".tabs a:nth-child(1)").on("click", function () {
        // deixa todas as abas inativas
        $(".tabs span").removeClass("active");

        // deixa a primeira aba ativa
        $(".tabs a:nth-child(1) span").addClass("active");

        // esvazia o conteúdo principal para que possamos recriá-lo
        $(".main .content").empty();

        // retorna false para não seguirmos o link
        return false;
    });

    $(".tabs a:nth-child(2)").on("click", function () {
        $(".tabs span").removeClass("active");
        $(".tabs a:nth-child(2) span").addClass("active");
        $(".main .content").empty();
        return false;
    });

    $(".tabs a:nth-child(3)").on("click", function () {
        $(".tabs span").removeClass("active");
        $(".tabs a:nth-child(3) span").addClass("active");
        $(".main .content").empty();
        return false;
    });
};

```

Se você estiver prestando bastante atenção, perceberá que violamos o princípio DRY nesse caso; definitivamente, há um pouco de código copiado e colado aqui. Isso nos diz que devemos abstrair certas partes desse código na forma de uma função. Experimente fazer isso agora, sem olhar para o que vem a seguir. Faça isso. Vou esperá-lo.

Refatorando o código usando uma função

Espero que você, pelo menos, tenha investido um tempo para pensar sobre o assunto antes de olhar a solução! Usar uma função que aceite um argumento que representa o número da aba foi uma ideia que lhe ocorreu? Se sim, ótimo! Do contrário, tudo bem; em

algum momento, você vai chegar lá. Isso apenas exige prática.

Não haverá problemas se a sua solução se parecer diferente da minha, mas o ponto principal é que você compreenda o funcionamento da minha solução:

```
var main = function () {  
    "use strict";  
  
    var makeTabActive = function (tabNumber) {  
        // constrói o seletor a partir de tabNumber  
        var tabSelector = ".tabs a:nth-child(" + tabNumber + ") span";  
        $(".tabs span").removeClass("active");  
        $(tabSelector).addClass("active");  
        $(".main .content").empty();  
    };  
  
    $(".tabs a:nth-child(1)").on("click", function () {  
        makeTabActive(1);  
        return false;  
    });  
  
    $(".tabs a:nth-child(2)").on("click", function () {  
        makeTabActive(2);  
        return false;  
    });  
  
    $(".tabs a:nth-child(3)").on("click", function () {  
        makeTabActive(3);  
        return false;  
    });  
};
```

Pode ser que você tenha tentado colocar o `return false` dentro da função `makeTabActive`, mas isso causará problemas. Devemos deixá-lo dentro do handler de cliques porque o listener *deve* retornar falso, pois, do contrário, o navegador tentará seguir o link.

Refatorando o código usando um laço

Reduzimos um pouco o número de linhas de código e, ao fazer isso, fizemos com que houvesse menos chance de cometermos erros. No entanto podemos fazer algo melhor ainda. Observe como criamos as três abas utilizando números: 1, 2, 3. Se empacotarmos o código

em um laço `for` que faça a iteração por esses números, poderemos remover facilmente mais código repetido.

No código a seguir, utilizei o objeto `event`, que é enviado ao handler de clique, como fizemos com os eventos `keyPress` anteriormente. Faço isso para que o elemento-alvo clicado do DOM acrescente a classe `active` a esse elemento. Observe que isso elimina totalmente a necessidade da função `makeTabActive`, pois estamos empacotando as linhas de código importantes no laço.

```
var main = function () {
    "use strict";

    var tabNumber;

    for (tabNumber = 1; tabNumber <= 3; tabNumber++) {
        var tabSelector = ".tabs a:nth-child(" + tabNumber + ") span";
        $(tabSelector).on("click", function (event) {
            $(".tabs span").removeClass("active");
            $(event.target).addClass("active");
            return false;
        });
    }
};
```

Refatorando o código usando um laço `forEach`

O fato é que existe ainda outra solução! A `jQuery` nos permite selecionar um conjunto de elementos e efetuar a iteração por ele como se fosse um array! Nessa simplificação, faremos a iteração por todos os nossos elementos `span` dentro das abas, criando um handler `click` para cada um:

```
var main = function () {
    "use strict";

    $(".tabs span").toArray().forEach(function (element) {
        // cria um handler click para este elemento
        $(element).on("click", function () {
            $(".tabs span").removeClass("active");
            $(element).addClass("active");
            $(".main .content").empty();
            return false;
        });
    });
};
```

```
});  
});  
};
```



O array criado pela jQuery é um array de elementos do DOM, e não de objetos jQuery. Devemos transformá-los em objetos jQuery encapsulando-os em uma chamada de função \$().

Então essas são as três maneiras de realizar a mesma tarefa. Pode até ser que haja outras soluções, mas acho que essas duas últimas são muito boas – são compactas, fáceis de compreender (quero dizer, se você entender de laços) e fazem com que seja fácil adicionar mais abas.

Manipulando o conteúdo

Agora temos outro problema para resolver. As abas irão preencher o elemento `main .content` com conteúdos diferentes, de acordo com a aba clicada. Isso será muito fácil se soubermos em qual filho dos elementos `.tabs` nós estamos. Por exemplo, se estivermos no primeiro filho do elemento `.tabs`, faremos uma tarefa, e se estivermos no segundo, faremos algo diferente. Mas acontece que isso é um pouco complicado porque, dentro de nosso handler click, temos acesso ao elemento `span`, que é filho de um elemento `a` no qual estamos interessados. Um diagrama de árvore de nossa situação tem um aspecto semelhante ao que está sendo mostrado na figura 4.14.

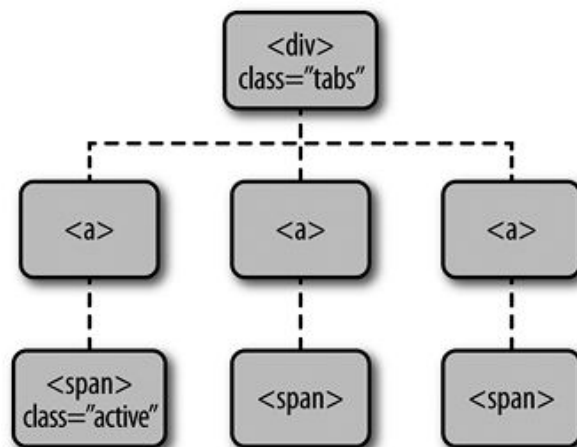


Figura 4.14 – Devemos saber o índice do elemento span no qual

clicamos.

Por acaso, a jQuery nos disponibiliza uma maneira realmente muito interessante de selecionar o pai de um objeto jQuery – é a função `parent`, cujo nome é bastante apropriado! Entretanto devemos determinar qual é o filho. A jQuery também nos disponibiliza a função `is`, que permite testar um seletor em relação ao objeto jQuery corrente. Provavelmente, isso é desafiador para entender de forma abstrata, porém torna-se bastante compreensível quando vemos isso em ação. Aqui está um exemplo rápido:

```
// testa se o pai do objeto jQuery $me
// é o primeiro filho de seu pai
if ($me.parent().is(":first-child")) {
    console.log("MY PARENT IS A FIRST CHILD!!");
} else {
    console.log("my parent is not a first child.");
}
```

Podemos utilizar esse padrão e o seletor `:nth-child` para determinar o que é preciso ser feito em nosso exemplo com abas:

```
var main = function () {
    "use strict";

    $(".tabs a span").toArray().forEach(function (element) {
        // cria um handler click para este elemento
        $(element).on("click", function () {
            // como estamos usando a versão jQuery do elemento,
            // devemos criar uma variável temporária
            // para que não seja necessário recriá-la continuamente
            var $element = $(element);

            $(".tabs a span").removeClass("active");
            $element.addClass("active");
            $(".main .content").empty();

            if ($element.parent().is(":nth-child(1)")) {
                console.log("FIRST TAB CLICKED!");
            } else if ($element.parent().is(":nth-child(2)")) {
                console.log("SECOND TAB CLICKED!");
            } else if ($element.parent().is(":nth-child(3)")) {
                console.log("THIRD TAB CLICKED!");
            }
        })
    })
}
```

```
        return false;
    });
});
};
```

Ao executar esse código com o JavaScript Console do Chrome aberto, você deverá ver a mensagem correta aparecer quando a aba associada for clicada! Muito fácil!

Definindo o conteúdo da aba

Definir o conteúdo da aba exige um pouco mais de trabalho. Em primeiro lugar, começaremos armazenando as tarefas propriamente ditas na forma de strings em um array. Para que isso aconteça, adicionaremos uma variável em nossa função `main`, que armazenará um array literal com os itens de nossa lista de tarefas:

```
var main = function () {
    "use strict";

    var toDos = [
        "Finish writing this book",
        "Take Gracie to the park",
        "Answer emails",
        "Prep for Monday's class",
        "Make up some new ToDos",
        "Get Groceries"
    ];

    //... outros códigos relacionados às abas
};
```

Agora, quando adicionarmos as tarefas, tudo o que deveremos fazer é inseri-las no final do array. Isso resulta nas tarefas mais antigas sendo mantidas no início do array, o que significa que as mais recentes ficarão no final. Desse modo, a segunda aba (Oldest) listará as tarefas na ordem em que elas estão no array, enquanto a primeira aba as listará na ordem inversa.

Para começar, vou mostrar como definir o conteúdo quando a segunda aba for clicada e deixarei a criação do conteúdo das outras abas ao seu cargo. Para criar o conteúdo da primeira e da segunda

abas, iremos simplesmente criar um elemento `ul` e, em seguida, percorreremos as tarefas em um laço, adicionando um elemento `li` a cada uma delas. Observe, porém, que na primeira aba devemos executar o laço percorrendo os elementos na ordem inversa; sendo assim, usaremos um laço `for` tradicional nesse caso. Contudo, na segunda aba, podemos usar o laço `forEach`, que é preferível. O código será alterado para algo como:

```
$(element).on("click", function () {
    var $element = $(element),
        $content;

    $(".tabs a span").removeClass("active");
    $element.addClass("active");
    $("main .content").empty();

    if ($element.parent().is(":nth-child(1)")) {
        console.log("FIRST TAB CLICKED!");
    } else if ($element.parent().is(":nth-child(2)")) {
        $content = $("<ul>");
        toDos.forEach(function (todo) {
            $content.append($("<li>").text(todo));
        });
        $("main .content").append($content);
    } else if ($element.parent().is(":nth-child(3)")) {
        console.log("THIRD TAB CLICKED!");
    }
});
```

Conforme mencionado anteriormente, criar o conteúdo da primeira aba será um trabalho bem semelhante. Feito isso, não precisaremos mais do conteúdo fixo no HTML! Podemos efetuar o disparo de um clique *fictício* na primeira aba ao adicionar uma única linha no final de nossa função `main`, logo após a definição dos handlers para cliques. Isso fará o nosso conteúdo ser criado dinamicamente:

```
$(".tabs a:first-child span").trigger("click");
```

Depois que isso estiver corretamente definido, poderemos remover os elementos relacionados às tarefas, que estão fixos no código, em *index.html*.

No caso da terceira aba, será preciso fazer algo um pouco diferente – devemos criar um elemento `input` e um elemento `button`, do mesmo modo que fizemos no exemplo dos comentários no início deste capítulo. Porém, nesse exemplo, devemos exercitar a criação da subárvore do DOM usando a jQuery no lugar de incorporá-la no HTML. Também será necessário adicionar um listener de evento para o botão. Nesse caso, em vez de adicionar um elemento ao DOM, como fizemos no exemplo anterior, nós simplesmente o adicionaremos ao nosso array `todo` utilizando a função `push`.

Você pode ver o exemplo finalizado em nosso repositório do GitHub (<http://github.com/semmypurewal/LearningWebAppDev/tree/master/Chapter4/Amazeriffic>).

Resumo

Neste capítulo, aprendemos a adicionar interatividade a uma aplicação web por meio do uso da jQuery e do JavaScript. A jQuery é uma biblioteca amplamente utilizada, que abstrai algumas das dificuldades com a manipulação do DOM e o tratamento de eventos (entre outras coisas). A natureza orientada a eventos de uma interface de usuário sendo executada em um navegador cria a necessidade de termos um código *assíncrono*. A maneira mais fácil de começar uma implementação desse tipo é por meio da associação de funções de *callback* aos eventos.

O JavaScript é uma linguagem de programação rica em recursos, suportada por todos os navegadores web. Como desenvolvedor e programador iniciante de aplicações web, é importante trabalhar no sentido de dominar alguns conceitos fundamentais, incluindo variáveis, instruções `if` e `if-else` e diversas estruturas de laço. Os arrays constituem outro componente essencial de todas as linguagens de programação, portanto entender como criá-los e manipulá-los também é importante.

O JSLint é uma ferramenta semelhante ao CSS Lint ou à ferramenta de validação de HTML – ele evita que você caia em armadilhas

comuns do JavaScript. Submeter periodicamente o seu código a essa ferramenta é uma boa ideia.

Práticas e leituras adicionais

Memorização

Vamos acrescentar alguns passos ao seu exercício de memorização:

1. Crie um diretório chamado *javascripts* para armazenar os seus arquivos *.js*.
2. Crie um programa JavaScript simples e armazene-o em um arquivo chamado *app.js*. O arquivo deve gerar o alerta “hello world” ou exibi-lo no console. Isso deve acontecer em uma função *main*, que deve ser chamada usando a função *document.ready* da jQuery.
3. Importe o seu script, juntamente com a jQuery, em seu documento HTML na parte final da tag `<body>`.
4. Recarregue a página no Chrome para confirmar se o seu comportamento está correto.
5. Adicione o arquivo e as alterações no HTML ao seu repositório Git.

Plug-ins da jQuery

Parte do motivo pelo qual a jQuery é tão popular é porque ela tem uma comunidade enorme de desenvolvedores web que criam plug-ins, os quais permitem incluir efeitos incríveis em sua página. Sugiro que você acesse <http://plugins.jquery.com> para ver os tipos de plug-ins que estão disponíveis. Também sugiro que você experimente usar alguns plug-ins jQuery. Será necessário ler a documentação dos plug-ins para ver se você conseguirá fazê-los funcionar, porém vale a pena investir o seu tempo nisso.

Um de meus plug-ins favoritos chama-se *colorbox*, que permite

incluir facilmente uma galeria de fotos animada em sua página. O autor criou uma documentação bastante clara (<http://www.jacklmoore.com/colorbox/>) para você fazer o plug-in funcionar.

Seletores jQuery

Digite o documento HTML a seguir e salve-o em um arquivo chamado *selectorpractice.html*:

```
<!doctype html>
<html>
  <head>
  </head>
  <body>
    <h1>Hi</h1>
    <h2 class="important">Hi again</h2>
    <p>Random unattached paragraph</p>
    <div class="relevant">
      <p class="a">first</p>
      <p class="a">second</p>
      <p>third</p>
      <p>fourth</p>
      <p class="a">fifth</p>
      <p class="a">sixth</p>
      <p>seventh</p>
    </div>
  </body>
</html>
```

Em seguida, crie um arquivo *app.js* simples com o conteúdo a seguir. Utilize duas tags `<script>` para importar a jQuery e depois esse arquivo no final do elemento `body`:

```
var main = function () {
  "use strict;"

  $("").css("color", "red");
};
$(document).ready(main);
```

A função `css` nos permite alterar o estilo dos elementos selecionados

usando a jQuery. O código default utiliza o seletor CSS universal para tornar todos os elementos do DOM vermelhos. Nos exercícios a seguir, iremos alterar o seletor jQuery para deixar somente os elementos especificados em vermelho. Por exemplo, se quisermos deixar a tag `<h1>` vermelha, usaremos este seletor:

```
$("#h1").css("color","red");
```

Os últimos passos serão muito mais fáceis se você der uma olhada na documentação sobre seletores da jQuery (<http://api.jquery.com/category/selectors/>). Juntamente com os seletores que vimos neste capítulo, preste atenção, em especial, nos seletores `:not` e `:gt`:

1. Selecione o elemento `<h2>` pela classe.
2. Selecione o primeiro parágrafo entre os parágrafos relevantes.
3. Selecione o terceiro parágrafo entre os parágrafos relevantes.
4. Selecione todos os parágrafos da página.
5. Selecione todos os parágrafos relevantes.
6. Selecione o segundo, o quarto e o sexto parágrafos relevantes.
7. Selecione o sétimo parágrafo relevante.
8. Selecione o quinto, o sexto e o sétimo parágrafos relevantes.
9. Selecione os parágrafos relevantes que não sejam da classe `a`.

FizzBuzz

Se estiver pensando em se candidatar a um emprego de programador de computadores, você realmente deve saber como solucionar o problema FizzBuzz. Até onde eu sei, o problema FizzBuzz se tornou famoso por causa de uma postagem de blog feita por Jeff Atwood, intitulada *Why Can't Programmers... Program?* (Por que os programadores não conseguem... programar?, em <http://bit.ly/1fYbhZd>). Nessa postagem, ele lamenta o fato de que tantas pessoas que se candidatam a empregos como programadores não conseguem resolver um problema simples.

O problema essencialmente é este:

Crie um programa que mostre os números de 1 a 100. No entanto, para os múltiplos de três, mostre “Fizz” no lugar do número e para os múltiplos de cinco, mostre “Buzz”. Para os números que são múltiplos tanto de três quanto de cinco, mostre FizzBuzz.

Com base nos conhecimentos anteriores e nas ferramentas apresentadas neste capítulo, você deverá ser capaz de resolver esse problema utilizando um laço `for`, juntamente com uma série de instruções `if-else` e o operador de resto de divisão.

Devo acrescentar que não estou totalmente certo de que esse seja, particularmente, um bom teste para saber se uma pessoa pode ou não programar, nem se a incapacidade de solucionar esse problema naquele instante deva desqualificar alguém para um cargo de programador. Há inúmeros motivos pelos quais uma pessoa pode não ser capaz de resolver esse problema e que podem não ter nenhuma relação com a sua capacidade de criar programas de computador em um nível iniciante. No entanto é assim que as coisas são, e é provável que essa pergunta seja feita a você em algum momento. Memorizar a solução é uma boa ideia.

Exercícios com arrays

Outro tipo comum de perguntas em entrevistas para emprego diz respeito aos arrays. Há um bom motivo para isso: definitivamente, os arrays são a estrutura de dados mais comumente utilizada em programas de computador e toda linguagem de programação os disponibiliza de alguma forma. Procure solucionar todos os problemas desta seção utilizando o material deste capítulo. Começarei com um problema simples e incluirei algumas soluções diferentes:

Crie uma função que aceite um array de números como argumento e retorne a soma desses números.

A solução mais simples para esse problema será algo deste tipo:


```

var sum = function (nums) {
  var sumSoFar = 0,
  i;

  // percorre o array com um laço, fazendo a soma
  for (i = 0; i < nums.length; i++) {
    sumSoFar = sumSoFar + nums[i];
  }

  // agora que acabamos de percorrer o array,
  // a variável sumSoFar deve conter a soma
  // de todos os números
  return sumSoFar;
};

sum([1,2,3,4]);
//=> 10

```

De modo semelhante, podemos usar um laço `forEach`:

```

var sum = function (nums) {
  var sumSoFar = 0;

  // use um laço forEach
  nums.forEach(function (value) {
    sumSoFar = sumSoFar + value;
  });

  return sumSoFar;
};

sum([1,2,3,4]);
//=> 10

```

O laço `forEach` é preferível nesse caso porque ele elimina a necessidade de manter a variável `i`. Em geral, isso é um aspecto positivo – eliminar uma variável que tenha o seu estado alterado regularmente em nosso programa torna o nosso código menos suscetível a erros. Com efeito, se quisermos nos livrar de todas as variáveis locais temporárias, podemos utilizar uma função de array chamada `reduce` que sintetiza tudo isso de maneira elegante:

```

var sum = function (nums) {
  return nums.reduce(function (sumSoFar, value) {
    return sumSoFar + value;
  }, 0);
}

```

```
};  
sum([1,2,3,4]);  
//=> 10
```

Não vou gastar mais tempo falando sobre a função `reduce`, mas, se ficar intrigado, sugiro que você leia mais a respeito dela na web.

Vale a pena observar também que não estamos fazendo a verificação das entradas de nossa função. Por exemplo, o que aconteceria se tentássemos enviar algo que não fosse um array?

```
sum(5);  
//=> TypeError!  
  
sum("hello world");  
//=> TypeError!
```

Há inúmeras maneiras de corrigir esse problema, porém não vamos gastar muito tempo nos preocupando com isso agora. Entretanto, se você estiver criando um código em uma entrevista de emprego, é sempre uma boa ideia fazer uma verificação das entradas em qualquer função.

Aqui está uma série de questões que permitirão fazer exercícios com arrays:

1. Crie uma função que aceite um array de números como argumento e que retorne a média desses números.
2. Crie uma função que aceite um array de números como argumento e que retorne o maior número do array.
3. Crie uma função que aceite um array de números e que retorne `true` se ele contiver pelo menos um número par e `false`, caso contrário.
4. Crie uma função que aceite um array de números e que retorne `true` se *todos* os números forem pares e `false`, caso contrário.
5. Crie uma função que aceite dois argumentos – um array de strings e uma string – e retorne `true` se a string estiver contida no array e `false`, caso contrário. Por exemplo, sua função deve se comportar da seguinte maneira:

```
arrayContains(["hello", "world"], "hello");  
//=> true  
  
arrayContains(["hello", "world"], "goodbye");  
//=> false  
  
arrayContains(["hello", "world", "goodbye"], "goodbye");  
//=> true
```

- 6.** Crie uma função que seja semelhante à função anterior, porém que retorne `true` somente se o array contiver a string especificada pelo menos duas vezes:

```
arrayContainsTwo(["a", "b", "a", "c"], "a");  
//=> true  
  
arrayContainsTwo(["a", "b", "a", "c"], "b");  
//=> false  
  
arrayContainsTwo(["a", "b", "a", "c", "a"], "a");  
//=> true
```

Depois que esse código estiver funcionando, crie uma função chamada `arrayContainsThree` que se comporte de modo semelhante, porém para três ocorrências no lugar de duas. Agora iremos generalizar o problema anterior. Crie uma função que aceite três argumentos e que retorne `true` se o array contiver o elemento `n` vezes, em que `n` corresponde ao terceiro argumento:

```
arrayContainsNTimes(["a", "b", "a", "c", "a"], "a", 3);  
//=> true  
  
arrayContainsNTimes(["a", "b", "a", "c", "a"], "a", 2);  
//=> true  
  
arrayContainsNTimes(["a", "b", "a", "c", "a"], "a", 4);  
//=> false  
  
arrayContainsNTimes(["a", "b", "a", "c", "a"], "b", 2);  
//=> false  
  
arrayContainsNTimes(["a", "b", "a", "c", "a"], "b", 1);  
//=> true  
  
arrayContainsNTimes(["a", "b", "a", "c", "a"], "d", 0);  
//=> true
```

Projeto Euler

Outra ótima fonte de problemas a serem usados como exercícios é o Projeto Euler (<http://projecteuler.net/>). Ele contém uma série de problemas de programação que não exigem conhecimentos avançados para que sejam corretamente solucionados. A maioria dos problemas se reduz a descobrir um único número. Após ter descoberto esse número, digite-o no site e – se estiver correto – você terá acesso a um painel de discussão em que outras pessoas postam soluções e discutem a melhor maneira de resolver o problema.

Certa vez, uma pergunta que havia sido retirada, praticamente palavra por palavra, de um problema do Projeto Euler me foi apresentada quando eu estava fazendo uma entrevista em uma famosa empresa de software.

Outras referências ao JavaScript

O JavaScript é uma linguagem popular, e por esse motivo há diversos livros que o discutem. Infelizmente, Doug Crockford argumenta que a maioria desses livros “é terrível. Eles contêm erros, exemplos pobres e promovem práticas ruins.”¹ Tenho a tendência de concordar com ele (embora espero que ele não coloque o meu livro nessa categoria!).

Considerando o que foi dito, eu ainda não encontrei um bom livro que discuta a programação JavaScript para programadores iniciantes ou para pessoas que sabem muito pouco de programação em geral. Por outro lado, há vários livros excelentes para programadores JavaScript de nível intermediário e avançado. Posso recomendar fortemente o livro de Crockford, *O melhor do JavaScript* (Alta Books, 2008), como um dos melhores livros sobre a linguagem JavaScript em geral. Também acho que o livro *Effective JavaScript* de David Herman (Addison-Wesley, 2012) contém uma boa quantidade de diretrizes práticas para programadores JavaScript de nível intermediário.

Se você estiver à procura de informações práticas em geral,

relacionadas com o desenvolvimento de software em JavaScript, sugiro que você dê uma olhada no livro *Maintainable JavaScript* de Nick Zacks (O'Reilly, 2012). E depois que você dominar a programação orientada a objetos em JavaScript, dê uma olhada no livro *Functional JavaScript* de Michael Fogus (O'Reilly, 2013), que oferece um ponto de vista diferente, porém totalmente envolvente.

¹ N.T.: No original: “are quite awful. They contain errors, poor examples, and promote bad practices.”

CAPÍTULO 5

A ponte entre o cliente e o servidor

Estamos praticamente no final de nossa jornada pela parte referente ao lado do cliente de uma aplicação web. Embora eu não tenha afirmado explicitamente, quando me refiro ao *lado do cliente* de uma aplicação web, estou falando da parte do programa que é executada em seu navegador web. A outra metade é a parte referente ao *lado do servidor* de uma aplicação, que é executada e armazena informações fora de seu navegador web, normalmente em um computador remoto.

Este capítulo não discute a parte do lado do servidor de uma aplicação web, mas sim um conjunto de tecnologias que permite que o cliente e o servidor troquem informações mais facilmente. Gosto de pensar nesse conjunto de tecnologias como a *ponte* entre o cliente e o servidor.

Estudaremos especificamente os objetos JavaScript, o JSON (JavaScript Object Notation, ou Notação de Objetos JavaScript) e o AJAX (Asynchronous JavaScript And XML, ou JavaScript Assíncrono e XML – uma espécie de nomenclatura equivocada). Esses assuntos nos prepararão para o Node.js, que será estudado no próximo capítulo.

Hello, objetos JavaScript!

Antes de começarmos a falar sobre a transferência de dados entre computadores, precisamos discutir outra primitiva importante do

JavaScript: os objetos. Você já deve ter ouvido falar de *programação orientada a objetos* antes e, se você já programou em C++ ou em Java, é provável que tenha visto o assunto em detalhes.

Embora essas ideias sejam muito importantes para a engenharia de software em geral, a programação orientada a objetos em JavaScript é uma criatura totalmente diferente, portanto é melhor esquecê-la ao começar a aprender JavaScript. Por enquanto, vamos assumir uma visão um pouco ingênua sobre os objetos: eles correspondem a coleções simples de variáveis relacionadas a uma entidade em particular. Vamos começar com um exemplo.

Representando um jogo de baralho

No capítulo anterior, vimos alguns exemplos que envolviam cartas de baralho. Uma carta de baralho tem dois atributos básicos: um naipe (paus, ouro, copas ou espada) e um valor (de dois a dez ou uma figura que pode ser um valete, uma dama, um rei ou um ás). Agora suponha que queremos desenvolver um pouco mais esse exemplo e criar uma aplicação web que jogue pôquer. Isso exigirá fazer a representação da mão de cinco cartas em JavaScript.

A abordagem mais básica que utiliza as ferramentas apresentadas envolve a manutenção de dez variáveis, uma para cada valor do baralho e uma para cada naipe:

```
var cardOneSuit = "hearts",
    cardOneRank = "two",
    cardTwoSuit = "spades",
    cardTwoRank = "ace",
    cardThreeSuit = "spades",
    cardThreeRank = "five",
    // ...
    cardFiveSuitRank = "seven";
```

Espero que você tenha percebido que essa é uma solução maçante. E se você prestou bastante atenção no capítulo anterior, espero que você esteja pensando que um array de cinco elementos pode simplificar a situação! Porém o problema é que cada carta possui

dois atributos, então como podemos fazer isso com um array?

Uma solução deste tipo pode ser considerada:

```
var cardHandSuits = ["hearts", "spades", "spades", "clubs", "diamonds"],  
    cardHandRanks = ["two", "ace", "five", "king", "seven"];
```

Definitivamente, esse código é melhor, porém ele ainda apresenta um grande problema em comum com a solução anterior: não há nada que associe um único naipe a um único valor em nosso programa – é necessário manter o controle dessa associação em nossas mentes. Sempre que houver entidades em nosso programa que estejam fortemente relacionadas, mas somente na mente do programador, em vez de estar em alguma estrutura de programação, adicionamos muita complexidade. E o fato é que isso é totalmente desnecessário se usarmos objetos!

Um *objeto* é simplesmente uma coleção de diversas variáveis relacionadas de alguma maneira. Para criar um, utilizamos chaves; podemos então acessar as variáveis internas de um objeto usando o operador ponto (.). Aqui está um exemplo que cria uma única carta:

```
// cria um objeto carta com o valor igual a 'two' (dois)  
// e um naipe igual a 'hearts' (copas)  
var cardOne = { "rank": "two", "suit": "hearts" };  
// exibe o valor de cardOne  
console.log(cardOne.rank);  
//=> two  
//exibe o naipe de cardOne  
console.log(cardOne.suit);  
//=> hearts
```

Após ter criado o objeto, sempre podemos alterá-lo posteriormente. Por exemplo, podemos mudar rank e suit de cardOne:

```
// muda a carta para um ás de espadas  
cardOne.rank = "ace";  
cardOne.suit = "spades";  
  
console.log(cardOne);  
//=> Object {rank: "ace", suit: "spades"}
```

Assim como ocorre com um array, podemos criar um objeto vazio e

adicionar atributos a ele posteriormente:

```
// cria um objeto vazio
var card = {};

// define o valor para ace (ás)
card.rank = "ace";

console.log(card);
//=> Object {rank: "ace"}

// define o naipe para hearts (copas)
card.suit = "hearts";

console.log(card);
//=> Object {rank: "ace", suit: "hearts"}
```

Agora, se quisermos representar uma mão de cartas, podemos criar um array e preenchê-lo com objetos do tipo carta em vez de manter dois arrays separados!

```
// cria um array vazio
var cards = [];

// insere o dois de copas no array
cards.push( {"rank": "two", "suit": "hearts"} );
cards.push( {"rank": "ace", "suit": "spades"} );
cards.push( {"rank": "five", "suit": "spades"} );
cards.push( {"rank": "king", "suit": "clubs"} );
cards.push( {"rank": "seven", "suit": "diamonds"} );

// exibe a primeira e a terceira cartas da mão
console.log(cards[0]);
//=> Object {rank: "two", suit: "hearts"}

console.log(cards[2]);
//=> Object {rank: "five", suit: "spades"}
```

Se você preferir, podemos também criar um array literal longo para criar uma mão de cartas:

```
// cria um array de cartas
// usando um array literal longo
var cards = [
  {"rank": "two", "suit": "hearts"},
  {"rank": "ace", "suit": "spades"},
  {"rank": "five", "suit": "spades"},
  {"rank": "king", "suit": "clubs"},
  {"rank": "seven", "suit": "diamonds"}
];
```

```
    {"rank": "seven", "suit": "diamonds"}  
  ];
```

Generalizações

Conforme mencionado anteriormente, podemos pensar nos objetos JavaScript como coleções de variáveis, cada qual com um nome e um valor. Para criar um objeto vazio, basta usar as chaves de abertura e de fechamento:

```
// cria um objeto vazio  
var s = {};
```

Em seguida, podemos acrescentar variáveis ao objeto utilizando o operador ponto (.):

```
s.name = "Semmy";
```

As variáveis dentro de um objeto podem ser de qualquer tipo, incluindo strings (que foram vistas em todos os exemplos anteriores), arrays ou até mesmo outros objetos!

```
s.age = 36; // um número  
s.friends = [ "Mark", "Emily", "Bruce", "Sylvan" ];  
s.dog = { "name": "Gracie", "breed": "Shepherd Mix" };  
console.log(s.age);  
//=> 36  
console.log(s.friends[1]);  
//=> "Emily"  
console.log(s.dog);  
//=> Object {name: "Gracie", breed: "Shepherd Mix"}  
console.log(s.dog.name);  
//=> "Gracie"
```

Também podemos criar objetos literais, que são simplesmente objetos completos definidos no código!

```
var g = {  
  "name": "Gordon",  
  "age": 36,  
  "friends": [ "Sara", "Andy", "Roger", "Brandon" ],  
  "dog": { "name": "Pi", "breed": "Lab Mix" }  
};
```

```
}  
console.log(g.name);  
//=> "Gordon"  
  
console.log(g.friends[2]);  
//=> "Roger"  
  
console.log(g.dog.breed);  
//=> "Lab Mix"
```

Ocasionalmente, será necessário utilizar o valor especial `null`, que representa “nenhum objeto”:

```
var b = {  
  "name": "John",  
  "age" : 45,  
  "friends" : [ "Sara", "Jim" ],  
  "dog" : null  
}
```

Nesse exemplo, John não tem um cachorro. Também podemos usar `null` para indicar que não precisamos mais de um objeto:

```
// atribui o objeto g a currentPerson  
var currentPerson = g;  
  
// ... realiza algumas operações com currentPerson  
  
// define currentPerson com null  
currentPerson = null;
```

Utilizaremos uma referência `null` como placeholder para um objeto, particularmente no capítulo 6, quando começaremos a falar sobre erros. Nossas funções de callback, especificamente, serão chamadas com `null` quando não houver nenhum erro associado à solicitação.

Falando de modo geral, os objetos nos proporcionam bastante flexibilidade no que diz respeito ao armazenamento e à manipulação de dados. Também vale a pena observar que, como as variáveis de função no JavaScript se comportam do mesmo modo que as demais variáveis, podemos armazená-las em objetos. Por exemplo, já vimos como acessar funções associadas a objetos jQuery:

```
// obtém um elemento do DOM  
var $headerTag = $("h1");
```

```
// $headerTag é um objeto que possui uma função  
// associada cujo nome é fadeOut  
$headerTag.fadeOut();
```

Objetos com funções associadas são ferramentas incríveis para efetuar abstrações (o que nos leva à ideia de programação orientada a objetos), porém iremos postergar essa discussão por enquanto. Neste capítulo, vamos focar no uso de objetos para trocar informações com outras aplicações web.

A comunicação entre computadores

É quase impossível criar uma aplicação web atualmente sem entrar em contato com outras aplicações previamente existentes. Como exemplo, suponha que você queira que sua aplicação permita aos usuários fazer login por meio de suas contas do Twitter. Ou você queira fazer com que sua aplicação publique atualizações no feed do Facebook de um usuário. Isso significa que a sua aplicação deverá ser capaz de trocar informações com esses serviços. O formato-padrão usado na Web hoje em dia é o JSON, e se você entende os objetos JavaScript, já compreenderá também o JSON!

JSON

Um objeto JSON nada mais é do que um objeto JavaScript literal na forma de uma string (com algumas ressalvas de caráter técnico, mas podemos ignorar a maior parte delas por enquanto). Isso significa que, quando quisermos enviar algumas informações a outro serviço, basta criar um objeto JavaScript em nosso código, convertê-lo em uma string e enviá-lo! Na maioria das vezes, as bibliotecas AJAX cuidarão de boa parte desse processo para nós, portanto, para um programador, pode parecer que os programas estão simplesmente trocando objetos!

Por exemplo, suponha que eu queira definir um objeto JSON em um arquivo externo. Posso simplesmente codificá-lo exatamente como eu faria em um programa JavaScript:

```
{  
  "rank":"ten",  
  "suit":"hearts"  
}
```

O fato é que converter uma string JSON em um objeto que possa ser utilizado por um programa de computador é muito fácil na maioria das linguagens de programação! Em JavaScript, porém, é superfácil porque o formato em si é somente uma versão em string de um objeto literal. A maioria dos ambientes JavaScript oferece um objeto JSON com o qual podemos interagir. Por exemplo, abra o console do Chrome e digite o código a seguir:

```
// criamos uma string JSON usando aspas simples  
var jsonString = '{"rank":"ten", "suit":"hearts"}'  
  
// JSON.parse a converte em um objeto  
var card = JSON.parse(jsonString);  
  
console.log(card.rank);  
//=> "ten"  
  
console.log(card.suit);  
//=> "hearts"
```



Observe que criamos a string jsonString usando aspas simples no lugar de nossas aspas duplas habituais. O fato é que o JavaScript não se importa com o tipo de aspa usado, mas o JSON sim. Como precisamos criar uma string dentro de uma string, usamos aspas simples na parte externa e aspas duplas na parte interna.

Também podemos converter um objeto JSON em uma string por meio da função `stringify`:

```
console.log(JSON.stringify(card));  
//=> {"rank":"ten","suit":"hearts"}
```

Agora que sabemos como criar objetos JSON em arquivos externos e como strings em nosso programa, vamos ver de que modo esses objetos são trocados entre os computadores.

AJAX

AJAX significa *Asynchronous JavaScript And XML* (JavaScript Assíncrono e XML), o que, como mencionei na introdução, é uma

espécie de nomenclatura equivocada. O formato comum de troca de dados anterior ao JSON chamava-se *XML*, que era muito semelhante ao HTML. E embora o XML ainda seja amplamente utilizado em diversas aplicações, houve uma mudança significativa para o JSON desde que o AJAX foi criado.

Apesar de ser um acrônimo cheio de termos técnicos, na realidade, o AJAX não é muito complicado. A ideia básica por trás dele é que a sua aplicação pode enviar e receber informações de outros computadores sem a necessidade de recarregar a página web. Um dos primeiros exemplos desse caso (e continua sendo um dos melhores) é a aplicação web Gmail do Google, que entrou em cena há cerca de dez anos! Se você já a usou, deve ter percebido como os novos emails simplesmente aparecem como em um *passé de mágica* em sua caixa de entrada; em outras palavras, não é necessário recarregar a página para obter as novas mensagens. Esse é um exemplo de AJAX.

Acessando um arquivo JSON externo

Tudo bem, chega de teoria. Vamos ver um exemplo do AJAX em ação. Inicialmente, vamos criar o esqueleto de uma aplicação. Vamos ignorar o CSS por enquanto, somente para ter o exemplo pronto para executar. Espero que, a essa altura, você possa criar uma página HTML a partir da memória que contenha “Hello World!” em um elemento `h1` e um elemento `main` vazio. Também queremos incluir um elemento `script` com um link para a jQuery a partir de uma CDN (explicado no capítulo 4) e uma tag `<script>` contendo um link para *app.js*, que é a nossa aplicação JavaScript básica. Esse arquivo deve estar em um diretório chamado *javascripts* e terá um aspecto semelhante a:

```
var main = function () {  
    "use strict";  
  
    console.log("Hello World!");  
}  
$(document).ready(main);
```

Abra a sua aplicação básica no Chrome e abra o JavaScript console. Como sempre, se tudo estiver funcionando corretamente, você deverá ver “Hello World!” sendo apresentado.

A seguir, criaremos um novo diretório chamado *cards* dentro de nosso diretório de exemplo. Nesse diretório, criaremos um arquivo chamado *aceOfSpades.json*, cujo conteúdo se parecerá exatamente com uma das definições anteriores de objetos JavaScript:

```
{  
  "rank" : "ace",  
  "suit" : "spades"  
}
```

Agora vamos acessar esse arquivo em nosso programa por meio do AJAX, o que, por ora, apresentará um pequeno problema.

Passando por cima de restrições de segurança dos navegadores

Quando foi criado, o JavaScript foi concebido para ser executado em um navegador web. Isso significa que, por causa de restrições de segurança, ele não tem permissão para acessar arquivos locais armazenados em seu computador. Você pode imaginar os tipos de problema que podem surgir se essa operação fosse permitida – qualquer site que você visitasse teria acesso total ao seu computador!

Nem é preciso dizer que essa é uma preocupação com a segurança que continua presente até os dias de hoje. Por outro lado, temos permissão para acessar determinados tipos de arquivo do mesmo servidor que enviou o arquivo JavaScript. Os arquivos JSON são um exemplo perfeito desse caso: posso fazer uma solicitação AJAX a um servidor e acessar qualquer arquivo JSON disponibilizado por ele. Infelizmente, não utilizaremos um servidor até o próximo capítulo, portanto, por enquanto, utilizaremos uma solução alternativa mais simples usando o Chrome.

Provavelmente, não é uma boa ideia surfar pela Internet com essas medidas de



segurança desabilitadas. Não se esqueça de reiniciar o seu navegador normalmente antes de acessar sites diferentes daqueles que foram criados por você mesmo.

Para iniciar o Chrome sem essa restrição de segurança em particular habilitada, começaremos fechando-o completamente. A seguir, se você estiver no Mac OS, abra um terminal e digite o comando a seguir:

```
hostname $ open -a Google\ Chrome --args --allow-file-access-from-files
```

Se estiver no Windows, podemos fazer o mesmo ao clicar no menu Start (Iniciar), digitar `run` na barra de pesquisa e então inserir o comando a seguir na caixa de execução:

```
%userprofile%\AppData\Local\Google\Chrome\Application\chrome.exe --allow-file-access-from-files
```

Isso deve fazer com que o Chrome seja aberto com a restrição de segurança citada anteriormente desabilitada. Como eu disse, usaremos esse truque somente no caso desse exemplo, neste capítulo – no próximo capítulo, executaremos um servidor de verdade e essa configuração não será mais necessária. Por enquanto, porém, vamos verificar se isso está funcionando.

Função `getJSON`

Se desabilitarmos com sucesso a restrição de segurança referente ao cross-site scripting do Chrome, podemos facilmente ter acesso ao arquivo JSON local em nosso programa. Para isso, utilizaremos a função `getJSON` da jQuery. Como em muitos de nossos exemplos com JavaScript, essa solicitação jQuery será assíncrona, portanto será necessário adicionar uma callback:

```
var main = function () {  
    "use strict";  
  
    // getJSON efetua até mesmo o parse do JSON para nós, portanto não é  
    // necessário chamar JSON.parse  
    $.getJSON("cards/aceOfSpades.json", function (card) {  
        // exibe a carta no console  
        console.log(card);  
    });  
};
```



```
};  
$(document).ready(main);
```

Se fizermos tudo corretamente, ao abrirmos nossa página no Chrome (com as restrições de cross-site scripting desabilitadas), deveremos ver a carta aparecer na janela do console. Agora podemos usá-la em nosso programa, assim como faríamos com qualquer outro objeto JavaScript!

```
var main = function () {  
    "use strict";  
  
    console.log("Hello World!");  
  
    $.getJSON("cards/aceOfSpades.json", function (card) {  
        // cria um elemento para armazenar a carta  
        var $cardParagraph = $("<p>");  
  
        // adiciona texto ao elemento parágrafo  
        $cardParagraph.text(card.rank + " of " + card.suit);  
  
        // adiciona o parágrafo da carta ao main  
        $("main").append($cardParagraph);  
    });  
}  
$(document).ready(main);
```

Um array JSON

Podemos até mesmo ter objetos JSON mais complexos no arquivo. Por exemplo, ele pode ser constituído de um array de objetos no lugar de um único objeto. Para ver isso em ação, crie o arquivo chamado *hand.json* a seguir:

```
[  
    { "suit" : "spades", "rank" : "ace" },  
    { "suit" : "hearts", "rank" : "ten" },  
    { "suit" : "spades", "rank" : "five" },  
    { "suit" : "clubs", "rank" : "three" },  
    { "suit" : "diamonds", "rank" : "three" }  
]
```

Agora adicione este `getJSON` logo abaixo do anterior:

```
var main = function () {
```

```

"use strict";
console.log("Hello World!");
$.getJSON("cards/aceOfSpades.json", function (card) {
    // cria um elemento para armazenar a carta
    var $cardParagraph = $("<p>");

    // cria o texto da carta
    $cardParagraph.text(card.rank + " of " + card.suit);

    // adiciona o parágrafo da carta ao main
    $("main").append($cardParagraph);
});
$.getJSON("cards/hand.json", function (hand) {
    var $list = $("<ul>");

    // hand é um array, portanto é possível iterar pelos seus elementos
    // usando um laço forEach
    hand.forEach(function (card) {
        // cria um item de lista para armazenar a carta
        // e insere na lista
        var $card = $("<li>");
        $card.text(card.rank + " of " + card.suit);
        $list.append($card);
    });

    // adiciona a lista ao main
    $("main").append($list);
});
$(document).ready(main);

```



Nesse exemplo, reutilizamos a variável `card` no laço `forEach` na segunda chamada a `getJSON`. Isso ocorre porque a variável **é removida** do escopo no final da primeira chamada a `getJSON`. Não iremos discutir as regras de escopo do JavaScript em detalhes aqui, porém acho que vale a pena dar uma ênfase nos casos em que elas possam causar confusão.

Ao executar a nossa aplicação, a página deverá ter um aspecto semelhante ao da figura 5.1.

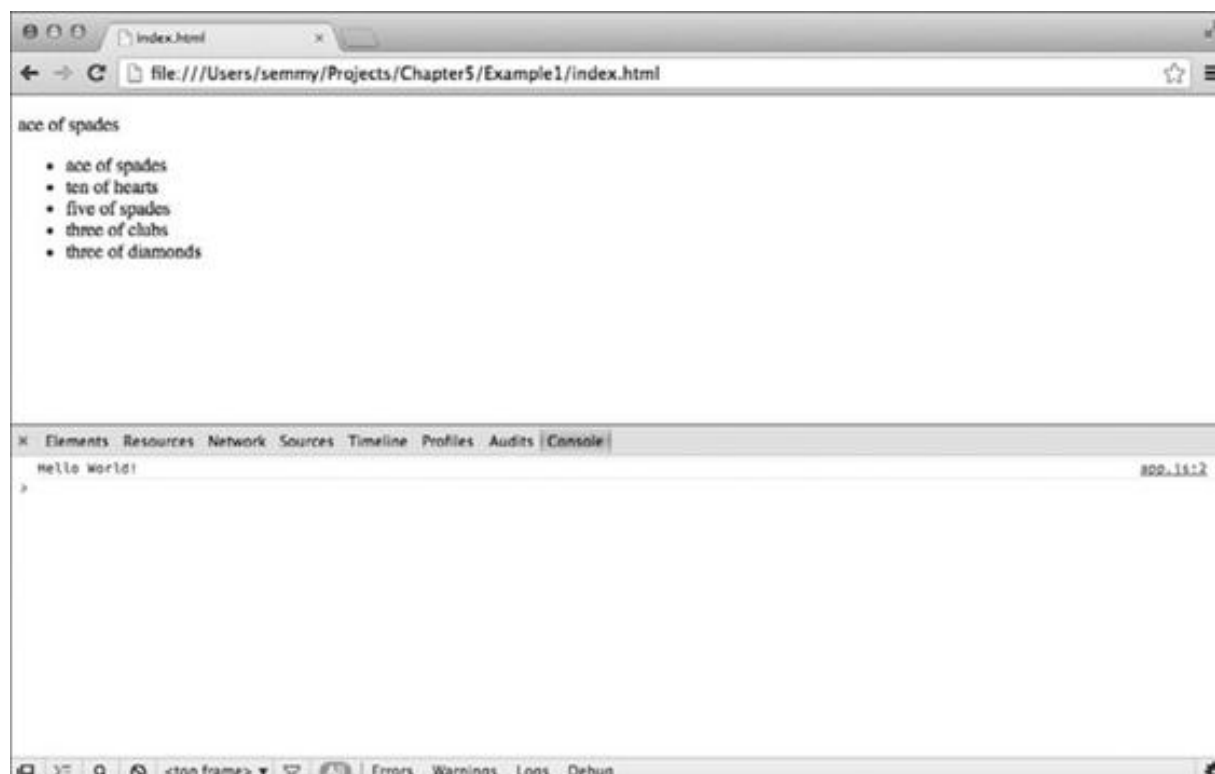


Figura 5.1 – Nossa primeira aplicação AJAX!

E daí?

Estou ciente de que esse exemplo não é extremamente empolgante. Com efeito, é provável que você esteja pensando que seria simplesmente mais fácil definir esses objetos em nosso código. Por que se importar em criar o arquivo separado?

Bem, suponha que os dados fossem provenientes de outro computador. Nesse caso, não poderíamos defini-los explicitamente em nosso código. Pode ser que não fôssemos sequer capazes de saber, com antecedência, como é o objeto! Por exemplo, suponha que queremos obter as fotos mais recentes de cães, publicadas no Flickr.

Obtendo imagens do Flickr

Vamos criar outro esqueleto de aplicação, desta vez com um arquivo CSS. Abra o Sublime e acesse o diretório *Chapter5* que está

em seu diretório *Projects*. Crie um diretório chamado *Flickr*. Nesse diretório, crie os seus diretórios *javascripts* e *stylesheets* usuais, juntamente com os arquivos *app.js* e *style.css*. Faça isso de modo que *app.js* exiba `hello world` no JavaScript console. Crie um arquivo *index.html*, que efetua o link de todos os arquivos para criar uma aplicação web básica.

Uau! Se fizer tudo isso de memória, você terá percorrido um longo caminho! Se tiver de efetuar algumas consultas ocasionalmente, tudo bem, mas memorizar e exercitar o processo básico de criação é uma boa ideia. Quanto mais tarefas você conseguir realizar de memória, melhor será para que sua mente possa focar em alguns pontos mais difíceis.

O meu arquivo HTML é semelhante a:

```
<!doctype html>
<html>
  <head>
    <title>Flickr App</title>
    <link rel="stylesheet" href="stylesheets/style.css">
  </head>
  <body>
    <header>
    </header>
    <main>
      <div class="photos">
      </div>
    </main>
    <footer>
    </footer>
    <script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>
    <script src="javascripts/app.js"></script>
  </body>
</html>
```

Agora tentaremos generalizar um exemplo retirado diretamente da documentação da jQuery, disponível em <http://api.jquery.com>. Vamos alterá-lo um pouco para que se torne mais adequado à nossa narrativa. Nesse caso, usaremos o JavaScript para extrair

imagens do Flickr, que é o serviço de compartilhamento de fotos do Yahoo. Antes de fazer isso, abra o seu navegador web e digite o URL a seguir:

*[http://api.flickr.com/services/feeds/photos_public.gne?
tags=dogs&format=json](http://api.flickr.com/services/feeds/photos_public.gne?tags=dogs&format=json)*

A resposta deve aparecer diretamente em seu navegador web na forma de JSON. Para mim, a resposta tem o seguinte aspecto:

```
jsonFlickrFeed({
  "title": "Recent Uploads tagged dogs",
  "link": "http://www.flickr.com/photos/tags/dogs/",
  "description": "",
  "modified": "2013-10-06T19:42:49Z",
  "generator": "http://www.flickr.com/",
  "items": [
    {
      "title": "Huck and Moxie ride stroller",
      "link": "http://www.flickr.com/photos/animaltourism/10124023233/",
      "media": {"m": "http://bit.ly/1bVvkn2"},
      "date_taken": "2013-09-20T09:14:25-08:00",
      "description": "...description string...",
      "published": "2013-10-06T19:45:14Z",
      "author": "nobody@flickr.com (animaltourism.com)",
      "author_id": "8659451@N03",
      "tags": "park dog beagle dogs brooklyn ride stroller prospect hounds"
    },
    {
      "title": "6th Oct Susie: \"You know that thing you're eating?\"",
      "link": "http://www.flickr.com/photos/cardedfolderol/10123495123/",
      "media": {"m": "http://bit.ly/1bVvbQw"},
      "date_taken": "2013-10-06T14:47:59-08:00",
      "description": "...description string...",
      "published": "2013-10-06T19:14:22Z",
      "author": "nobody@flickr.com (Cardedfolderol)",
      "author_id": "79284220@N08",
      "tags": "pets dogs animal mammal canine"
    },
    {
      "title": "6th Oct Susie ready to leave",
      "link": "http://www.flickr.com/photos/cardedfolderol/10123488173/",
```

```

    "media": {"m": "http://bit.ly/1bVvpXJ"},
    "date_taken": "2013-10-06T14:49:59-08:00",
    "description": "...description string...",
    "published": "2013-10-06T19:14:23Z",
    "author": "nobody@flickr.com (Cardedfolderol)",
    "author_id": "79284220@N08",
    "tags": "pets dogs animal mammal canine"
  }
]
});

```

Você verá que a aparência é bem mais complicada quando comparada aos exemplos que vimos até então, mas a estrutura e o formato básicos permanecem iguais. O código começa com algumas informações básicas sobre a solicitação e, em seguida, apresenta uma propriedade chamada `items`, que é um array de imagens. Cada elemento do array contém outro objeto chamado `media`, que possui uma propriedade `m`, a qual inclui um link para a imagem. No segundo capítulo, aprendemos que podemos adicionar uma imagem ao nosso documento HTML utilizando a tag ``. Faremos isso agora.

Vamos executar essa tarefa passo a passo. Inicialmente, adicionaremos uma linha à nossa função `main`, que define o URL em uma variável, e iremos chamar a função `getJSON` da jQuery como fizemos anteriormente:

```

var main = function () {
  "use strict";

  // esta, na realidade, é somente uma string,
  // porém eu a dividi em duas linhas
  // para torná-la mais legível
  var url = "http://api.flickr.com/services/feeds/photos_public.gne?" +
    "tags=dogs&format=json&jsoncallback=?";

  $.getJSON(url, function (flickrResponse) {
    // iremos simplesmente exibir a resposta no console
    // por enquanto
    console.log(flickrResponse);
  });
};

```

```
$(document).ready(main);
```

Se tudo for definido corretamente, ao executarmos esse código, o objeto com que o Flickr responde será exibido no console e poderemos examiná-lo ao detalhar o seu conteúdo utilizando as setas suspensas. Isso nos ajudará a resolver os problemas caso eles surjam à medida que prosseguirmos.

A seguir, iremos modificar o código de modo que, em vez de exibir todo o objeto, iremos exibir apenas cada URL individualmente. Em outras palavras, iremos fazer uma iteração pelo objeto `items` utilizando um laço `forEach`:

```
$.getJSON(url, function (flickrResponse) {  
    flickrResponse.items.forEach(function (photo) {  
        console.log(photo.media.m);  
    });  
});
```

Esse código deve exibir uma sequência de URLs no console – será possível até mesmo clicar neles para ver a imagem! A seguir, iremos inseri-los no DOM. Para isso, usaremos a função `attr` da jQuery, que ainda não tivemos a oportunidade de utilizar. Usaremos essa função para definir o atributo `src` manualmente em nossa tag ``:

```
$.getJSON(url, function (flickrResponse) {  
    flickrResponse.items.forEach(function (photo) {  
        // cria um novo elemento jQuery para armazenar a imagem  
        var $img = $("<img>");  
  
        // define o atributo usando o url  
        // contido na resposta  
        $img.attr("src", photo.media.m);  
  
        // associa a tag img ao elemento main  
        // photos  
        $("main .photos").append($img);  
    });  
});
```

Agora, ao recarregarmos a página, iremos ver as imagens do Flickr! E, se alterarmos o URL original para uma tag diferente de `dog`, toda a página será modificada! Como sempre, podemos adicionar efeitos

jQuery facilmente para que tudo aconteça de maneira um pouco mais elegante:

```
$.getJSON(url, function (flickrResponse) {  
  flickrResponse.items.forEach(function (photo) {  
    // cria um novo elemento jQuery para armazenar a imagem, mas  
    // oculta-o para que possamos fazer o fade in (aparecimento gradual)  
    var $img = $("<img>").hide();  
  
    // define o atributo usando o url  
    // contido na resposta  
    $img.attr("src", photo.media.m);  
  
    // associa a tag img ao elemento main  
    // photos e faz o seu fade in  
    $("main .photos").append($img);  
    $img.fadeIn();  
  });  
});
```

Agora as imagens aparecerão gradualmente quando a página for recarregada. Nos exercícios práticos no final do capítulo, modificaremos esse código para que a página percorra ciclicamente as imagens, mostrando uma imagem de cada vez.

Adicionando um recurso de tags ao Amazeriffic

Agora que já sabemos usar objetos JavaScript e JSON, pode ser útil integrar parte do que aprendemos em nossa aplicação Amazeriffic. Nesse exemplo, adicionaremos tags a cada item da lista de tarefas a fazer. Podemos usar essas tags para ordenar a nossa lista de tarefas de maneira diferente, porém de modo a fazer sentido. Além disso, podemos inicializar nossa lista de tarefas a partir de um arquivo JSON em vez de usar o array fixo no código.

Para começar, podemos copiar todo o nosso diretório *Amazeriffic* que está no diretório *Chapter4* para o nosso diretório *Chapter5*. Isso nos dará um ponto de partida sólido, de modo que não será preciso reescrever todo aquele código.

A seguir, vamos adicionar um arquivo JSON que contém a nossa lista de tarefas. Podemos salvá-la em um arquivo chamado *todos.json* no diretório raiz de nosso projeto (o diretório raiz é aquele que contém o arquivo *index.html*):

```
[
  {
    "description": "Get groceries",
    "tags": [ "shopping", "chores" ]
  },
  {
    "description": "Make up some new Todos",
    "tags": [ "writing", "work" ]
  },
  {
    "description": "Prep for Monday's class",
    "tags": [ "work", "teaching" ]
  },
  {
    "description": "Answer emails",
    "tags": [ "work" ]
  },
  {
    "description": "Take Gracie to the park",
    "tags": [ "chores", "pets" ]
  },
  {
    "description": "Finish writing this book",
    "tags": [ "writing", "work" ]
  }
]
```

Você verá que esse arquivo JSON contém um array de itens da lista de tarefas e que cada item contém um array de strings que correspondem às tags. Nosso objetivo é fazer com que as tags funcionem como um método secundário de organização de nossa lista de tarefas.

Para incorporar esse recurso novo, será preciso adicionar um pouco de código jQuery que leia o nosso arquivo JSON. Entretanto, como nossa função `main` do capítulo anterior depende das tarefas, será

necessário modificá-la para que `getJSON` seja chamado antes de `main`. Para isso, acrescentaremos uma função anônima à nossa chamada de `document.ready`, que chamará `getJSON` e, em seguida, chamará `main` com o resultado:

```
var main = function (todoObjects) {  
  "use strict";  
  // agora main tem acesso à nossa lista de tarefas!  
};  
$(document).ready(function () {  
  $.getJSON("todos.json", function (todoObjects) {  
    // chama main com as tarefas como argumento  
    main(todoObjects);  
  });  
});
```

Um pequeno problema com o nosso código é que ele não irá funcionar porque alteramos a estrutura do nosso objeto que representa as tarefas. Anteriormente, esse objeto era constituído de um array de strings que continha a descrição das tarefas, mas agora é um array de objetos. Se quisermos que o nosso código funcione como anteriormente, podemos criar o nosso tipo antigo de array a partir do tipo novo usando a função `map`.

A função `map`

A função `map` recebe um array e cria um novo array a partir dele ao aplicar uma função a cada elemento. Inicialize o console do Chrome e experimente executar o código a seguir:

```
// criaremos um array de números  
var nums = [1, 2, 3, 4, 5];  
// agora aplicaremos a função map,  
// que criará um novo array  
var squares = nums.map(function (num) {  
  return num*num;  
});  
console.log(squares);  
//=> [1, 4, 9, 16, 25]
```

Nesse exemplo, a função que retorna `num*num` é aplicada a cada elemento para criar o novo array. Esse pode parecer um exemplo estranho, porém aqui está outro mais interessante:

```
// criaremos um array de nomes
var names = [ "emily", "mark", "bruce", "andrea", "pablo" ];

// agora criaremos um novo array de nomes
// em que a primeira letra será maiúscula
var capitalizedNames = names.map(function (name) {
  // obtém a primeira letra
  var firstLetter = name[0];
  // retorna a primeira letra transformada em maiúscula,
  // juntamente com a string que começa no índice 1
  return firstLetter.toUpperCase() + name.substring(1);
});

console.log(capitalizedNames);
//=> [ "Emily", "Mark", "Bruce", "Andrea", "Pablo" ]
```

Você pode notar que criamos um array com nomes iniciados com letra maiúscula, sem nem mesmo efetuarmos a iteração pelo array!

Agora que entendemos o modo como o `map` funciona, criar o nosso antigo array a partir do novo é uma tarefa quase trivial:

```
var main = function (todoObjects) {
  "use strict";

  var todos = todoObjects.map(function (todo) {
    // simplesmente retornaremos a descrição
    // desse todoObject
    return todo.description;
  });

  // agora todo o nosso código antigo deve funcionar exatamente como antes!
  // ...
};

$(document).ready(function () {
  $.getJSON("todos.json", function (todoObjects) {
    // chamaremos main com os todos como argumento
    main(todoObjects);
  });
});
```

Agora que temos todo o nosso código antigo funcionando

exatamente como antes, podemos criar uma aba Tags.

Acrescentando uma aba Tags

Começaremos adicionando a aba Tags à nossa UI. Pelo fato de termos criado o nosso código para que fosse (relativamente) flexível, isso não será muito difícil. Iniciamos simplesmente abrindo *index.html* e adicionando o código para uma aba chamada Tags entre as abas Oldest (Mais antigas) e Add (Adicionar):

```
<div class="tabs">
  <a href=""><span class="active">Newest</span></a>
  <a href=""><span>Oldest</span></a>
  <a href=""><span>Tags</span></a>
  <a href=""><span>Add</span></a>
</div>
```

Essa linha adicional acrescentará uma aba à nossa UI e (quase) tudo funcionará exatamente como esperado. O único problema é que criamos as nossas abas de acordo com as suas posições nessa lista. Desse modo, quando clicarmos na aba Tags, veremos a interface contendo o botão Add. Definitivamente, esse não é o comportamento esperado, mas, felizmente, apenas uma pequena modificação será necessária.

Tudo o que é preciso fazer é acrescentar um bloco `else-if` a mais no meio do código da aba e reorganizar os números. Quando fiz isso em meu código, a seção relevante acabou ficando desta maneira:

```
} else if ($element.parent().is(":nth-child(3)")) {
  // ESTE É O CÓDIGO PARA A ABA TAGS
  console.log("the tags tab was clicked!");
} else if ($element.parent().is(":nth-child(4)")) {
  $input = $("<input>"),
  $button = $("<button>").text("+");

  $button.on("click", function () {
    toDos.push($input.val());
    $input.val("");
  });
  $content = $("<div>").append($input).append($button);
```

}

Criando a UI

Agora que sabemos como criar a aba, vamos dar uma olhada rápida em nosso objetivo para essa aba na figura 5.2.

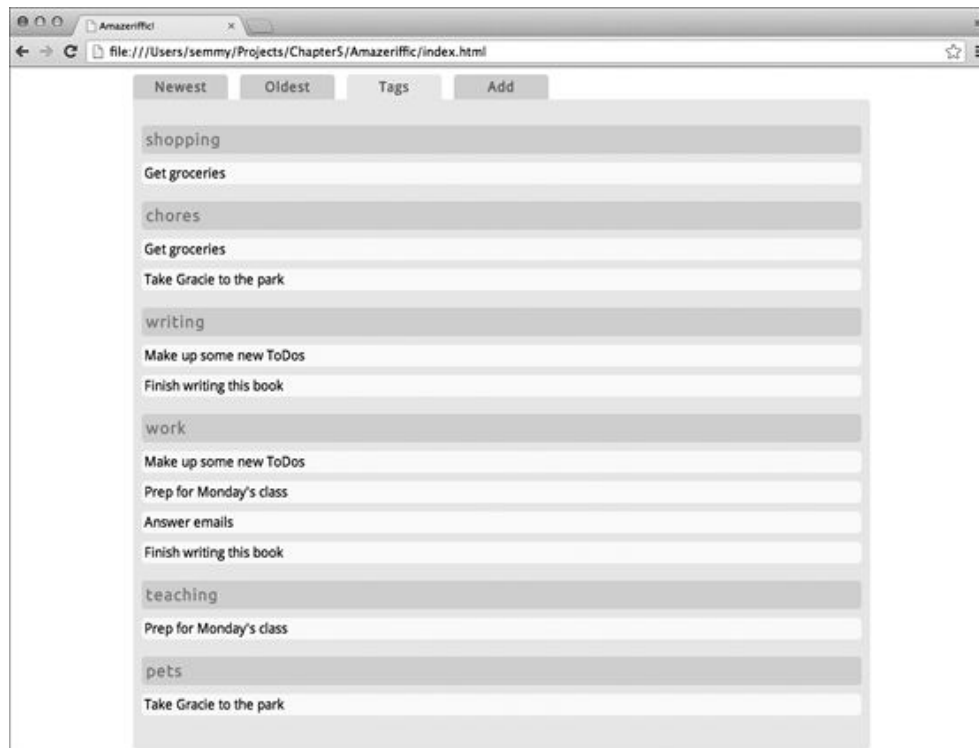


Figura 5.2 – Nosso objetivo para a aba Tags.

Nessa aba, pretendemos listar todas as tags como cabeçalho e, em seguida, adicionar as descrições das tarefas cujas tags sejam dessa categoria. Isso significa que as descrições de nossas tarefas podem aparecer em vários locais.

No entanto o problema nesse caso é que o nosso objeto JSON não está realmente armazenado de maneira a tornar essa tarefa muito fácil. Seria mais simples se nossos `toDoObjects` estivessem armazenados em um formato organizado de acordo com as tags:

```
[  
  {  
    "name": "shopping",  
    "toDos": ["Get groceries"]
```

```

    },
    {
      "name": "chores",
      "toDos": ["Get groceries", "Take Gracie to the park"]
    },
    {
      "name": "writing",
      "toDos": ["Make up some new ToDos", "Finish writing this book"]
    },
    {
      "name": "work",
      "toDos": ["Make up some new ToDos", "Prep for Monday's class",
        "Answer emails", "Finish writing this book"]
    },
    {
      "name": "teaching",
      "toDos": ["Prep for Monday's class"]
    },
    {
      "name": "pets",
      "toDos": ["Take Gracie to the park"]
    }
  ]

```

Felizmente, podemos modificar facilmente o nosso objeto `toDoObjects` original para que tenha esse formato por meio de uma série de `forEachs` e de chamadas à função `map`! Contudo deixaremos essa transformação para a próxima seção e, em contrapartida, focaremos na criação da UI. Vamos deixar isso (ou talvez uma versão simplificada disso) fixo no código de nossa aba Tags na forma de uma variável chamada `organizedByTag`:

```

} else if ($element.parent().is(":nth-child(3)")) {
  // ESTE É O CÓDIGO PARA A ABA TAGS
  console.log("the tags tab was clicked!");
  var organizedByTag = [
    {
      "name": "shopping",
      "toDos": ["Get groceries"]
    },

```

```

    {
      "name": "chores",
      "toDos": ["Get groceries", "Take Gracie to the park"]
    },
    /* etc. */
  ];
}

```

Agora o nosso objetivo é efetuar a iteração por esse objeto, adicionando uma nova seção ao elemento `.content` de nossa página à medida que fizermos essa operação. Para isso, basta adicionar um elemento `h3`, juntamente com um `ul` e uma lista de elementos `li` para cada tag. Os elementos `ul` e `li` são criados exatamente da mesma maneira que no capítulo anterior, portanto o estilo deve permanecer exatamente o mesmo. Também acrescentei uma estilização para o meu elemento `h3` no arquivo `style.css` para que a aparência seja semelhante à do exemplo anterior:

```

} else if ($element.parent().is(":nth-child(3)")) {
  // ESTE É O CÓDIGO PARA A ABA TAGS
  console.log("the tags tab was clicked!");

  var organizedByTag = [
    /* etc. */
  ]

  organizedByTag.forEach(function (tag) {
    var $tagName = $("<h3>").text(tag.name),
        $content = $("<ul>");

    tag.toDos.forEach(function (description) {
      var $li = $("<li>").text(description);
      $content.append($li);
    });

    $("main .content").append($tagName);
    $("main .content").append($content);
  });
}

```

Agora restam duas atividades a serem feitas. Precisamos modificar a aba Add para aceitar uma lista de categorias para um item da lista de tarefas e temos de descobrir uma maneira de modificar a nossa

lista corrente de objetos referentes às tarefas para uma lista de itens organizada de acordo com os nomes das tags. Vamos começar pela última atividade para que nossa aplicação seja dinamicamente atualizada quando adicionarmos itens novos.

Criando uma estrutura de dados intermediária para as tags

Quando estou diante de um problema que parece necessitar de mais do que algumas linhas de código para ser resolvido, geralmente isso é um indício de que devo abstraí-lo e criar uma função. Nesse caso, queremos implementar uma função chamada `organizeByTags` que receba um objeto armazenado em nosso programa desta maneira:

```
[
  {
    "description" : "Get groceries",
    "tags" : [ "shopping", "chores" ]
  },
  {
    "description" : "Make up some new Todos",
    "tags" : [ "writing", "work" ]
  },
  /* etc. */
]
```

e o converta em um objeto armazenado assim:

```
[
  {
    "name": "shopping",
    "todos": ["Get groceries"]
  },
  {
    "name": "chores",
    "todos": ["Get groceries", "Take Gracie to the park"]
  },
  /* etc. */
]
```



```
]
```

Após termos a função, podemos simplesmente modificá-la e colocá-la no lugar de nosso objeto fixo no código, desta maneira:

```
} else if ($element.parent().is(":nth-child(3)")) {  
  // ESTE É O CÓDIGO PARA A ABA TAGS  
  console.log("the tags tab was clicked!");  
  var organizedByTag = organizeByTag(toDoObjects);  
}
```

Definindo um ambiente de testes

Poderíamos tentar fazer algo como isso funcionar no console do Chrome, mas isso se torna um pouco complicado quando as funções são longas ou complexas. Minha maneira preferida de criar um ambiente de teste é implementando um programa JavaScript externo que contenha a função e testá-la exibindo informações no console. Depois que eu estiver satisfeito com o funcionamento do código, irei incorporá-lo em meu código de trabalho.

Para isso, às vezes, é interessante ter um arquivo HTML não utilizado e simples, disponível em nossa hierarquia de arquivos. Esse arquivo não faz nada além de adicionar um arquivo de script que podemos usar para efetuar experimentos. O arquivo HTML não precisa nem mesmo ter o nosso conteúdo boilerplate¹ usual:

```
<script src="http://code.jquery.com/jquery-2.0.3.min.js"></script>  
<script src="test.js"></script>
```

Se salvarmos esse arquivo como *index.html* e criarmos *test.js* com o conteúdo a seguir, devemos ver a mensagem “organizeByTag called” (organizeByTag foi chamado) no console, juntamente com o objeto de exemplo que criamos:

```
var toDoObjects = [  
  {  
    "description" : "Get groceries",  
    "tags" : [ "shopping", "chores" ]  
  },  
  {  
    "description" : "Make up some new ToDos",
```

```

        "tags" : [ "writing", "work" ]
    },
    /* etc. */
];

var organizeByTags = function (todoObjects) {
    console.log("organizeByTags called");
};

var main = function () {
    "use strict";

    var organizeByTags = function () {
        console.log("organizeByTags called");
    };

    organizeByTags(todoObjects);
};

$(document).ready(main);

```

Feito isso, sugiro que você invista um pouco de tempo tentando resolver esse problema por conta própria. É complicado, mas, após ver a solução, espero que você consiga compreendê-la. Tentar solucionar o problema por conta própria irá ajudá-lo a ter uma boa noção do quanto você é capaz de fazer.

Também vou acrescentar que há diversas maneiras de resolver esse problema, o que, com frequência, é o que ocorre quando se trata de programação de computadores. Mostrarei a minha solução; porém, geralmente, ver como a sua solução difere da minha pode ser instrutivo.

Minha solução

A minha solução é relativamente fácil de explicar e contém duas partes. Em primeiro lugar, eu crio um array que contém todas as tags possíveis, efetuando a iteração pela estrutura inicial usando um laço `forEach`. Feito isso, utilizo a função `map` para mapear o array de tags ao meu objeto desejado efetuando a iteração pela lista de tarefas e descobrindo aquelas que possuem a respectiva tag.

Vamos começar pela primeira parte. A única novidade nesse caso é a função `indexOf`, incluída em todos os arrays. Podemos ver como ela

funciona ao efetuarmos uma interação com o console do Chrome:

```
var nums = [1, 2, 3, 4, 5];
nums.indexOf(3);
//=> 2

nums.indexOf(1);
//=> 0

nums.indexOf(10);
//=> -1
```

Podemos generalizar esse comportamento: se o objeto estiver contido no array, a função retornará o índice (começando em 0) do objeto. Se não estiver no array, a função retornará -1. Podemos utilizá-la também com strings:

```
var msgs = ["hello", "goodbye", "world"];
msgs.indexOf("goodbye");
//=> 1

msgs.indexOf("hello");
//=> 0

msgs.indexOf("HEY!");
//=> -1
```

Usaremos esta função para evitar a adição de elementos duplicados ao nosso array de tags:

```
var organizeByTags = function (todoObjects) {
  // cria um array vazio de tags
  var tags = [];

  // faz a iteração por todos os todos
  todoObjects.forEach(function (todo) {
    // faz a iteração por todas as tags deste todo
    todo.tags.forEach(function (tag) {
      // verifica se a tag já não está no array de tags
      if (tags.indexOf(tag) === -1) {
        tags.push(tag);
      }
    });
  });

  console.log(tags);
}
```

```
};
```

Ao executar esse código, ele deverá exibir os nomes das tags, sem que haja repetições. Isso significa que estamos a meio caminho! Na segunda parte da solução, vou usar a função `map`:

```
var organizeByTags = function (todoObjects) {  
  /* a primeira parte anterior */  
  
  var tagObjects = tags.map(function (tag) {  
    // neste ponto, encontraremos todos os objetos referentes às tarefas,  
    // que contêm essa tag  
    var toDosWithTag = [];  
    todoObjects.forEach(function (todo) {  
      // verifica se o resultado de indexOf *não* é igual a -1  
      if (todo.tags.indexOf(tag) !== -1) {  
        toDosWithTag.push(todo.description);  
      }  
    });  
  
    // mapeamos cada tag a um objeto que  
    // contém o nome da tag e um array  
    return { "name": tag, "toDos": toDosWithTag };  
  });  
  console.log(tagObjects);  
};
```

Agora que criamos a função e que ela está funcionando corretamente, podemos incorporá-la à função `main` do código de nossa aplicação e ela deverá funcionar! Como já mencionei anteriormente, esse problema admite diversas soluções diferentes, portanto é interessante tentar descobrir outras!

As tags como parte de nossa entrada

Agora conseguimos fazer os objetos que representam as tarefas da lista ficarem organizados e serem apresentados de acordo com as tags; mas como podemos adicionar tags aos novos elementos inseridos na aba Add? Isso exige a modificação do código que apresenta a interface da aba. Gostaria de configurá-la de modo que se pareça com a figura 5.3.

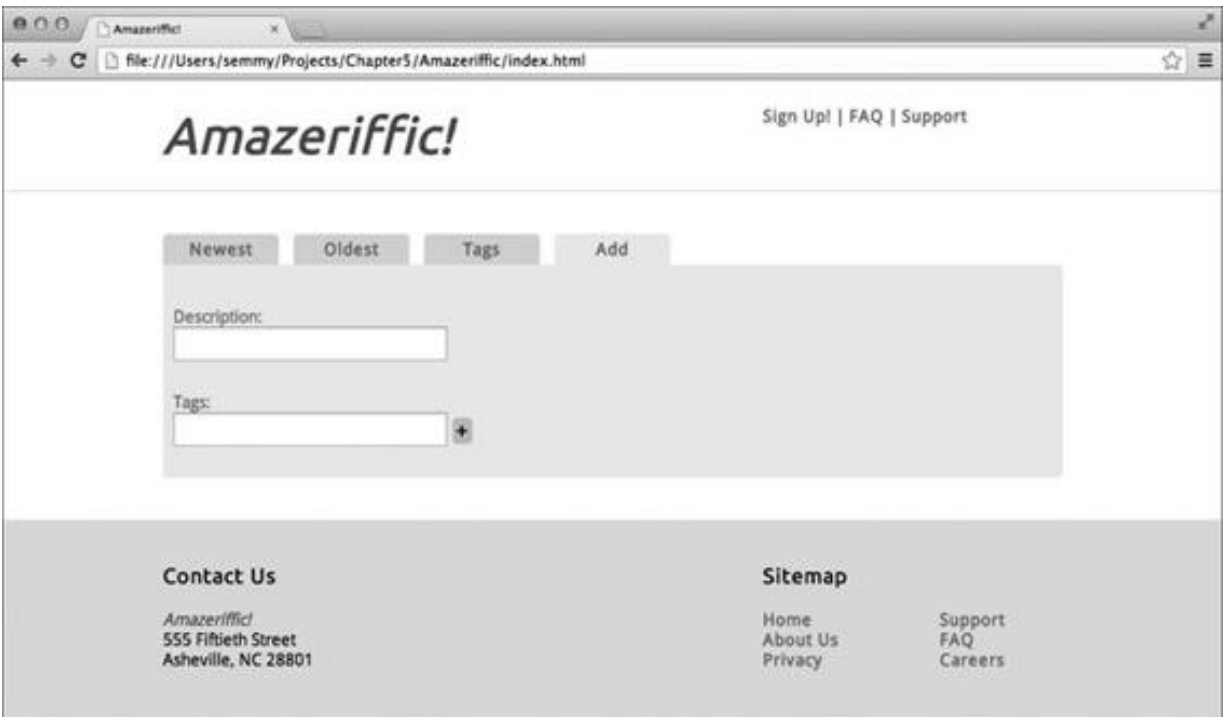


Figura 5.3 – Nosso objetivo para a aba Add.

Como você pode observar, a aba agora contém duas caixas de entrada. Na segunda caixa, deixaremos o usuário inserir uma lista de tags separadas por vírgula, que será incorporada ao objeto quando o adicionarmos.

Se você terminou o último exemplo do capítulo anterior, provavelmente terá um código semelhante a:

```
} else if ($element.parent().is(":nth-child(4)")) {
    var $input = $("<input>"),
        $button = $("<button>").text("+");

    $button.on("click", function () {
        toDos.push($input.val());
        $input.val("");
    });

    $content = $("<div>").append($input).append($button);
}
```

Para que a UI tenha um aspecto semelhante ao da figura 5.3, devemos acrescentar mais uma caixa de entrada e os labels (rótulos). Em seguida, devemos modificar o handler `$button` para que crie um array a partir das tags e o insira nos objetos.

Uma tarefa que devemos aprender a fazer é separar o objeto string. Acontece que todas as strings têm uma função pronta chamada `split` que faz exatamente isso – ela separa uma única string em um array de strings. E, assim como a maioria das demais funções que aprendemos, podemos ver como ela funciona no console do Chrome:

```
var words = "hello,world,goodbye,world";  
// isto separa o array nos pontos em que houver uma vírgula  
var arrayOfWords = words.split(",");  
console.log(arrayOfWords);  
//=> ["hello","world","goodbye","world"]  
arrayOfWords[1];  
//=> "world"  
arrayOfWords[0];  
//=> "hello"
```

Deixaremos o usuário inserir uma string de palavras separadas por vírgula que correspondem às tags, e então as adicionaremos ao objeto na forma de um array.

Por fim, teremos de recriar o nosso array `toDos` a partir do novo array `toDoObjects`. Para isso, recortei e coleí o código usado no início da função `main` (e, ao fazer isso, violei o princípio DRY – sugiro que você tente corrigir isso). O meu código final tem o seguinte aspecto:

```
} else if ($element.parent().is(":nth-child(4)")) {  
  var $input = $("<input>").addClass("description"),  
    $inputLabel = $("<p>").text("Description: "),  
    $tagInput = $("<input>").addClass("tags"),  
    $tagLabel = $("<p>").text("Tags: "),  
    $button = $("<button>").text("+");  
  
  $button.on("click", function () {  
    var description = $input.val(),  
    tags = $tagInput.val().split(","); // separa onde houver vírgula  
    toDoObjects.push({"description":description, "tags":tags});  
    // atualiza toDos  
    toDos = toDoObjects.map(function (toDo) {  
      return toDo.description;
```

```

    });
    $input.val("");
    $tagInput.val("");
    });
    $content = $("<div>").append($inputLabel)
        .append($input)
        .append($tagLabel)
        .append($tagInput)
        .append($button);
}

```

Resumo

Neste capítulo, estudamos três assuntos importantes em um nível básico: objetos JavaScript, JSON e AJAX. Apresentei os objetos JavaScript como uma maneira de armazenar dados relacionados, como se fossem uma única entidade em um programa. Os objetos são criados com o uso de chaves, e as propriedades são adicionadas e acessadas por meio do operador `.` (ponto).

Você pode pensar no JSON simplesmente como uma representação em string de um objeto JavaScript, que pode ser processado por qualquer linguagem de programação. O JSON é usado para transferir dados entre programas de computador – em nosso caso, entre o navegador web (o cliente) e o servidor. Além disso, muitos web services (incluindo o Flickr, o Twitter e o Facebook) disponibilizam *APIs* (Application Programming Interfaces, ou Interfaces de Programação de Aplicativos) que nos permitem enviar e solicitar informações de e para os serviços usando o formato JSON.

Normalmente, quando queremos enviar ou receber informações de e para um programa do lado do servidor, utilizamos uma tecnologia chamada AJAX. Ela nos permite atualizar dinamicamente as informações de nossa página web sem recarregá-la. A jQuery disponibiliza várias funções AJAX sobre as quais aprenderemos nos capítulos seguintes, embora já tenhamos visto a função `getJSON`

neste capítulo.

Práticas e leituras adicionais

Slideshow do Flickr

Neste problema, criaremos uma aplicação simples que permite ao usuário inserir um termo de pesquisa e, em seguida, apresentaremos uma série de imagens do Flickr que tenham essa tag. As imagens devem aparecer e desaparecer gradualmente (fazer fade in e fade out) na sequência.

Para isso, utilizaremos a função `setTimeout`, mencionada rapidamente no capítulo 4. Essa função nos permite agendar um evento para que aconteça após um período de tempo especificado. Por exemplo, suponha que queremos efetuar o log de “hello world!” após cinco segundos. Podemos fazer isso da seguinte maneira:

```
// exibe 'hello world' após cinco segundos
setTimeout(function () {
  console.log("hello world!");
}, 5000);
```

Observe que os argumentos dessa função estão em ordem inversa em relação ao que estamos acostumados – normalmente, a callback aparece no final. Essa é uma idiossincrasia da qual simplesmente devemos nos lembrar. Observe também que o segundo argumento representa o número de *milissegundos* que devemos esperar até que a função de callback seja executada.

Agora suponha que queremos simplesmente fazer o fade in e o fade out de um array de mensagens:

```
var messages = ["first message", "second message", "third", "fourth"];
```

Podemos começar com um corpo HTML simples:

```
<body>
  <div class="message"></div>
</body>
```

Em seguida, podemos definir nossa função `main` em *app.js* para que

faça o seguinte:

```
var main = function () {  
  var messages = ["first message", "second message", "third", "fourth"];  
  var displayMessage = function (messageIndex) {  
    // cria e oculta o elemento do DOM  
    var $message = $("<p>").text(messages[messageIndex]).hide();  
    // limpa o conteúdo corrente;  
    // é melhor selecionar o parágrafo corrente e efetuar o seu fade out.  
    $(".message").empty();  
    // adiciona a mensagem com messageIndex ao DOM  
    $(".message").append($message);  
    // faz o fade in da mensagem  
    $message.fadeIn();  
    setTimeout(function () {  
      // Dentro de 3 segundos, chama displayMessage novamente com o  
      // próximo índice  
      messageIndex = messageIndex + 1;  
      displayMessage(messageIndex);  
    }, 3000);  
  };  
  displayMessage(0);  
};  
$(document).ready(main);
```

Esse código fará as mensagens aparecerem gradualmente na sequência, efetuando o ciclo a cada três segundos. No entanto há um pequeno problema aqui – ao chegarmos ao final, começaremos a ver a palavra `undefined` aparecer porque teremos ultrapassado o final do array. Podemos corrigir esse problema por meio da inclusão de uma instrução `if` para verificar se estamos no final e, em caso afirmativo, definir o índice de volta para 0.



Provavelmente não vamos querer de fato acrescentar esse efeito em uma página web. É um equivalente moderno da terrível tag `<blink>`, eliminada do HTML porque era extremamente irritante.

Agora podemos efetuar generalizações. Inicialmente, devemos fazer uma solicitação AJAX para obter os dados das imagens do Flickr e, em seguida, em vez de adicionar um elemento do tipo parágrafo ao

DOM, acrescentaremos um elemento `img` no lugar de `p`. Definiremos o atributo `src` do elemento `img` com a imagem do Flickr.

Depois que isso estiver funcionando, criaremos uma interface que permita ao usuário inserir uma tag para pesquisar e então criamos o slideshow com as imagens que incluam a tag especificada. Esse é um ótimo exercício que deverá consumir um pouco de tempo, especialmente se você decidir adicionar um pouco de estilização básica e realmente efetuar a coordenação correta das imagens. Recomendo que você tente fazer isso!

Exercício com objetos

Um de meus problemas prediletos para compartilhar com os programadores iniciantes relaciona-se com a identificação de mãos de pôquer. Tudo bem se você não estiver familiarizado com a família de jogos de cartas baseada no pôquer. Todos os jogos estão relacionados a padrões que ocorrem em mãos de cinco cartas de baralho. Podemos ter as seguintes mãos:

Par (Pair)

Duas cartas de mesmo valor.

Dois pares (Two pair)

Duas cartas de mesmo valor, mais duas cartas de outro valor.

Trinca ou trio (Three of a kind)

Três cartas de mesmo valor.

Sequência (Straight)

Cinco valores em sequência, porém um ás pode contar como o maior ou o menor valor.

Flush

Cinco cartas do mesmo naipe.

Full house

Duas cartas de mesmo valor, mais três cartas de outro valor.

Quadra (Four of a kind)

Quatro cartas de mesmo valor.

Sequência de mesmo naipe (Straight flush)

Cinco cartas do mesmo naipe, em que os valores formam uma sequência.

Sequência real (Royal flush)

Uma sequência de mesmo naipe, iniciada com um 10 e terminada com um ás.

Iremos nos referir a qualquer mão que não atenda a um desses critérios como *bust*. Observe que as mãos não são mutuamente exclusivas – uma mão que contenha uma *trinca* (three of a kind) também contém um *par* (pair); um *full house* contém *dois pares* (two pair). Nesse conjunto de problemas, criaremos funções JavaScript que testam se um array de cinco cartas apresenta uma dessas propriedades.

Há diversas maneiras de resolver esses problemas e algumas são muito mais eficientes ou exigem muito menos código do que outras. Contudo nosso objetivo aqui é exercitar o uso de objetos, arrays e instruções condicionais. Portanto seguiremos uma abordagem que nos permita criar uma série de funções auxiliares que possamos usar para determinar se um conjunto de cinco cartas corresponde a um determinado tipo de mão.

Para começar, vamos ver como será a aparência de uma mão:

```
var hand = [  
  { "rank": "two", "suit": "spades" },  
  { "rank": "four", "suit": "hearts" },  
  { "rank": "two", "suit": "clubs" },  
  { "rank": "king", "suit": "spades" },  
  { "rank": "eight", "suit": "diamonds" }  
];
```

Nesse exemplo, a nossa mão contém um par de dois. Pode ser uma

boa ideia criar um exemplo para cada um dos demais tipos de mão antes de prosseguir.

Como determinar se uma mão atende a um dos critérios? Iremos reduzir o problema a uma situação resolvida no final do capítulo 4! Se você acabou de resolver aqueles problemas, deverá se lembrar de que fiz você criar uma função chamada `containsNTimes`, a qual aceitava três argumentos: um array, um item para pesquisar e um número mínimo de vezes que o item deveria aparecer. Agora imagine se enviássemos um array de valores de cartas àquela função:

```
// há dois 2
containsNTimes(["two","four","two","king","eight"], "two", 2);
//=> true
```

Esse código nos informa que o array de valores contém um par! De modo semelhante, podemos usar essa função para determinar se há uma trinca ou uma quadra!

```
// não há 3 dois
containsNTimes(["two","four","two","king","eight"], "two", 3);
//=> false
```

Desse modo, reduzimos o problema a transformar o nosso array de objetos carta em um array de valores. Acontece que é muito fácil fazer isso usando a função `map` que aprendemos neste capítulo:

```
// nossa "mão" corresponde ao array hand definido anteriormente
hand.map(function (card) {
  return card.rank;
});
//=> ["two","four","two","king","eight"]
```

Podemos então atribuir o resultado da chamada à `map` a uma variável e, em seguida, enviá-la para a nossa função `containsNTimes` para determinar se ela contém um par de dois:

```
var handRanks,
    result;

handRanks = hand.map(function (card) {
  return card.rank;
```

```
});
// result contém 'true'
result = containsNTimes(handRanks, "two", 2);
```

Para finalizar, criamos um array de todos os valores possíveis do baralho e usamos um laço `forEach` para determinar se a mão contém um par de *qualquer um* desses valores:

```
// estes são todos os valores possíveis
var ranks = ["two","three","four","five","six","seven","eight",
    "nine","ten","jack","queen","king","ace"];

var containsPair = function (hand) {
    // iremos supor que não há um par
    // até encontrarmos evidências que provem o contrário
    var result = false,
        handRanks;

    // cria o nosso array de valores das mãos
    handRanks = hand.map(function (card) {
        return card.rank;
    });

    // procura um par de qualquer valor
    ranks.forEach(function (rank) {
        if (containsNTimes(handRanks, rank, 2)) {
            // encontramos um par!
            result = true;
        }
    });

    // essa variável estará definida com true se um par foi encontrado;
    // será false, caso contrário
    return result;
};
```

Agora que já vimos um exemplo, procure efetuar generalizações de modo a criar uma função para cada um dos demais valores (por exemplo, `containsThreeOfAKind`). Para dois pares e um full house, será preciso guardar os valores dos elementos encontrados. No caso do flush, será preciso extrair os naipes do array de objetos, em vez dos valores.

Uma sequência (straight) é um pouco mais difícil, mas aqui vai uma

dica: uma sequência não contém nenhum par (o que pode ser determinado pela inversão do resultado da função `containsPair` usando o operador `!`), e a diferença entre a carta de maior valor e a de menor valor é 4. Sendo assim, é uma boa ideia criar algumas funções auxiliares para descobrir o valor da carta mais alta e da carta mais baixa na forma de números (por exemplo, um valete seria 11, uma dama seria 12, um rei, 13, e um ás seria 14). Depois que tiver essas funções, você poderá determinar se uma mão contém uma sequência ao confirmar que ela não contém um par e que a diferença entre a maior e a menor carta é igual a 4.

Você também pode implementar o restante das funções utilizando as funções existentes. Por exemplo, uma sequência de mesmo naipe (straight flush) é somente um flush que também contém uma sequência. E uma sequência real (royal flush) é uma sequência de mesmo naipe (straight flush) em que a carta mais alta é um ás (ou a carta mais baixa é um 10).

Outras APIs

Agora que você já viu que pode facilmente extrair dados do Flickr e trabalhar com os resultados, talvez seja divertido extrair dados de outras APIs. O fato é que muitas APIs permitem acessar dados da mesma maneira que acessamos os dados do Flickr.

O Programmable Web (<http://www.programmableweb.com/>) mantém uma lista dessas APIs. Para que o site funcione com a função `getJSON` da jQuery, ele deve suportar uma tecnologia chamada *JSONP*. Você pode obter uma lista de APIs que suporte JSONP (<http://www.programmableweb.com/apis/directory/1?format=JSONP>) e pode ler mais a respeito desse assunto na Wikipedia (<http://en.wikipedia.org/wiki/JSONP>).

Será necessário ler a documentação da API específica para determinar a maneira de formatar a sua URL para efetuar a query, mas experimentar o uso de outras APIs é uma ótima maneira de exercitar as suas habilidades com o AJAX.

¹ N.T.: Seções de código que podem ser incluídas em vários lugares com pouca ou nenhuma alteração (Fonte: http://en.wikipedia.org/wiki/Boilerplate_code).

CAPÍTULO 6

Servidor

A essa altura, já vimos as principais tecnologias associadas ao lado do cliente em uma aplicação web. Também aprendemos um pouco sobre o modo como o navegador se comunica com um servidor usando JSON e AJAX. A seguir, vamos mergulhar de cabeça na programação do lado do servidor.

Entender a parte da aplicação referente ao lado do servidor exigirá aprender mais sobre o modelo cliente-servidor, o protocolo HTTP e o Node.js. O Node.js é uma tecnologia relativamente nova (e empolgante) que nos permite facilmente criar servidores orientados a eventos utilizando o JavaScript.

Configurando o ambiente

Configurar um ambiente de desenvolvimento que suporte a criação de aplicações orientadas a banco de dados pode ser uma tarefa assustadora, e descrever o procedimento para o Windows, o Mac OS e o Linux está muito além do escopo deste livro. Para simplificar o processo, criei um conjunto de scripts que deixará tudo pronto para você, de maneira relativamente rápida, utilizando o Vagrant e o VirtualBox.

O Vagrant é uma ferramenta que ajuda a criar um ambiente de desenvolvimento em uma máquina virtual. Você pode pensar em uma máquina virtual como um computador separado sendo executado de forma totalmente confinada em seu computador. Daremos mais explicações sobre esse assunto em “Modelos mentais”. Por enquanto, basta saber que usaremos o Vagrant,

juntamente com o VirtualBox, para criar um ambiente de desenvolvimento virtual para o lado do servidor. Esse ambiente incluirá a maioria das ferramentas que usaremos no restante deste livro.

Obviamente, parte desse processo diz respeito à conveniência: mesmo que a instalação de um ambiente de desenvolvimento com Node.js em seu computador local seja muito fácil (e vou sugerir que você faça isso ao resolver os problemas práticos que estão no final do capítulo), adicionar uma pilha completa de servidor que inclua o MongoDB e o Redis não é uma tarefa trivial e exige um pouco de tempo e paciência.

Outro motivo pelo qual estamos fazendo isso tem a ver com a consistência. Como estou escrevendo os scripts, tenho um pouco mais de controle sobre as versões do Node.js, do MongoDB e do Redis que serão instalados. Além do mais, usar o Vagrant nos permite ter o mesmo ambiente de desenvolvimento em execução, independentemente de você estar usando o Windows, o Mac OS ou o Linux. Isso minimizará qualquer problema que surgir em consequência de diferenças entre os sistemas operacionais (ou assim espero).

Também acho que executar o nosso ambiente de desenvolvimento dentro do VirtualBox faz com que a parte da aplicação referente ao servidor fique separada de maneira bem clara. Isso cria uma abstração bastante útil do ponto de vista pedagógico para alguém que está aprendendo o básico sobre o desenvolvimento de aplicações web.

Instalando o VirtualBox e o Vagrant

Inicialmente, será necessário instalar a versão mais recente do VirtualBox. Na época desta publicação, essa versão era a 4.2.18. Você pode acessar <http://www.virtualbox.org> (mostrado na figura 6.1), clicar no link Downloads à esquerda da página e efetuar o download da versão adequada ao seu sistema operacional. Feito o

download, o processo de instalação e de configuração será diferente de acordo com o seu sistema operacional, portanto não se esqueça de seguir as instruções apropriadas.



Figura 6.1 – A página inicial do VirtualBox.

A seguir, será necessário instalar a versão mais recente do Vagrant. Na época desta publicação, essa versão era a 1.4. Para obter a versão mais recente, acesse <http://www.vagrantup.com> (veja a figura 6.2), clique no link Downloads no canto superior direito e obtenha o instalador mais recente para a sua plataforma. Após efetuar o download, dê um clique duplo no pacote e instale-o da mesma maneira que você instalou o VirtualBox.

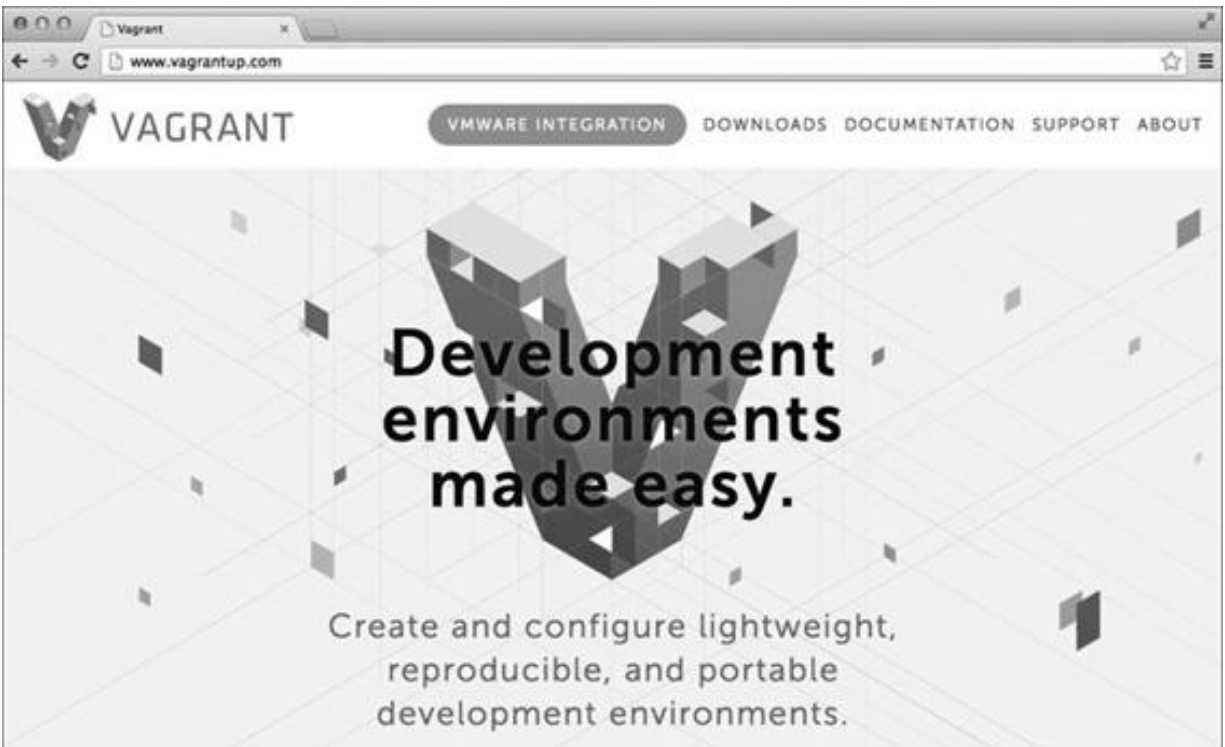


Figura 6.2 – A página inicial do Vagrant.

Se você estiver executando o Windows, pode ser que seja necessário reiniciar o seu computador após a instalação para garantir que o path de seus arquivos seja configurado corretamente. Isso pode ser testado ao abrir uma janela de comando (clique no menu Start [Iniciar] e digite `cmd` na caixa de pesquisa) e digitar `vagrant --version` no prompt:

```
C:\Users\semmmy>vagrant --version  
Vagrant version 1.4.0
```

Se isso funcionar, você estará pronto para começar. Caso contrário, reinicie o seu computador e tente digitar o comando `vagrant` novamente.

Criando a sua máquina virtual

Se tudo correr bem, você estará pronto para clonar o repositório Git chamado `node-dev-bootstrap` a partir de minha página no GitHub. Basicamente, isso significa que você fará o download de um repositório Git completo criado por mim e o colocará em seu

computador. Se estiver usando o Windows, você deverá acessar o prompt do `git-bash`. Qualquer que seja o caso, vá para o seu diretório *Projects* e clone o repositório `node-dev-bootstrap` utilizando o comando a seguir:

```
hostname $ git clone https://github.com/semmypurewal/node-dev-bootstrap
Chapter6
```

Esse comando fará um diretório de nome *Chapter6* ser criado e clonará o repositório `node-dev-bootstrap` nesse diretório. A seguir, entre no diretório do projeto e digite o comando `vagrant up`:

```
hostname $ cd Chapter6
hostname $ vagrant up
```



Se você estiver executando o Windows, pode ser que você receba um aviso do firewall perguntando se o programa “`vboxheadless`” pode ter permissão de acesso à rede. É seguro responder que “sim”.

Se tudo correu bem, o Vagrant criará a sua máquina virtual. Esse processo levará alguns minutos. Tenha paciência.

Conectando-se à sua máquina virtual usando o SSH

Após terminada a instalação, a sua máquina virtual deverá estar executando. Como podemos verificar? Você deve usar uma tecnologia de rede chamada SSH (que quer dizer Secure Shell) para se “conectar” ao servidor por meio de sua janela do terminal.

Se você estiver usando o Mac OS ou o Linux, será bem fácil, porque a sua plataforma já vem com um cliente SSH incluído. Porém, se você estiver no Windows, pode ser que seja necessário instalar manualmente um cliente SSH.

De qualquer maneira, vá em frente e digite:

```
hostname $ vagrant ssh
```

Se você estiver no Mac OS, esse comando fará a conexão com a sua máquina virtual. Se o Windows estiver sendo executado, isso poderá ou não funcionar, dependendo da versão. Se não funcionar, o Vagrant disponibilizará credenciais de login para você (é bem

provável que o host seja 127.0.0.1 e a porta seja 2222). Faça o download de um cliente SSH para efetuar a conexão – eu recomendo o PuTTY, que está disponível na página de download desse aplicativo (<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>). A página inicial do PuTTY está sendo mostrada na figura 6.3.

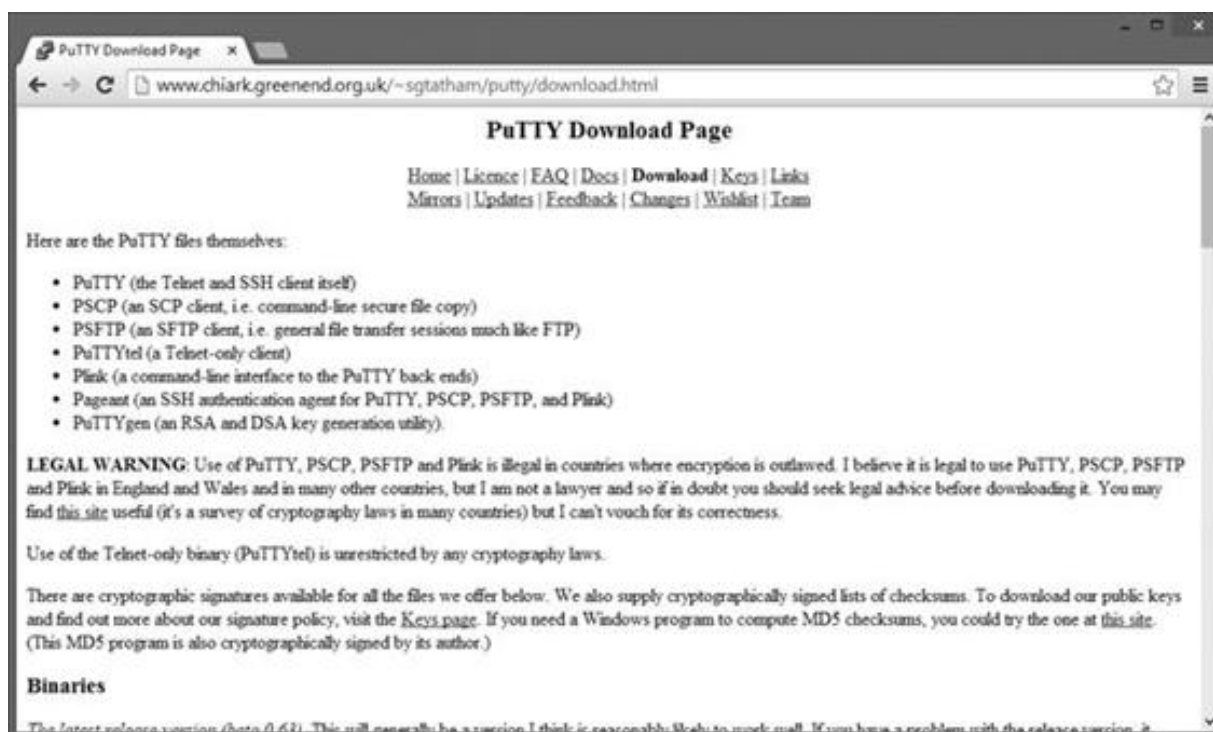


Figura 6.3 – A página inicial e de download do PuTTY.

Ao abrir o PuTTY, digite o nome do host e a porta especificados pelo Vagrant nas caixas de entrada apropriadas (Host Name e Port), conforme mostrado na figura 6.4.

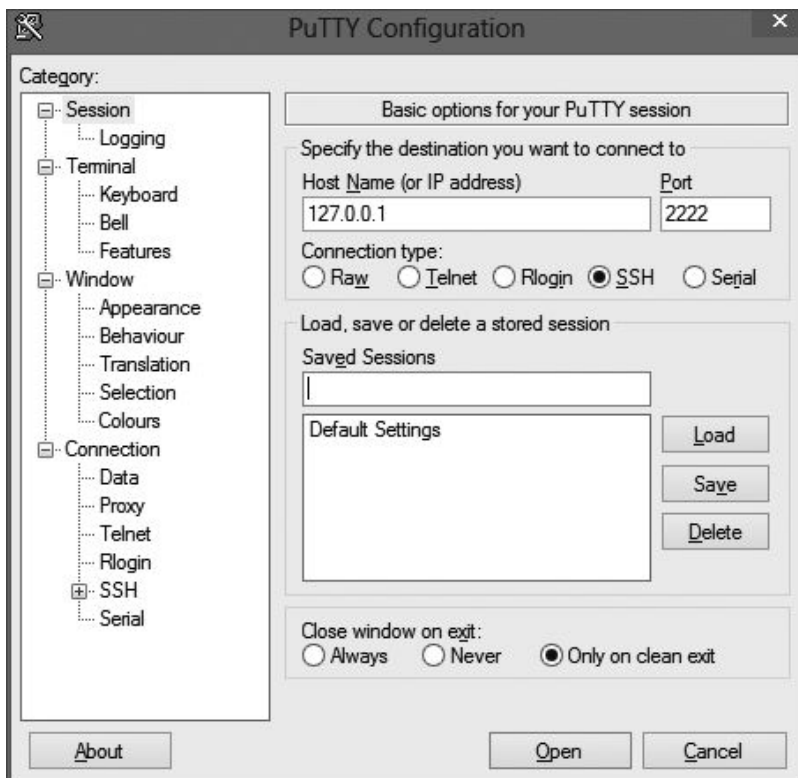


Figura 6.4 – Configurando o PuTTY para se conectar à sua máquina virtual.

Em seguida, clique em Open (Abrir). O PuTTY fará a conexão com o seu servidor virtual e pedirá um nome de usuário e uma senha. Ambos devem estar previamente definidos como “vagrant”.

Hello, Node.js!

Após ter feito login, você poderá navegar pelo shell do terminal de seu computador remoto da mesma maneira que você navega no shell do terminal de seu computador local. Por exemplo, você pode listar o conteúdo do diretório de sua máquina virtual da mesma maneira que você examina o conteúdo de um diretório em seu computador local – ou seja, por meio do comando `ls`.

```
vagrant $ ls  
app postinstall.sh
```

Por enquanto, você pode ignorar o arquivo *postinstall.sh*. Usando as habilidades relacionadas ao uso da linha de comando adquiridas nos capítulos anteriores, vá até o diretório *app* de sua máquina

virtual:

```
vagrant $ cd app  
vagrant $ ls  
server.js
```

Você deverá ver um arquivo chamado `server.js` nesse diretório. Vamos garantir que tudo está funcionando inicializando-o por meio do comando `node`:

```
vagrant $ node server.js  
Server listening on port 3000
```

Você verá uma mensagem com a informação “Server listening on port 3000” (Servidor está ouvindo na porta 3000) e o prompt do terminal não estará mais disponível. Isso ocorre porque o seu servidor está sendo executado. Essa condição pode ser confirmada ao abrir o Chrome e digitar `localhost:3000` na barra de endereço. Se tudo estiver funcionando corretamente, você deverá ver a mensagem “Hello World!”. A aparência da tela deverá ser semelhante àquela mostrada na figura 6.5.

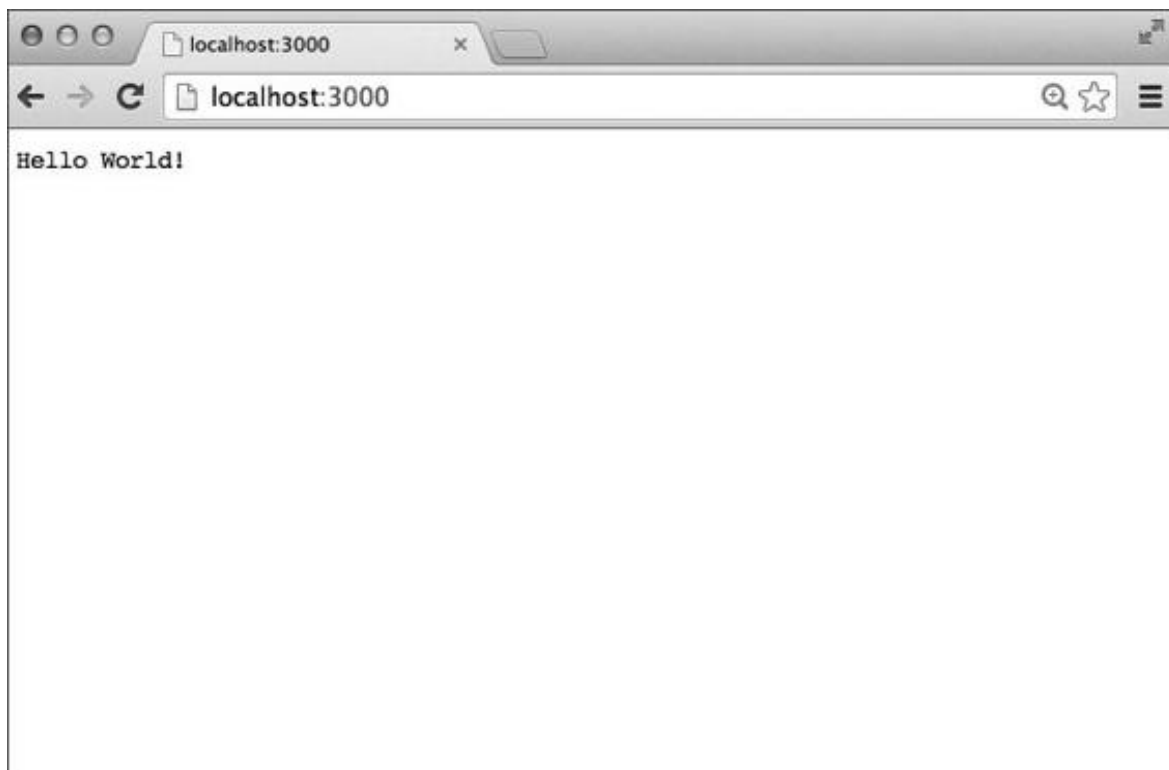


Figura 6.5 – O servidor Node default visualizado a partir do Chrome.

Vamos começar pelo início: o programa pode ser encerrado a partir da janela do terminal usando as teclas Ctrl-C. Após terminado o seu trabalho, você poderá sair do servidor virtual ao digitar **exit**:

```
vagrant $ exit  
logout  
Connection to 127.0.0.1 closed.
```

Se você efetuou login em sua máquina virtual usando o PuTTY no Windows, esse comando encerrará o programa. Então você poderá retornar ao terminal utilizado para iniciar a sua máquina virtual. Se estiver no Mac OS, você retornará imediatamente para a linha de comando. De qualquer maneira, o servidor pode ser encerrado por meio do comando **vagrant halt**:

```
hostname $ vagrant halt  
[default] Attempting graceful shutdown of VM...
```

Você deve encerrar a sua VM sempre que ela não estiver sendo utilizada, pois ter uma máquina virtual executando em background certamente irá consumir os recursos de seu computador. Após ter sido encerrado, o seu box¹ poderá ser reiniciado por meio do comando **vagrant up**.



Para remover completamente a máquina virtual criada pelo Vagrant de seu computador, digite **vagrant destroy**. Isso resultará na recriação de sua VM na próxima vez que você digitar **vagrant up** nesse diretório. Portanto eu recomendo ater-se ao **vagrant halt** por enquanto.

Modelos mentais

Nesta seção, discutiremos algumas maneiras de pensar em clientes, servidores, hosts e guests.

Clientes e servidores

Na área de redes de computadores, normalmente pensamos nos programas de computador como programas *clientes* ou programas *servidores*. Tradicionalmente, um programa *servidor* faz a abstração de alguns recursos que vários programas *clientes* querem acessar

por meio de uma rede. Por exemplo, alguém pode querer transferir arquivos de um computador remoto para um computador local. Um servidor FTP é um programa que implementa o *File Transfer Protocol* (Protocolo de Transferência de Arquivos), que permite aos usuários fazer exatamente isso, ou seja, transferir arquivos. Um cliente FTP é um programa que pode se conectar a um servidor FTP e transferir programas a partir daí. O modelo cliente-servidor mais comum está sendo mostrado na figura 6.6.

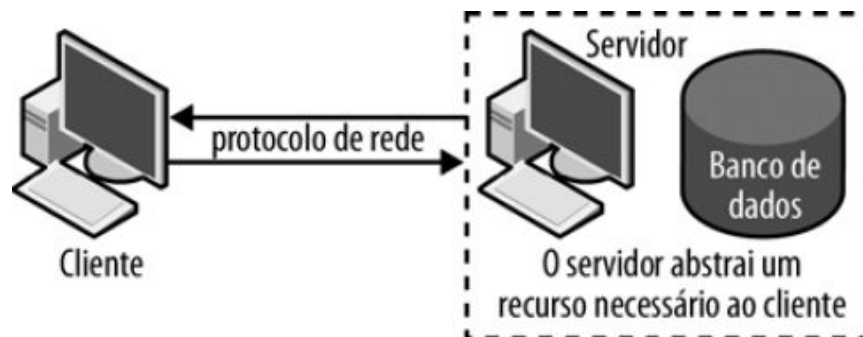


Figura 6.6 – O modelo cliente-servidor.

Há muitos detalhes secundários ao desenvolvimento de aplicações web no mundo das redes de computadores, porém vale a pena entender alguns pontos importantes.

O primeiro é que (na maioria das vezes) o cliente é um navegador web e o servidor é um computador remoto que efetua a abstração dos recursos por meio do *Hypertext Transfer Protocol* (Protocolo de Transferência de Hipertexto), ou HTTP, para ser mais conciso. Embora tenha sido originalmente concebido para transferir documentos HTML entre computadores, o protocolo HTTP atualmente pode ser utilizado para abstrair vários tipos diferentes de recursos em um computador remoto (por exemplo, documentos, bancos de dados ou qualquer outro tipo de armazenamento). Teremos muito mais a dizer sobre o HTTP posteriormente neste livro, porém, por enquanto, podemos pensar nele como sendo o protocolo que estamos usando para conectar navegadores a computadores remotos.

Os nossos servidores HTTP serão usados para enviar a parte de

uma aplicação referente ao lado do cliente, que será interpretada pelo navegador web. Em particular, todo o HTML, o CSS e o JavaScript que aprendemos até agora serão enviados ao navegador por meio do servidor. O programa do lado do cliente que estiver em execução no navegador será responsável por obter ou enviar informações de e para o nosso servidor, normalmente por meio de AJAX.

Hosts e guests

Normalmente, o nosso servidor HTTP executa em um computador remoto (com efeito, mais adiante no livro, faremos isso acontecer). Isso causa problemas aos desenvolvedores – se estivermos executando o nosso código em um servidor remoto, será necessário editar o código nesse servidor e reiniciá-lo, ou teremos de editar o código localmente e enviá-lo sempre que for necessário testá-lo. Esse processo pode ser altamente ineficiente.

Estamos contornando esse problema ao executar o servidor localmente em nosso computador. Na verdade, estamos dando um passo além. Em vez de simplesmente executar um programa servidor e todo o software localmente, estamos criando uma máquina virtual que executará o servidor dentro de nosso computador local. Para nós, é quase como se fosse um computador remoto (o que significa que faremos a conexão com ele por meio de SSH, que é a maneira pela qual nos conectamos aos computadores remotos de verdade). Porém, pelo fato de estarmos executando o servidor localmente, podemos compartilhar facilmente a nossa pasta de desenvolvimento com o computador local para podermos editar os arquivos.

O nosso computador local (que está executando a máquina virtual) será chamado de computador *host*. A máquina virtual será chamada de computador *guest*. No restante deste capítulo, ambos estarão executando e farei a diferenciação entre eles.

Aspectos práticos

Essa abstração pode resultar em certa confusão – que vale a pena esclarecer o mais rápido possível. Inicialmente, é uma boa ideia listar as aplicações que estarão sendo executadas no processo de desenvolvimento e a janela à qual elas estarão associadas:

O navegador

Você usará o Chrome para testar as alterações feitas em sua aplicação.

O editor de texto

Se você está nos acompanhando até agora, sabe que provavelmente esse aplicativo é o Sublime. Ele permitirá editar os arquivos que estiverem em uma pasta compartilhada, tanto no computador host quanto no guest.

Git

Você terá uma janela de terminal aberta para o Git. O Git será usado a partir do computador host. Se você estiver no Mac OS ou no Linux, será uma janela do terminal. Se estiver no Windows, será o prompt do `git-bash`.

Vagrant

É mais provável que você vá interagir com o Vagrant usando a mesma janela utilizada para efetuar a interação com o Git. Isso significa que você pode digitar `vagrant up` e `vagrant halt` no mesmo lugar em que digitar `git commit`.

SSH

Esse aplicativo será executado em seu computador host, porém você o utilizará para interagir com o computador guest. Se estiver no Windows, esse aplicativo será o PuTTY ou uma janela adicional do `git-bash`, conforme o comando `vagrant ssh` conseguir conectar você diretamente ou não. Se estiver no Mac OS, ter uma janela adicional do terminal aberta será uma boa ideia.

Normalmente, eu tenho pelo menos duas janelas de terminal abertas, além do navegador e do meu editor de texto. Isso está parcialmente sendo mostrado na figura 6.7, embora a janela do editor de texto não esteja sendo apresentada.

Hello, HTTP!

Agora que entendemos, de modo geral, o relacionamento entre a máquina virtual guest e o nosso computador host, vamos tentar entender o código propriamente dito. Como já mencionei anteriormente, a grande vantagem de executar o nosso código em uma máquina virtual em nosso computador local (*versus* executá-lo remotamente) é que podemos acessar e editar o código diretamente em nosso computador local utilizando qualquer editor de texto escolhido por nós.



Figura 6.7 – O git-bash, o PuTTY e o Chrome executando no Windows.

Utilizando o Sublime, abra o diretório *app* que está no diretório *Chapter6*. Você deverá ver um arquivo chamado *server.js*, que deve

ter um aspecto semelhante a:

```
var http = require("http"),
    server;

server = http.createServer(function (req, res) {
  res.writeHead(200, {"Content-Type": "text/plain"});
  res.end("Hello World!");
});

server.listen(3000);

console.log("Server listening on port 3000");
```

Esse código não deve lhe parecer totalmente estranho – você deverá ser capaz de identificar imediatamente algumas declarações de variáveis, uma instrução `console.log`, uma função anônima, um objeto JSON e o padrão de callback usados quando criamos elementos de UI nos capítulos anteriores.

Mas o que esse código faz? O fato é que ele faz muita coisa – ele cria um servidor HTTP que responde às solicitações do navegador web! Conforme mencionado anteriormente, HTTP quer dizer *Hypertext Transfer Protocol* (Protocolo de Transferência de Hipertexto) e é a tecnologia básica por trás da World Wide Web! Você já deve ter ouvido falar de programas como o Apache ou o Nginx – eles são programas servidores de HTTP configuráveis, com forte presença no mercado, projetados para hospedar sites de grande porte.

O nosso servidor HTTP é muito mais simples: ele somente aceita uma solicitação do navegador e responde com uma mensagem de texto que contém “Hello World”.

Podemos pensar no código como algo que se comporta exatamente como o handler `click` da jQuery – a diferença é que, em vez de a callback ser chamada quando um usuário clicar, ela será chamada sempre que um cliente (nesse caso, o navegador) se conectar. O parâmetro `req` corresponde a um objeto que representa a solicitação HTTP do cliente, a qual chega até o nosso servidor; o parâmetro `res` é um objeto que representa a resposta HTTP a ser enviada de volta

ao cliente. A função `res.writeHead` cria o cabeçalho HTTP que define os atributos da resposta, e a função `res.end` completa a resposta ao adicionar “Hello World”.

Após termos criado o servidor, fazemos com que ele fique *ouvindo* na porta 3000 (motivo pelo qual digitamos ~~3000~~ após `localhost` na barra de endereço de nosso navegador web). A instrução `console.log` exibe a frase no terminal do servidor quando executamos o programa.

Os módulos e o Express

Nossa! Isso provavelmente parece muito para um programa tão pequeno! É mais do que você deve ter se dado conta – um servidor HTTP é constituído de uma porção não trivial de software que exige uma boa dose de habilidades para ser implementado de forma correta. Felizmente, não precisamos nos preocupar com os detalhes porque importamos o código do módulo `http` do Node.js. Um módulo nada mais é do que um conjunto de códigos que podemos usar sem que seja necessário compreender totalmente o seu funcionamento interno – tudo o que devemos conhecer é a API que o módulo nos expõe. A primeira linha de nosso código corresponde à instrução `require`, que importa o módulo `http` e o armazena na variável `http`.

Esse módulo de servidor HTTP é interessante se precisarmos de um servidor HTTP simples, sem sofisticções, que somente aceite e responda às solicitações dos clientes. No entanto, quando quisermos começar a enviar arquivos HTML ou CSS a partir do servidor, a situação se tornará muito mais complicada. Podemos criar um servidor mais complexo com base no servidor HTTP básico disponibilizado pelo Node.js, mas, felizmente, alguém já resolveu esse problema também! O módulo Express cria uma camada sobre o módulo `http` nuclear; o Express cuida de vários aspectos complexos com os quais não queremos lidar, como servir HTML estático, arquivos CSS e JavaScript do lado do cliente.

Na verdade, uma das maravilhas de programar no Node é que podemos tirar vantagem de vários módulos como o Express, muitos

dos quais executam tarefas muito úteis. O módulo `http` usado anteriormente faz parte da distribuição principal do Node.js, portanto não precisamos fazer nada de especial para usá-lo. O Express não faz parte da distribuição principal do Node, portanto é necessário um pouco mais de trabalho para que possamos acessá-lo tão facilmente quanto acessamos o módulo `http`.

Felizmente, toda distribuição do Node vem acompanhada de outro programa chamado *Node Package Manager*, ou gerenciador de pacotes do Node (NPM para ser mais conciso). Essa ferramenta permite instalar módulos facilmente e tirar vantagem imediata desses módulos em nosso código.

Instalando o Express com o NPM

O NPM é bem fácil de usar. Vamos começar pela criação de um novo diretório dentro do diretório *app*, que se chamará *Express*. Isso pode ser feito no Sublime ou na janela do terminal. Qualquer que seja a maneira, após o diretório ter sido criado, você deve acessar o box do Vagrant por meio do SSH e navegar até o diretório *Express*. Em seguida, digite `npm install express@3`:

```
vagrant $ ls
app postinstall.sh
vagrant $ cd app
vagrant $ ls
Express server.js
vagrant $ cd Express
vagrant $ npm install express@3
```



Ocasionalmente, o NPM e o VirtualBox não funcionarão bem juntos em um computador Windows. Se ocorrerem erros quando você tentar instalar o Express, tente executar o comando `npm install express@3 --no-bin-links` e veja se isso ajuda.

O NPM responderá instalando o Express, além de uma longa lista de dependências. Em que local eles são instalados? O fato é que eles são instalados em seu diretório corrente, dentro de um subdiretório chamado *node_modules*. Podemos confirmar isso facilmente:

```
vagrant $ pwd
/home/vagrant/app/Express
vagrant $ ls
server.js node_modules
vagrant $ ls node_modules
express
```

Nosso primeiro servidor Express

Agora que o módulo `Express` está instalado, podemos criar um servidor `Express` simples. Vamos criar um arquivo chamado `server.js` em nosso diretório `Express` e adicionar o conteúdo a seguir:

```
var express = require("express"),
    http = require("http"),
    app;

// Cria o nosso servidor HTTP baseado no Express
// e faça-o ficar ouvindo na porta 3000
app = express();
http.createServer(app).listen(3000);

// define as nossas rotas
app.get("/hello", function (req, res) {
    res.send("Hello World!");
});
app.get("/goodbye", function (req, res) {
    res.send("Goodbye World!");
});
```

Esse código pode fazer você se lembrar do exemplo anterior, porém há algumas diferenças perceptíveis. Em primeiro lugar, não precisamos definir cabeçalhos HTTP nas callbacks porque o `Express` faz isso por nós. Além disso, estamos usando apenas `res.send` em vez de usar `res.write` ou `res.end`. E, por fim, mas provavelmente é o aspecto mais importante, definimos duas rotas: `hello` e `goodbye`. Veremos o que isso faz quando abrirmos o nosso navegador, porém, antes de tudo, vamos inicializar o servidor.

Lembre-se de que podemos fazer isso ao digitar `node server.js` no diretório `Express` de nosso computador guest. Agora, quando

acessarmos o nosso navegador web, poderemos digitar *localhost:3000*, como fizemos anteriormente. Desta vez, porém, devemos ver um erro contendo a mensagem “Cannot GET /”. Entretanto, se digitarmos *localhost:3000/hello* ou *localhost:3000/goodbye*, deveremos ver a mensagem especificada nas callbacks. Experimente fazer isso agora.

Como você pode ver, a adição de *hello* e de *goodbye* após o URL principal de nossa aplicação especifica a função que será disparada. E você também perceberá que o Express não define uma rota default para nós. Se quisermos fazer com que *localhost:3000/* funcione como antes, basta definirmos uma rota raiz por meio da adição de três outras linhas de código em *server.js*:

```
app.get("/", function (req, res) {  
  res.send("This is the root route!");  
});
```

Se encerrarmos o nosso servidor (teclando Ctrl-C) e o iniciarmos novamente, poderemos acessar *localhost:3000* como fizemos antes! O Express está cuidando das complexidades associadas ao roteamento – nós simplesmente dizemos a ele o que queremos fazer quando determinadas rotas são solicitadas.

Enviando a sua aplicação cliente

Vimos que podemos enviar informações ao navegador web a partir do servidor. Mas e se quiséssemos enviar algo como uma página HTML básica? Então a situação poderia se complicar de forma relativamente rápida. Por exemplo, podemos tentar fazer algo como:

```
app.get("/index.html", function (req, res) {  
  res.send("<html><head></head><body><h1>Hello World!</h1></body></html>");  
});
```

Embora isso vá funcionar, criar um HTML que seja maior do que esse pequeno exemplo não será nada prático. Felizmente, o Express resolve esse problema para nós ao permitir que ele seja usado como um *servidor de arquivos estáticos*. Ao fazer isso,

podemos criar arquivos HTML, CSS e JavaScript do lado do cliente, como viemos fazendo no restante do livro! E o fato é que isso exige somente uma linha adicional de código. Podemos modificar o arquivo *server.js* como a seguir:

```
var express = require("express"),
    http = require("http"),
    app = express();

// cria um diretório de arquivos estáticos a ser usado para o roteamento default;
// veja também a observação mais adiante a respeito do Windows
app.use(express.static(__dirname + "/client"));

// Cria o nosso servidor HTTP baseado no Express
http.createServer(app).listen(3000);

// define as nossas rotas
app.get("/hello", function (req, res) {
    res.send("Hello World!");
});

app.get("/goodbye", function (req, res) {
    res.send("Goodbye World!");
});
```



Se você tentar executar esse exemplo no Windows em vez de executá-lo em sua máquina virtual, é provável que vá se deparar com problemas por causa dos nomes dos arquivos. Por questões de simplicidade quanto às explicações, não utilizei o módulo *path* que faz parte do núcleo, o qual faz com que os paths dos diretórios sejam compatíveis entre diferentes plataformas. Por outro lado, eu o usei no código que está em nosso repositório no GitHub, portanto, se você estiver tendo problemas, dê uma olhada nos exemplos que se encontram lá.

No exemplo anterior, utilizamos *app.use* para criar um diretório de arquivos estáticos no servidor. Isso significa que qualquer solicitação enviada ao nosso servidor será resolvida inicialmente pelo diretório de arquivos estáticos (*client*) antes de ser enviada às nossas rotas. Quer dizer que, se tivermos um arquivo chamado *index.html* em nosso diretório *client*, e acessarmos *localhost:3000/index.html*, o conteúdo desse arquivo será retornado. Se o arquivo não existir, será feita uma verificação para saber se há alguma correspondência em relação às nossas rotas.

É nesse ponto que pode haver um pouco de confusão – se você tiver uma rota



de mesmo nome que um arquivo que estiver em seu diretório *client*, como o Express responderá? Ele tentará solucionar primeiro de acordo com o diretório *client*, portanto, se houver uma correspondência, suas rotas nem sequer serão consultadas. Tome cuidado para não ter rotas e arquivos de mesmo nome – é muito provável que não seja isso o que você pretende fazer.

Vamos experimentar executar esse código ao copiar uma de nossas aplicações do lado do cliente para o diretório *client*. Escolha qualquer uma das aplicações criadas anteriormente (ou, melhor ainda, exercite criando uma aplicação a partir do zero) e copie-a para o diretório *client* que está no diretório *Express*.

Agora podemos executar o nosso servidor a partir do computador guest usando `node server.js`. Pelo fato de a página HTML principal estar armazenada no arquivo *index.html*, quando abrirmos a página *localhost: 3000/index.html* em nosso navegador, deveremos vê-la – incluindo o CSS e o JavaScript!

Generalizações

Em geral, criaremos todas as nossas aplicações web de acordo com esse padrão. Teremos o código do navegador armazenado no diretório *client* e teremos um servidor Express definido em um arquivo chamado *server.js*. Também iremos importar e/ou criar módulos que suportem a parte de nosso programa referente ao lado do servidor.

A única tarefa que ainda não vimos é como estabelecer uma comunicação entre o cliente e o servidor. Como mencionado rapidamente no capítulo anterior, utilizaremos o AJAX para efetuar a comunicação e o formato JSON para representar nossas mensagens. O próximo exemplo irá demonstrar isso, além de tirar proveito dos módulos que realizam tarefas mais interessantes – por exemplo, conectar-se com a API do Twitter.

Contabilizando tuítes

Nesse exemplo, faremos a conexão com a API do Twitter e faremos

o stream de alguns dados para o nosso servidor. Vamos começar inicializando a nossa máquina virtual, caso ela ainda não esteja executando. Portanto, se você ainda não o fez, digite `vagrant up` a partir do diretório *app* que está no diretório *Chapter6*. A seguir, acesse o computador guest por meio do SSH. Se você estiver no Windows, essa operação poderá exigir o uso do PuTTY e, se você estiver no Mac OS ou no Linux, bastará digitar `vagrant ssh`.

No diretório *app*, crie um novo diretório chamado *Twitter*. Lembre-se de que, apesar de esse diretório estar sendo criado em nosso computador guest, ele estará espelhado no computador host, portanto esse diretório (o diretório *app* dentro do diretório *Projects*) também pode ser aberto no Sublime.

Obtendo as suas credenciais do Twitter

Para acessar a API de streaming do Twitter, inicialmente será necessário criar uma aplicação (<http://dev.twitter.com/>) e fazer o login usando as suas credenciais do Twitter. É claro que, se você não tiver uma conta no Twitter, será preciso criar uma, mas não se preocupe! Não farei você tuitar nada!

Depois de ter feito login na página de desenvolvedores do Twitter, clique no botão “Create a new application” (Crie uma nova aplicação), que está no canto superior à direita. Feito isso, você verá um formulário que tem a aparência mostrada na figura 6.8. Preencha todos os campos e submeta a sua solicitação.

A screenshot of a web browser showing the 'Create an application' page on the Twitter Developers website. The browser's address bar shows 'https://dev.twitter.com/apps/new'. The page has a dark header with the Twitter logo and navigation links like 'API Health', 'Blog', 'Discussions', and 'Documentation'. Below the header, there's a breadcrumb 'Home -> My applications' and a main heading 'Create an application'. The form itself is titled 'Application Details' and contains four input fields: 'Name', 'Description', 'Website', and 'Callback URL'. Each field has a small asterisk indicating it's required. Below the 'Website' field, there is a detailed note explaining its purpose for attribution and authorization. The 'Callback URL' field also has a note about its use for OAuth 1.0a applications.

Figura 6.8 – O formulário usado para criar a sua primeira aplicação para o Twitter!



Há um campo com um asterisco no formulário que exige que você adicione um site para a sua aplicação. É provável que você ainda não tenha um, portanto sinta-se à vontade para colocar um placeholder (gosto de usar ***www.example.com***). Não é preciso se preocupar com um URL para callback (Callback URL) nesse exemplo.

Se a sua solicitação for bem-sucedida, você verá uma página que inclui todas as informações de sua aplicação, incluindo o *Consumer key* e o *Consumer secret*. Para acessar a API de streaming, será necessário criar também um *Access token* (Token de acesso) para a sua aplicação. Você pode obter um ao clicar no botão na parte inferior que contém o rótulo “Create my access token” (Criar o meu token de acesso). Quando a página for recarregada, você deverá ter todas as credenciais necessárias para fazer o próximo exemplo funcionar.

Agora vamos criar um arquivo simples chamado *credentials.json* que será acessado a partir de nosso programa. Esse arquivo conterá um único objeto JSON que incluirá as suas credenciais do

Twitter. Substitua as strings a seguir pelas suas credenciais que estão na página inicial da aplicação no Twitter (não se esqueça de manter as aspas ao redor, pois elas são armazenadas na forma de strings):

```
{
  "consumer_key": "your app's API key here",
  "consumer_secret": "your app's API secret here",
  "access_token_key": "your app's Access token here",
  "access_token_secret": "your app's Access token secret here"
}
```



Sempre que você criar aplicações que exijam credenciais, será uma boa ideia deixá-las fora de seu repositório Git, somente para o caso de você querer compartilhar o seu código com outras pessoas. É por isso que é melhor armazenar suas credenciais em um arquivo separado. O Git permite ter um arquivo *.gitignore* especial que especifica os arquivos locais a serem mantidos fora de seu repositório.

Conectando-se com a API do Twitter

Começaremos pela instalação do módulo `ntwitter` usando o NPM:

```
agrant $ mkdir Twitter
agrant $ cd Twitter
agrant $ npm install ntwitter
agrant $ ls node_modules
express ntwitter
```

Agora criaremos outro arquivo chamado *tweet_counter.js* que incluirá o código que realmente acessa a API do Twitter. Observe que fazemos o `require` do módulo `ntwitter` da mesma maneira que o fizemos para o módulo `http` no exemplo do servidor. Quando fizermos o `require` do arquivo *credentials.json*, devemos informar ao Node o local em que esse arquivo poderá ser encontrado porque, de modo diferente do `express` e do `ntwitter`, não o instalamos com o NPM. É por esse motivo que prefixamos o arquivo com `./` – isso diz ao Node para procurar no diretório corrente:

```
var ntwitter = require("ntwitter"),
    credentials = require("./credentials.json"),
    twitter;
```

```

// configura o nosso objeto twitter
twitter = ntwitter(credentials);

// configura o nosso stream do twitter com três parâmetros
// separados por vírgulas
twitter.stream(
  // o primeiro parâmetro corresponde a uma string
  "statuses/filter",
  // o segundo parâmetro é um objeto que contém um array
  { "track": ["awesome", "cool", "rad", "gnarly", "groovy"] },
  // o terceiro parâmetro é nossa callback, a ser chamada quando a
  // stream for criada
  function(stream) {
    stream.on("data", function(tweet) {
      console.log(tweet.text);
    });
  }
);

```

Se tudo estiver definido corretamente, poderemos agora digitar **node tweet_counter** a partir de nossa máquina virtual e deveremos ver a janela do terminal sendo preenchida com tuítes que contenham as palavras especificadas! Se você estiver preocupado, achando que o seu computador está ficando louco, tenha calma e tecle Ctrl-C para interromper o stream.

O que está acontecendo aqui?

Vamos dar uma olhada no código novamente. Veremos diversos códigos familiares. Em primeiro lugar, declaramos três variáveis e importamos o módulo `ntwitter`, juntamente com o nosso arquivo `credentials.json`. A seguir, criamos uma variável chamada `twitter` e armazenamos o resultado da chamada da função `ntwitter`, que recebeu nossas credenciais como argumento. Esse código efetua a inicialização do objeto `twitter`, de modo que podemos dar início ao streaming.

Depois disso, definimos o stream por meio da chamada à função `stream` do objeto `twitter`. Essa função recebe três argumentos. O

primeiro corresponde a uma string que representa o tipo de stream – filtraremos os tuítes de acordo com uma lista de palavras. O segundo é um objeto que contém informações sobre as regras de filtragem (nesse caso, estamos apenas em busca da ocorrência de determinadas palavras – também podemos filtrar por localização, entre outros critérios). E, por fim, enviamos uma callback que será chamada quando o stream for criado.

O argumento `stream` propriamente dito é um objeto em relação ao qual podemos ficar prestando atenção em eventos (como um elemento do DOM na jQuery). O evento no qual estamos prestando atenção é “data”, e esse evento é disparado sempre que um novo tuíte chegar pelo stream. O que fazemos quando obtivermos um novo tuíte? Simplesmente o exibimos no console! Isso é tudo o que é necessário para efetuarmos o stream do Twitter em nossa janela do terminal!

Armazenando contadores

Em vez de somente exibir os tuítes no console, vamos manter um contador da quantidade de vezes em que vimos determinadas palavras. Isso significa que devemos ter uma variável separada para cada palavra. Como vamos manter diversas variáveis, todas elas representando algo semelhante, provavelmente faz sentido utilizar um objeto que represente os contadores. O atributo associado a cada contador será a palavra, e o valor do atributo será o número de vezes que essa palavra ocorre.

Podemos modificar o início de nosso código de modo a declarar e criar um objeto desse tipo:

```
var ntwitter = require("ntwitter"),
    credentials = require("./credentials.json"),
    twitter,
    counts = {};
counts.awesome = 0;
counts.cool = 0;
counts.rad = 0;
```



```
counts.gnarly = 0;  
counts.groovy = 0;
```

Esse código inicializa todos os valores com 0, o que significa que ainda não vimos nenhuma das palavras.

Utilizando a função `indexOf` para encontrar palavras

Quando obtemos dados a partir do stream do Twitter, devemos verificar se a propriedade `text` do objeto `tweet` contém a palavra que estamos procurando. No capítulo anterior, vimos que os arrays possuem um método chamado `indexOf` que verifica se um array contém um objeto. Acontece que as strings apresentam exatamente a mesma funcionalidade! A função `indexOf` de um objeto string retorna o índice da primeira ocorrência de uma substring dentro de uma string, ou `-1` se a substring não estiver presente. Por exemplo, experimente digitar o código a seguir no JavaScript console do Chrome:

```
var tweetText = "This is a tweet! It has lots of words but less than 140 characters."  
  
// Usamos indexOf para ver se a string contém determinadas palavras  
tweetText.indexOf("tweet");  
//=> 10  
  
tweetText.indexOf("This");  
//=> 0  
  
tweetText.indexOf("words");  
//=> 32  
  
tweetText.indexOf("char");  
//=> 56  
  
tweetText.indexOf("hello");  
//=> -1  
  
// observe que indexOf diferencia letras maiúsculas de minúsculas  
// (é case-sensitive)  
tweetText.indexOf("Tweet");  
//=> -1
```

Você verá que `indexOf` irá pesquisar o tuíte em busca de uma determinada palavra e, se ela estiver presente na forma de uma substring, o resultado será maior que `-1`. Isso nos permite remover

a instrução `console.log` e modificar o nosso código para que faça algo como:

```
function(stream) {
  stream.on("data", function(tweet) {
    if (tweet.text.indexOf("awesome") > -1) {
      // incrementa o contador de awesome
      counts.awesome = counts.awesome + 1;
    }
  });
}
```

Utilizando a função `setInterval` para agendar tarefas

Sem a instrução `console.log`, como podemos saber se o nosso código está funcionando? Uma abordagem consiste em exibir os contadores a intervalos de alguns segundos. Isso é um pouco mais administrável do que exibir todos os tuítes que virmos. Para isso, podemos adicionar uma chamada à função `setInterval` no final de nosso código de streaming:

```
// exibe o contador de awesome a cada 3 segundos
setInterval(function () {
  console.log("awesome: " + counts.awesome);
}, 3000);
```

Do mesmo modo que a função `setTimeout` utilizada anteriormente, a função `setInterval` agenda uma função para que seja chamada mais tarde. A diferença é que a chamada à função será repetida sempre que houver decorrido a quantidade especificada de milissegundos. Sendo assim, nesse caso, o programa efetuará o log do contador `awesome` a cada três segundos.

Se estivermos interessados somente em contabilizar o número de vezes que a palavra `awesome` ocorre, o nosso arquivo *twitter.js* completo se parecerá com:

```
var ntwitter = require("ntwitter"),
    credentials = require("./credentials.json"),
    twitter,
    counts = {};
```

```

// cria o nosso objeto twitter
twitter = ntwitter(credentials);

// inicializa os nossos contadores
counts.awesome = 0;

twitter.stream(
  "statuses/filter",
  { "track": ["awesome", "cool", "rad", "gnarly", "groovy"] },
  function(stream) {
    stream.on("data", function(tweet) {
      if (tweet.text.indexOf("awesome") > -1) {
        // incrementa o contador de awesome
        counts.awesome = counts.awesome + 1;
      }
    });
  }
);

// exibe o contador de awesome a cada 3 segundos
setInterval(function () {
  console.log("awesome: " + counts.awesome);
}, 3000);

```

Excute esse código em sua máquina virtual digitando `node tweet_counter.js` para vê-lo em ação.

Modularizando o nosso contador de tuítes

Mesmo que seja ótimo poder ver os tuítes aparecerem na janela do terminal, seria melhor se pudéssemos, de alguma maneira, associar os contadores de tuítes a um servidor HTTP para que possamos exibi-los em nosso navegador web. Há duas maneiras possíveis de fazer isso. A primeira possibilidade envolve a criação de um servidor que gere a página dentro de nosso código de contagem de tuítes. Essa é uma solução perfeitamente razoável para um iniciante.

A segunda solução consiste em converter o código de contagem de tuítes em um módulo e, em seguida, importá-lo em um programa que tenha um servidor HTTP (como em nosso exemplo original de servidor). Nesse caso, a segunda solução é melhor porque permite tornar o nosso código de contagem de tuítes reutilizável para outros

projetos. Com efeito, eu diria que essa solução está de acordo com o que está surgindo como um dos princípios centrais da filosofia do Node.js: criar módulos pequenos que executem uma tarefa e a executem muito bem.

Como podemos converter o nosso código de contagem de tuítes em um módulo? Na realidade, isso é bem simples. Todo módulo possui uma variável especial chamada `module.exports`, que armazena tudo o que queremos expor ao mundo externo. Nesse caso, queremos expor o objeto `counts`, que será atualizado pelo módulo `tweet_counter` enquanto o nosso programa estiver executando. No final do código que está em *tweet_counter.js*, podemos simplesmente acrescentar a linha a seguir para exportar o objeto:

```
module.exports = counts;
```



Nesse exemplo, exportamos um objeto que contém vários inteiros, porém qualquer tipo de objeto JavaScript pode ser exportado, inclusive as funções. Na seção prática no final deste capítulo, você terá a oportunidade de tornar esse módulo mais utilizável ao efetuar uma generalização de modo que ele exporte uma função em vez de exportar um objeto.

Importando o nosso módulo no Express

Lembre-se de que o nosso servidor Express simples tinha o seguinte aspecto:

```
var express = require("express"),
    http = require("http"),
    app = express();

// configura a aplicação para que use o diretório client para arquivos estáticos
app.use(express.static(__dirname + "/client"));

// cria o servidor e faz ele ficar ouvindo
http.createServer(app).listen(3000);

// define as rotas
app.get("/", function (req, res) {
  res.send("Hello World!");
});
```

Podemos copiar esse arquivo (*server.js*) para o nosso diretório *Twitter* (o mesmo que contém *tweet_counter.js*) e, em seguida,

modificá-lo para que ele importe o nosso módulo de contagem de tuítes e utilize os contadores que exportamos. Também retornaremos os contadores ao navegador em formato JSON:

```
var express = require("express"),
    http = require("http"),
    tweetCounts = require("../tweet_counter.js"),
    app = express();

// configura a aplicação para que use o diretório client para arquivos estáticos
app.use(express.static(__dirname + "/client"));

// cria o servidor e faz ele ficar ouvindo
http.createServer(app).listen(3000);

// define as rotas
app.get("/counts.json", function (req, res) {
    // res.json retorna o objeto inteiro na forma de um arquivo JSON
    res.json(tweetCounts);
});
```

Se fizermos login em nosso computador guest, poderemos ir até o diretório *app/Twitter* e executar o servidor usando `node server.js`. Ao fazermos isso, veremos que o terminal exibirá os contadores a cada três segundos, porém agora podemos abrir o nosso navegador e conectá-lo a *localhost:3000/counts.json*; nesse caso, devemos ver o objeto JSON sendo retornado para nós!



Tenha em mente que a rota *counts.json* não é realmente um arquivo. Você pode pensar nela como um arquivo virtual criado pela servidor de sua aplicação enquanto ela estiver executando. Ao configurar a rota usando a função `app.get`, você estará dizendo ao servidor que, quando um cliente solicitar esse arquivo em particular, ele deve ser criado usando o objeto `tweetCounts` armazenado internamente.

Como podemos tirar proveito do objeto JSON no código de nosso cliente? Usando o AJAX, é claro!

Criando um cliente

Em nosso diretório *Twitter*, vamos criar o esqueleto de um cliente que inclua os nossos arquivos *index.html*, *styles/style.css* e *javascripts/app.js* usuais com os quais estamos acostumados a

trabalhar para programar o lado do cliente. Como feito anteriormente, nós os colocaremos no diretório *client* de nosso diretório *Twitter*. O servidor ainda pode ser executado a partir do diretório *app/Twitter* de seu computador *guest*:

```
vagrant $ node server.js
```

Se você criou devidamente o arquivo *javascripts/app.js* de modo a exibir `hello world` e acessar *localhost:3000/index.html* em seu navegador, você deverá ver a mensagem `hello world` ser exibida no JavaScript console, como fizemos anteriormente.

Se isso funcionar, vamos modificar o arquivo *app.js* do lado do cliente para acessar o objeto `counts.json` por meio de uma chamada à função `getJSON` da jQuery:

```
var main = function () {  
  "use strict";  
  
  $.getJSON("/counts.json", function (wordCounts) {  
    // Agora, "wordCounts" será o objeto  
    // retornado pela rota counts.json  
    // definida no Express  
    console.log(wordCounts);  
  });  
}  
  
$(document).ready(main);
```

Nossa chamada a `getJSON` está efetuando a conexão com a rota `counts.json` que definimos no Express, a qual retornará os contadores. Se tudo estiver corretamente definido, esse código deverá exibir o objeto `counts` no console. Podemos explorar o objeto no console ao detalhar o seu conteúdo usando as setas suspensas. E se recarregarmos a página, poderemos ver os valores atualizados!

Agora podemos modificar o nosso código de modo a inserir os contadores no DOM usando os recursos de manipulação do DOM da jQuery. Aqui está uma maneira simples de fazer isso:

```
var main = function () {  
  "use strict";  
  
  var insertCountsIntoDOM = function (counts) {
```

```

    // insira aqui o seu código de manipulação do DOM
};

// observe que insertCountsIntoDOM é chamado com
// o parâmetro counts quando getJSON retornar
$.getJSON("counts.json", insertCountsIntoDOM);
}

$(document).ready(main);

```

Por que criar uma solução dessa maneira? Porque isso, na realidade, nos permite fazer algo que foi feito no código do lado do servidor – podemos utilizar a função `setInterval` para atualizar a página dinamicamente!

```

var main = function () {
    "use strict";

    var insertCountsIntoDOM = function (counts) {
        // insira aqui o seu código de manipulação do DOM
    };

    // verifica o valor dos contadores a cada 5 segundos,
    // e insere a versão atualizada no DOM
    setInterval(function () {
        $.getJSON("counts.json", insertCountsIntoDOM);
    }, 5000);
}

$(document).ready(main);

```

Desafio você a trabalhar com esse código e realmente tentar personalizá-lo para entender o seu funcionamento. Esse é o nosso primeiro exemplo de uma aplicação web completa – que tem tanto um componente cliente quanto um componente servidor, em que ambos se comunicam por meio de AJAX.

Criando um servidor para o Amazeriffic

Se você trabalhou até o último exemplo do capítulo anterior, deve ter conseguido fazer o Amazeriffic cuidar de uma lista de tarefas a serem feitas (to-do list), servida por um arquivo JSON, em vez de tê-la fixa no código de seu programa. Podemos partir desse código e, com algumas modificações pequenas, manter a lista de tarefas

armazenada no servidor. Nesse exemplo, você deve começar pelo Git para que as alterações possam ser controladas à medida que o seu trabalho for feito.

Criando os nossos diretórios

Começaremos pela criação de um diretório dentro de nosso diretório *Chapter6/app*, que chamaremos de *Amazeriffic*. Lembre-se de que não importa se vamos fazer isso no computador host ou no guest. Nesse diretório, criaremos um diretório para armazenar o código do cliente. Vamos chamar esse diretório de *client* e copiar o conteúdo de nosso último exemplo do Amazeriffic do capítulo 5.

Uma diferença importante entre esse exemplo e o exemplo do capítulo 5 é que iremos armazenar a nossa lista de tarefas no servidor. Isso significa que não precisaremos mais do arquivo *todo.json* que armazenava o estado inicial da lista de tarefas. No entanto, como queremos copiar o conteúdo desse arquivo para o nosso servidor em algum momento, será útil preservá-lo com um nome diferente. Para renomear um arquivo a partir da linha de comando, basta utilizar o comando `mv`, que quer dizer “move” (mover):

```
hostname $ mv client/todos.json client/todos.OLD.json
```

Inicializando um repositório Git

A seguir, iremos inicializar o nosso repositório Git para incluir os arquivos copiados. Lembre-se de que interagimos com o Git a partir de nosso computador host. A partir daqui, deixarei que você decida quais são os momentos apropriados para efetuar os commits:

```
hostname $ git init
Initialized empty Git repository ...
```

Depois disso, verificaremos o status de nosso repositório e efetuaremos o commit dos arquivos do lado do cliente, como fizemos nos capítulos anteriores. Agora podemos alternar para o nosso computador guest a fim de instalar o módulo `Express`. Vá para

o diretório *Amazeriffic* e digite o comando `npm` a seguir:

```
vagrant $ pwd
/home/vagrant/app/Amazeriffic
vagrant $ npm install express@3
```

Criando o servidor

No ponto em que estamos agora, nossa aplicação funciona perfeitamente com a suposição de que o usuário nunca mudará para outro navegador ou para outro computador e que ele nunca recarregará a página.

Infelizmente, se o usuário realizar uma dessas tarefas, ele perderá todos os itens da lista de tarefas criados e retornará à estaca zero.

Resolveremos esse problema se armazenarmos a nossa lista de tarefas no servidor e inicializarmos o nosso cliente com os dados armazenados nesse local. Para começar, criaremos uma lista de tarefas em uma variável no servidor e definiremos uma rota JSON que a retornará:

```
var express = require("express"),
    http = require("http"),
    app = express(),
    todos = {
      // cria a lista de tarefas aqui copiando o conteúdo de todos.OLD.json
    };
app.use(express.static(__dirname + "/client"));
http.createServer(app).listen(3000);
// Esta rota tomará o lugar de nosso arquivo todos.json do exemplo do cap 5
app.get("/todos.json", function (req, res) {
  res.json(todos);
});
```

Executando o servidor

Certifique-se de que você está no terminal de seu computador guest e execute *server.js* com o Node:

```
vagrant $ node server.js
```

Depois que a aplicação estiver executando, devemos ser capazes de acessá-la ao abrir o Chrome e inserir *localhost:3000/index.html* na barra de endereço! Com efeito, devemos simplesmente digitar *localhost:3000* na barra de endereço e o servidor irá automaticamente efetuar o roteamento para *index.html* que, normalmente, é considerada a página default.

Nesse ponto, tudo deverá funcionar exatamente como antes, porém ainda não resolvemos nenhum de nossos problemas. Isso porque a nossa lista de tarefas está sendo entregue pelo servidor, porém o nosso cliente, por ora, ainda não está enviando nenhuma atualização. E ainda não aprendemos a enviar informações para o servidor quando algo é atualizado no cliente.

Enviando informações ao servidor

Até agora, fizemos somente os nosso programas clientes receberem dados do servidor. Isso significa que nossas aplicações suportam uma comunicação de uma só via entre cliente e servidor. Para isso, contamos com a função `getJSON` da jQuery. O fato é que a jQuery tem uma função que permite, de modo igualmente fácil, enviar dados JSON ao servidor, porém o nosso servidor deve estar preparado para aceitá-los e fazer algo com esses dados.

O processo de enviar dados do cliente para o servidor por meio do protocolo HTTP chama-se *post* (postar). Podemos começar pela criação de uma rota `post` em nosso servidor Express. O exemplo a seguir define uma rota para `post`, que simplesmente exibe uma string no terminal de nosso servidor:

```
app.post("/todos", function (req, res) {  
  console.log("data has been posted to the server!");  
  
  // envia de volta um objeto simples  
  res.json({"message": "You posted to the server!"});  
});
```

Essa rota é muito semelhante às nossas rotas `get`, exceto pelo fato de termos substituído a chamada à função `get` por uma chamada à

função `post`. Na prática, a diferença é que não podemos simplesmente acessar a rota utilizando o nosso navegador, como fizemos com as rotas `get`. Em vez disso, devemos modificar o JavaScript do lado do cliente para que efetue o `post` para a rota de modo que a mensagem possa ser vista. Faremos isso quando o usuário clicar no botão `Add` (Adicionar). Modifique o código do cliente para que tenha o aspecto a seguir:

```
// este código está dentro de uma função main
// em que toDoObjects foi previamente definido

$button.on("click", function () {
    var description = $input.val(),
        tags = $tagInput.val().split(",");

    toDoObjects.push({"description":description, "tags":tags});

    // neste ponto, faremos um post rápido para a nossa rota "todos"
    $.post("todos", {}, function (response) {
        // esta callback será chamada quando o servidor responder
        console.log("We posted and the server responded!");
        console.log(response);
    });

    // atualiza toDos
    toDos = toDoObjects.map(function (toDo) {
        return toDo.description;
    });

    $input.val("");
    $tagInput.val("");
});
```

O primeiro argumento da função `$.post` é a rota para a qual queremos efetuar o `post`, o segundo corresponde aos dados (representados na forma de um objeto) que queremos enviar e o terceiro é a callback para a resposta do servidor. Nesse ponto, o nosso código deve funcionar exatamente como antes, exceto pelo fato de que, ao clicar no botão `Add`, veremos o servidor exibir uma mensagem em seu console e o cliente exibirá a resposta do servidor.

Sendo assim, embora estejamos a meio caminho, ainda não enviamos realmente um item da lista de tarefas ao servidor. Como já

mençãoi, o segundo argumento da função `post` corresponde ao objeto enviado ao servidor. Acontece que é bastante fácil substituir o objeto vazio pelo nosso objeto novo referente à lista de tarefas, porém é necessário um pouco mais de trabalho no servidor para tornar esse objeto utilizável.

A maneira mais fácil de fazer isso é por meio do uso do plug-in *urlencoded* do Express, que transformará o JSON enviado pela jQuery em um objeto JavaScript que poderá ser usado pelo servidor:

```
var express = require("express"),
    http = require("http"),
    app = express()
    todos = {
      // ...
    };

app.use(express.static(__dirname + "/client"));

// diz ao Express para efetuar o parse dos objetos JSON de entrada
app.use(express.urlencoded());

app.post("/todos", function (req, res) {
  // o objeto agora está armazenado em req.body
  var newToDo = req.body;

  console.log(newToDo);

  todos.push(newToDo);

  // envia de volta um objeto simples
  res.json({"message": "You posted to the server!"});
});
```

Essa iteração do servidor faz o objeto ser exibido no console do servidor quando ele é enviado. Depois disso, o novo item da lista de tarefas será acrescentado à lista. Agora, uma pequena modificação em nossa aplicação cliente fará com que o novo item da lista de tarefas seja enviado:

```
$button.on("click", function () {
  var description = $input.val(),
      tags = $tagInput.val().split(", "),
      // cria o novo item da lista de tarefas
      newToDo = {"description": description, "tags": tags};
```

```
$.post("todos", newToDo, function (result) {
  console.log(result);

  // vamos esperar para adicionar o novo objeto
  // no cliente até que o servidor tenha retornado
  toDoObjects.push(newToDo);

  // atualiza toDos
  toDos = toDoObjects.map(function (toDo) {
    return toDo.description;
  });

  $input.val("");
  $tagInput.val("");
});
});
```

Esse código fará o novo item da lista de tarefas ser enviado ao servidor e a mensagem de resposta do servidor será exibida no JavaScript console. Depois disso, o novo objeto `toDo` será adicionado à lista de tarefas do cliente.

A essa altura, podemos abrir a nossa aplicação nas janelas de dois navegadores diferentes, adicionar itens à lista de tarefas em uma das janelas do navegador e recarregar a página em outra; então veremos o conteúdo ser atualizado!

Resumo

Neste capítulo, aprendemos o básico sobre a programação do lado do servidor usando o Node.js. O Node.js é uma plataforma que permite criar servidores na linguagem JavaScript. Ele é comumente usado para criar servidores HTTP que se comunicam com nossas aplicações clientes executadas no navegador.

Programas Node.js normalmente são constituídos de vários módulos individuais. Podemos criar os nossos próprios módulos ao implementar uma porção de código e, em seguida, associar as partes que queremos expor ao módulo `module.exports`. Depois disso, podemos importá-lo em outro programa utilizando a função `require`.

O NPM corresponde ao *Node.js Package Manager*; ele nos permite

instalar módulos do Node.js que não foram implementados por nós e que não estão incluídos na distribuição do Node.js. Por exemplo, o módulo `Express` foi instalado por meio do NPM. O `Express` é um wrapper que encapsula o módulo `http` do Node.js; esse módulo atende várias de nossas expectativas comuns em relação a um servidor HTTP. Especificamente, ele permite servir facilmente arquivos HTML, CSS e JavaScript do lado do cliente. Esse módulo também permite definir rotas personalizadas, associadas a comportamentos do servidor.

Usamos o VirtualBox e o Vagrant para configurar um ambiente básico. Essas ferramentas não são essenciais para entender o funcionamento das aplicações web, porém elas ajudam a impulsionar o processo de desenvolvimento. Além do mais, elas criam uma separação clara entre o cliente (o navegador sendo executado em nosso computador host) e o servidor (o programa Node.js sendo executado no computador guest).

Por fim, aprendemos a configurar uma solicitação HTTP post usando a jQuery e a fazer com que o nosso servidor responda às solicitações HTTP post por meio do `Express`.

Práticas e leituras adicionais

Instalando o Node.js localmente

Neste capítulo, instalamos o Node em uma máquina virtual usando o Vagrant. Um dos motivos principais para fazer isso é que os scripts do Vagrant também instalam softwares que utilizaremos nos capítulos 7 e 8. Acontece, porém, que instalar o Node.js em seu computador local é muito, muito fácil, e todos os exemplos deste capítulo devem funcionar em seu computador local com um mínimo de alterações. Se acessar <http://nodejs.org/download>, você deverá encontrar pacotes de instalação para a sua plataforma.

Tentar fazer isso é uma boa ideia. Se conseguir fazer a instalação em seu computador host, você poderá instalar globalmente outras

ferramentas que poderão ser úteis, incluindo o JSHint, o CSS Lint e uma ferramenta de validação de HTML5.

O JSHint e o CSS Lint usando o NPM

No capítulo 4, aprendemos a respeito do JSLint – uma ferramenta de controle de qualidade de código que nos informa se o nosso código está violando determinadas práticas do JavaScript consideradas boas. Utilizamos a ferramenta online disponível em <http://jshint.com>, que exigia que recortássemos e colássemos o nosso código no navegador. Se isso ainda não se tornou um problema para você, certamente o será à medida que a sua base de código aumentar.

O fato é que, quando temos o Node.js instalado em nosso computador local, podemos usar o NPM para instalar outras ferramentas de linha de comando que podem vir a ser práticas, por exemplo, o JSHint (<http://www.jshint.com/>). O JSHint é muito semelhante ao JSLint e pode ser usado para verificar o seu código. Vá até a linha de comando e instale-o usando o NPM. Utilizamos a opção `-g` para informar ao NPM que queremos instalar esse pacote globalmente. Isso torna o `jshint` disponível como uma aplicação-padrão de linha de comando, em vez de instalá-lo como uma biblioteca no diretório `node_modules` local:

```
hostname $ sudo npm install jshint -g
```



No Mac OS e no Linux, devemos utilizar `sudo` para instalar pacotes globalmente. Isso não é necessário no Windows.

Feito isso, podemos executar o JSHint diretamente em nossos arquivos. Considere o arquivo a seguir chamado `test.js`:

```
var main = function () {  
    console.log("hello world");  
}  
$(document).ready(main);
```

É claro que esse código irá executar perfeitamente no navegador, porém há um ponto e vírgula faltando após a definição da função

main. O JSHint nos informará a esse respeito:

```
hostname $ jshint test.js
test.js: line 3, col 2, Missing semicolon.
1 error
```

Se o nosso código estiver correto, o JSHint não mostrará nada. Se adicionarmos o ponto e vírgula que está faltando e executarmos o JSHint, não veremos nenhuma resposta:

```
hostname $ jshint test.js
hostname $
```

Ter o JSHint instalado em nosso computador facilita verificar a qualidade de nosso código tanto do lado do servidor quanto do lado do cliente, portanto não há desculpas para não fazê-lo! Você também perceberá que o JSHint é muito mais flexível, pois permite que suas opções sejam facilmente definidas a partir da linha de comando. Consulte a documentação em <http://jshint.com/docs>.

De modo semelhante, o NPM permite instalar e executar o CSS Lint a partir da linha de comando, o que torna muito mais fácil conferir o nosso CSS em busca de erros e de práticas inadequadas:

```
hostname $ sudo npm install csslint -g
```

Há diversas opções para as ferramentas de validação de HTML e para os linters também. Se você se vir trabalhando muito mais com o HTML, encontre uma que seja adequada às suas necessidades.

Generalizando o nosso código de contagem de tuítes

O nosso contador de tuítes monitora um array de palavras no Twitter. No entanto, em nosso código, tivemos de listar as palavras em vários locais diferentes. Por exemplo, no início do arquivo *tweet_counter.js*, vemos o seguinte:

```
counts.awesome = 0;
counts.cool = 0;
counts.rad = 0;
counts.gnarly = 0;
```



```
counts.groovy = 0;
```

E, posteriormente, vimos que informamos o módulo `ntwitter` para que monitorasse as seguintes palavras:

```
{ "track": ["awesome", "cool", "rad", "gnarly", "groovy"] },
```

Se você já deixou esse código genérico, é possível que tenha feito algo semelhante para incrementar os seus contadores:

```
if (tweet.text.indexOf("awesome") > -1) {  
  // incrementa o contador de awesome  
  counts.awesome = counts.awesome + 1;  
}  
  
if (tweet.text.indexOf("cool") > -1) {  
  // incrementa o contador de cool  
  counts.cool = counts.cool + 1;  
}
```

Por que isso é um problema? Se quisermos adicionar ou remover palavras de nosso código, será preciso modificá-lo em três lugares! Como isso pode ser melhorado? Em primeiro lugar, podemos definir o array em um só local: no início de nosso módulo `tweet_counter`:

```
var trackedWords = ["awesome", "cool", "rad", "gnarly", "groovy"];
```

Isso nos permite utilizar essa variável quando criarmos o código para a contagem de tuítes:

```
{ "track": trackedWords };
```

Para aperfeiçoar o restante do código, utilizaremos um recurso incrível dos objetos JavaScript: a capacidade de usar um objeto como uma estrutura de dados para *mapeamento*, em que os atributos são strings. O fato é que as duas abordagens a seguir para indexar um objeto são equivalentes:

```
// este código acessa o contador de awesome utilizando o operador ponto  
counts.awesome = 0;  
counts.awesome = counts.awesome + 1;  
  
// este código acessa o contador de awesome utilizando uma string  
counts["awesome"] = 0;  
counts["awesome"] = counts["awesome"] + 1;
```

Geralmente, é uma boa ideia contar com o operador ponto porque

os programadores JavaScript tendem a achar essa opção mais legível (e o JSLint reclamará se você não o fizer). Porém a abordagem que utiliza uma string entre colchetes tem como vantagem o fato de podermos usar variáveis para acessar os valores:

```
var word = "awesome";  
counts[word] = 0;  
counts[word] = counts[word] + 1;
```

Você percebe aonde quero chegar com isso? Aqui está uma solução para inicializar o nosso objeto contadores, que depende somente do array:

```
// cria um objeto vazio  
var counts = {};  
trackedWords.forEach(function (word) {  
    counts[word] = 0;  
});
```

Esse código percorre todas as palavras e inicializa o contador com 0. De modo semelhante, podemos atualizar o código que verifica se o tuíte contém uma palavra utilizando um laço `forEach`! Feito isso, podemos simplesmente adicionar ou remover uma palavra de nosso array inicial e toda a nossa aplicação estará atualizada!

De modo geral, a manutenção de nosso módulo torna-se um pouco mais fácil dessa maneira, mas podemos melhorar mais ainda. Suponha que, em vez de exportar os contadores de palavras específicas sendo monitoradas, nós permitíssemos que o consumidor de nosso módulo decidisse quais palavras devem ser monitoradas. Por exemplo, um consumidor de nosso módulo poderá preferir um uso deste tipo:

```
var tweetCounter = require("./tweet_counter.js"),  
    counts;  
  
// este código inicia o nosso tweetCounter com as palavras  
// especificadas, em vez de usar a lista de palavras previamente definida  
counts = tweetCounter(["hello", "world"]);
```

Podemos fazer isso ao exportar uma função em vez de exportar o

objeto counts:

```
var setUpTweetCounter = function (words) {  
  // define o objeto counts e o stream do ntwitter  
  // usando o array de palavras  
  
  // ...  
  
  // no final, retorna counts  
  return counts;  
}  
  
module.exports = setUpTweetCounter;
```

Esse código torna o nosso módulo muito mais flexível do ponto de vista do consumidor. Um programa que utilize o nosso módulo poderá decidir quais palavras devem ser monitoradas sem nem mesmo ver o nosso código. Implementar isso é um excelente exercício, portanto experimente fazê-lo!

API para pôquer

Aqui está um projeto simples que permitirá exercitar a criação de uma API usando o Express. Crie uma aplicação Express que responda a uma única rota de post: */hand*. A rota deve aceitar um objeto que represente uma mão de pôquer e deve responder com um objeto JSON que especifica a melhor mão presente. Por exemplo, se o objeto a seguir for postado:

```
[  
  { "rank": "two", "suit": "spades" },  
  { "rank": "four", "suit": "hearts" },  
  { "rank": "two", "suit": "clubs" },  
  { "rank": "king", "suit": "spades" },  
  { "rank": "eight", "suit": "diamonds" }  
]
```

Nossa API deve responder da maneira apresentada a seguir, em que usamos a referência `null` (introduzida no capítulo 5) para representar “nenhum objeto”:

```
{  
  "handString": "Pair",  
  "error": null  
}
```

```
}
```

Em outras palavras, nesse exemplo, enviamos uma mão válida (cinco cartas) e o valor mais alto da mão é um par. Definimos a propriedade `error` com `null` para indicar que não houve erros. De modo alternativo, se alguém enviar uma mão inválida (por exemplo, se ela contiver valores inválidos ou cartas demais), definiremos `handString` com `null` e enviaremos uma string de erro de volta:

```
{
  "handString": null,
  "error": "Invalid Card Hand!"
}
```

Para facilitar, vamos empacotar as funções que criamos no capítulo 5 na forma de um módulo Node.js. Para usá-lo em nossa aplicação, devemos importá-lo em nosso servidor Express deste modo:

```
var poker = require("./poker.js");
```

Em seguida, utilizamos as funções desta maneira:

```
var hand = [
  { "rank": "two", "suit": "spades" },
  { "rank": "four", "suit": "hearts" },
  { "rank": "two", "suit": "clubs" },
  { "rank": "king", "suit": "spades" },
  { "rank": "eight", "suit": "diamonds" }
]

var hasPair = poker.containsPair(hand);
// hasPair será verdadeiro agora
```

Como podemos fazer isso? Criaremos um objeto `poker` com várias funções na definição de nosso módulo:

```
var poker = {};

poker.containsPair = function (hand) {
  // ... define a função
}

poker.containsThreeOfAKind = function (hand) {
  // ... define a função
}

module.exports = poker;
```

Nosso módulo `poker` terá diversas funções internas que não serão incluídas em nosso objeto exportado. Por exemplo, incluiremos a função `containsNTimes` em nosso módulo, porém não iremos exportá-la:

```
var poker = {},
containsNTimes; // declara a função

// neste ponto, definimos nossas funções 'privadas';
// não iremos adicioná-las ao objeto poker
containsNTimes = function (array, item, n) {
  // ... define a função
};

poker.containsPair = function (hand) {
  // ... usa containsNTimes aqui
}

poker.containsThreeOfAKind = function (hand) {
  // ... usa containsNTimes aqui
}

// exporta somente as funções relacionadas a poker
module.exports = poker;
```

Queremos incluir mais algumas funções que validam se uma mão é realmente uma mão de pôquer, e podemos até mesmo incluir uma função que retorne um objeto conforme especificado pela API. Então a callback de nossa rota será constituída essencialmente de uma única linha:

```
app.post("/hand", function (req, res) {
  var result = poker.getHand(req.body.hand);
  res.json(result);
});
```

Para fazer esse código funcionar, será preciso implementar uma aplicação cliente que crie algumas mãos de pôquer e as envie ao servidor usando a jQuery. Se conseguir fazer com que isso aconteça, você realmente estará começando a entender como tudo funciona!

¹ N.T.: Formato-padrão para um ambiente criado pelo Vagrant.

CAPÍTULO 7

Armazenamento de dados

Nos capítulos anteriores, aprendemos a criar servidores básicos usando o Node.js e a efetuar a comunicação entre o nosso cliente e o servidor por meio do AJAX. Uma das aplicações mais interessantes envolveu a monitoração do número de ocorrências de determinadas palavras no Twitter e a apresentação dos contadores no cliente.

Um dos principais problemas dessa aplicação é que todas as suas informações são armazenadas na memória do programa Node.js. Isso significa que, se encerrarmos o nosso processo servidor, os contadores de palavra desaparecerão com ele. Esse não é o comportamento que desejamos, pois, com frequência, será necessário encerrar o servidor a fim de atualizá-lo ou, mais comumente, o servidor se encerrará por conta própria em decorrência de algum bug. Quando qualquer uma dessas situações ocorrer, gostaríamos que todos os contadores obtidos até então permanecessem intactos.

Para resolver o problema, devemos usar algum tipo de aplicação para armazenamento de dados que execute independentemente de nosso programa. Neste capítulo, aprenderemos a respeito de duas abordagens diferentes para solucionar esse problema utilizando um sistema de armazenamento de dados NoSQL. Especificamente, estudaremos o Redis e o MongoDB e aprenderemos a integrá-los em nossas aplicações Node.js.

NoSQL versus SQL

Se você já estudou bancos de dados, provavelmente já viu o acrônimo SQL (às vezes, pronunciado como a palavra *sequel* em inglês), que quer dizer *Structured Query Language* (Linguagem de consulta estruturada). O SQL é uma linguagem utilizada para fazer perguntas a um banco de dados armazenado em formato *relacional*. Bancos de dados relacionais armazenam dados em células de tabelas, em que podemos facilmente fazer referências cruzadas das linhas dessas tabelas com outras tabelas. Por exemplo, um banco de dados relacional pode armazenar uma tabela de atores e de atrizes, além de uma tabela separada de filmes. Uma tabela relacional pode ser usada para mapear atores e atrizes aos filmes nos quais eles atuam.

Essa abordagem apresenta diversas vantagens, porém o motivo principal pelo qual ela é amplamente usada está no fato de que armazenar informações dessa maneira minimiza a redundância e, sendo assim, normalmente exige menos espaço em relação às abordagens alternativas. No passado, esse era um recurso importante, pois armazenar grandes volumes de dados era relativamente caro. Nos últimos anos, porém, o preço do armazenamento foi drasticamente reduzido, e minimizar a redundância não é mais tão importante quanto costumava ser.

Essa mudança fez os pesquisadores e os engenheiros repensarem suas suposições a respeito do armazenamento de dados e começaram a experimentar armazenamentos em formatos não relacionais. As novas soluções para armazenamento de dados, às vezes chamadas de armazenamento de dados *NoSQL*, exigem um compromisso: de vez em quando, informações redundantes são armazenadas em troca de mais facilidade de uso do ponto de vista da programação. Além do mais, algumas dessas soluções de armazenamento de dados foram concebidas com casos de uso específicos em mente, por exemplo, para aplicações em que a leitura de dados deve ser mais eficiente do que a escrita.

Redis

O Redis é um exemplo perfeito de uma solução de armazenamento de dados NoSQL. Ele foi concebido para proporcionar rapidez de acesso a dados utilizados regularmente. O Redis oferece esse aumento de velocidade em detrimento da confiabilidade, pois armazena os dados na memória em vez de armazená-los em disco. (T tecnicamente, o Redis grava os dados em disco periodicamente, porém, de modo geral, você pode pensar nele como uma solução de armazenamento de dados em memória.)

O Redis armazena informações em um formato *chave-valor* – pense nisso como algo semelhante ao modo como o JavaScript armazena propriedades de objetos. Por essa razão, o Redis permite que um desenvolvedor organize informações em estruturas de dados tradicionais (hashes, listas, conjuntos, etc.), o que passa a impressão de que ele é uma extensão natural para armazenamento de dados em nossos programas. A solução é perfeita para dados que devem ser acessados rapidamente ou para armazenar temporariamente dados que são acessados com frequência (o que chamamos de *caching*) a fim de melhorar o tempo de resposta de nossas aplicações.

Não iremos explorar esses casos de uso neste livro, porém tê-los em mente à medida que você prosseguir em sua jornada é uma boa ideia. O nosso uso do Redis será relativamente simples – queremos apenas armazenar os dados relativos aos contadores do Twitter de forma independente de nosso servidor Node.js. Para isso, utilizaremos uma chave para cada uma das palavras, e cada valor será um inteiro que representa o número de vezes que a palavra ocorrer. Antes de aprendermos a fazer isso no programa, aprenderemos a interagir com o Redis por meio da linha de comando.

Interagindo com o cliente de linha de

comando do Redis

Para manter uma separação clara em relação ao nosso trabalho anterior, vamos começar efetuando novamente uma clonagem do projeto `node-dev-bootstrap` para o nosso diretório *Projects*, como *Chapter7*. Se precisar relembrar de que maneira essa tarefa deve ser feita, dê uma olhada no capítulo 6.

Feito isso, entre no diretório, inicialize a sua máquina virtual e acesse o seu guest por meio do SSH. Como você está recriando a máquina a partir do zero, a tarefa consumirá um pouco de tempo:

```
hostname $ cd Projects/Chapter7
hostname $ vagrant up
...vagrant build stuff...
hostname $ vagrant ssh
```

Agora devemos estar logados em nossa máquina virtual, e o Redis já estará instalado e configurado. Podemos usar o comando `redis-cli` para iniciar o cliente Redis interativo:

```
vagrant $ redis-cli
redis 127.0.0.1:6379>
```

Armazenar dados no Redis é tão simples quanto utilizar o comando `set`. Neste caso, criaremos uma chave para `awesome` e definiremos o seu valor como 0:

```
redis 127.0.0.1:6379> set awesome 0
OK
```

Se tudo correr bem, o Redis deve responder com “OK”. Podemos conferir o valor da chave utilizando o comando `get`:

```
redis 127.0.0.1:6379> get awesome
"0"
```

Após termos o valor armazenado, podemos manipulá-lo de diversas maneiras. É claro que, provavelmente, estaremos mais interessados em incrementá-lo de 1 ao vermos a palavra sendo utilizada em um tuíte. Felizmente, o Redis tem um comando `incr` que faz exatamente isso:

```
redis 127.0.0.1:6379> incr awesome
```

```
(integer) 1
redis 127.0.0.1:6379> incr awesome
(integer) 2
redis 127.0.0.1:6379> get awesome
"2"
```

O comando `incr` adiciona 1 ao valor que está associado no momento à chave especificada (nesse caso, a chave é `awesome`). Para sair de nosso cliente Redis interativo, podemos digitar `exit`.

Esses três comandos (`set`, `get` e `incr`) são tudo o que precisamos conhecer para começar a armazenar os nossos contadores, mas o Redis oferece muito mais do que foi apresentado aqui. Para ter um “gostinho” de alguns de seus recursos, experimente usar o incrível tutorial interativo do Redis (<http://try.redis.io/>), que está sendo mostrado na figura 7.1.

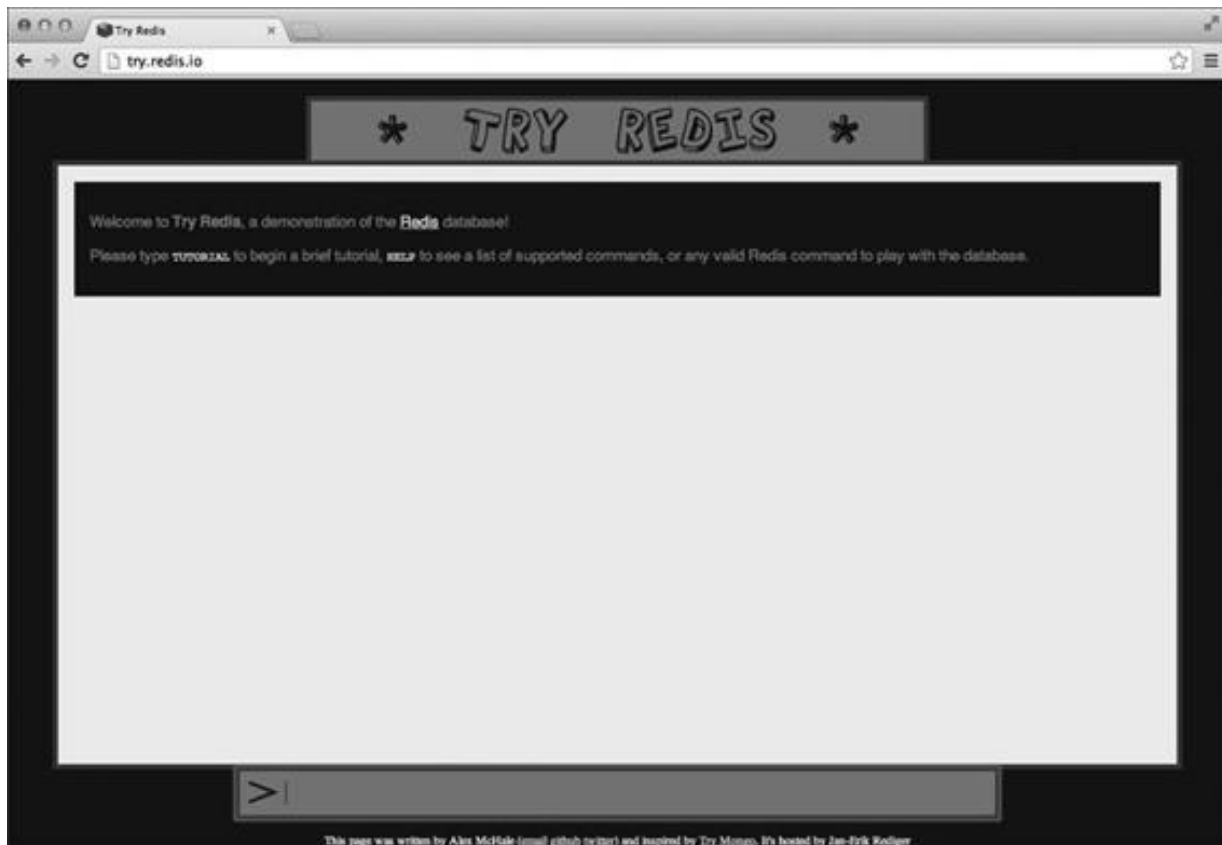


Figura 7.1 – A página inicial do tutorial interativo do Redis.

Instalando o módulo Redis por meio de um

arquivo package.json

A essa altura, sabemos criar uma chave para cada palavra e incrementar o valor interativamente, porém queremos fazer isso por meio de nossos programas Node.js.

Vamos trabalhar a partir de nosso projeto com o Twitter no último capítulo, portanto copie o nosso diretório *Twitter* que está no diretório *Chapter6/app* para o nosso diretório *Chapter7/app*. Para isso, você pode utilizar o comando `cp` a partir da linha de comando ou pode usar a sua interface GUI normal.

Para fazer com que o nosso programa Node.js se comunique com o Redis, devemos instalar o módulo `node-redis`. Anteriormente, utilizamos o comando `npm` a partir da linha de comando para instalar o módulo `ntwitter`. Isso faz sentido quando estamos começando, porém, à medida que prosseguirmos, será importante monitorar nossas dependências de maneira mais formal. Para isso, utilizaremos um arquivo especial chamado *package.json*, que controla diversos aspectos de nosso projeto, inclusive as dependências.

Em seu diretório *app*, crie um arquivo chamado *package.json* e copie o conteúdo a seguir:

```
{
  "name": "tutorial",
  "description": "a tutorial on using node, twitter, redis, and express",
  "version": "0.0.1",
  "dependencies": {
    "ntwitter": "0.5.x",
    "redis": "0.8.x"
  }
}
```



O NPM oferece uma abordagem interativa para criar um arquivo *package.json*, mas serão solicitadas muito mais informações além dessas. Se quiser experimentá-la, digite `npm init` no diretório de seu projeto.

Como você pode ver, esse arquivo especifica algumas características de nosso projeto, incluindo suas dependências

(como o `node-redis` é o cliente recomendado para o Node.js, ele é simplesmente referenciado como `redis` no NPM). Isso será utilizado no capítulo 8 para a implantação de nossa aplicação e de todas as suas dependências no Cloud Foundry. Por enquanto, porém, utilizar um arquivo *package.json* tem somente a vantagem de simplificar a instalação. Especificamente, podemos instalar agora todas as dependências ao digitar o comando a seguir em nosso computador guest:

```
vagrant $ cd app
vagrant $ npm install
```

Interagindo com o Redis em nosso código

Em seu computador host, utilize o seu editor de texto para abrir o arquivo *tweet_counter.js* que criamos no capítulo anterior. Esse arquivo deve estar no diretório *Twitter* de sua pasta *app*. Se você resolveu os exemplos e os problemas, o seu código provavelmente deve estar um pouco mais detalhado que o meu. De qualquer modo, siga as minhas orientações e modifique o seu código para que ele importe e utilize o módulo `redis`:

```
var ntwitter = require("ntwitter"),
    redis = require("redis"), // faz o require do módulo redis
    credentials = require("./credentials.json"),
    redisClient,
    twitter,
    // declara um objetos counts para armazenar os contadores
    counts = {};

twitter = ntwitter(credentials);

// cria um cliente para se conectar ao Redis
redisClient = redis.createClient();

// inicializa os nossos contadores
counts.awesome = 0;

twitter.stream(
  "statuses/filter",
  { track: ["awesome", "cool", "rad", "gnarly", "groovy"] },
  function(stream) {
```

```

stream.on("data", function(tweet) {
  if (tweet.text.indexOf("awesome") > -1) {
    // incrementa a chave no cliente
    redisClient.incr("awesome");
    counts.awesome = counts.awesome + 1;
  }
});
}
);

module.exports = counts;

```

Utilizamos a função `incr` exatamente como fizemos no cliente Redis interativo. Com efeito, podemos usar qualquer comando do Redis exatamente da mesma maneira – o cliente expõe uma função com o mesmo nome do comando.

Podemos executar esse código da maneira usual. Vamos executá-lo por alguns instantes – o código deverá registrar a quantidade de tuítes que contêm a palavra “awesome” no Redis e, se o código referente a `setInterval` foi mantido, poderemos vê-lo exibindo os valores periodicamente. Depois de termos visto alguns tuítes e que estivermos satisfeitos, poderemos verificar se os contadores estão sendo armazenados no Redis ao interromper o programa (usando `Ctrl-C`) e então efetuar a conexão novamente por meio do programa `redis-cli`. Estando nesse programa, podemos utilizar o comando `get` para verificar o valor armazenado para a chave `awesome`:

```

vagrant $ redis-cli
redis 127.0.0.1:6379> get awesome
(integer) "349"

```



Se, a qualquer momento, você quiser limpar os dados armazenados no Redis, digite **flushall** no prompt do `redis-cli`.

Inicializando os contadores a partir dos dados armazenados no Redis

A essa altura, estamos armazenando os dados enquanto o programa está executando, porém, ao reiniciar o nosso servidor,

continuamos a reinicializar o objeto `counts` com 0. Para solucionar totalmente o nosso problema, queremos que o nosso contador de tuítes seja inicializado com os dados que estão armazenados no Redis. Podemos utilizar o comando `get` para obter os valores antes de iniciarmos o stream. No entanto, como ocorre com quase tudo no Node, a função `get` é assíncrona, por isso temos de trabalhar com ela com certo cuidado:

```
var ntwitter = require("ntwitter"),
    redis = require("redis"), // faz o require do módulo redis
    credentials = require("./credentials.json"),
    redisClient,
    counts = {},
    twitter;

twitter = ntwitter(credentials);
redisClient = redis.createClient();

// a callback recebe dois argumentos
redisClient.get("awesome", function (err, awesomeCount) {
  // verifica para certificar-se de que não há nenhum erro
  if (err !== null) {
    console.log("ERROR: " + err);

    // sai da função
    return;
  }

  // inicializa o nosso contador com a versão inteira do valor armazenado
  // no Redis, ou com 0 caso o valor não esteja definido
  counts.awesome = parseInt(awesomeCount,10) || 0;

  twitter.stream(
    "statuses/filter",
    { track: ["awesome", "cool", "rad", "gnarly", "groovy"] },
    function(stream) {
      stream.on("data", function(tweet) {
        if (tweet.text.indexOf("awesome") > -1) {
          // incrementa a chave no cliente
          redisClient.incr("awesome");
          counts.awesome = counts.awesome + 1;
        }
      });
    }
  );
});
```

```
);  
});  
module.exports = counts;
```

Você perceberá que a callback para `get` aceita dois parâmetros: `err` e `awesomeCount`. O parâmetro `err` representa uma condição de erro e será configurado com um objeto erro se houver algum tipo de problema na solicitação. Se não houver nenhum problema, esse objeto estará definido com `null`. Normalmente, ao fazermos uma solicitação para a nossa aplicação de armazenamento de dados, a primeira tarefa a ser feita no recebimento da resposta é verificar se há erros e tratá-los de algum modo. No caso anterior, exibimos o erro somente para que possamos saber que há algum tipo de problema; contudo você deve, definitivamente, lidar com os erros de forma mais elegante caso suas aplicações cheguem a ser executadas em ambiente de produção.

A seguir, você verá que é necessário realizar um determinado processamento no valor de `awesomeCount`. Isso ocorre porque o Redis armazena todos os valores na forma de strings, portanto devemos converter esse valor para um número para podermos realizar operações aritméticas sobre ele no JavaScript. Nesse caso, utilizamos a função global `parseInt`, que extrai o valor numérico da string retornada pelo Redis. O segundo parâmetro de `parseInt` chama-se *radix* (base) e indica que queremos o valor do número na base 10. Se o valor não for um número, `parseInt` retornará o valor `NaN`, que quer dizer – você já adivinhou – *Not a Number*.

Lembre-se de que `||` refere-se ao operador *or* (ou) do JavaScript. Esse operador retornará o primeiro valor de uma lista de valores que não seja *falsy*, ou seja, que não se refira a valores como `false` ou `0` ou `NaN`. Se todos os valores forem *falsy*, esse operador retornará o último.

Essencialmente, essa linha de código pode ser traduzida como “utilize o valor de `awesomeCount` caso ele esteja definido; do contrário, utilize `0`”. Isso nos permite inicializar o nosso código com `0` quando

awesome não estiver definido usando uma única linha de código. A esta altura, é provável que você queira tornar o código genérico para todas as palavras que estamos monitorando, porém será útil aprender mais um comando do Redis antes.

Utilizando mget para obter vários valores

O comando `get` é ótimo para um único par chave-valor, mas o que acontece se quisermos solicitar o valor associado a várias chaves? Isso é quase tão fácil de fazer utilizando a função `mget`. Podemos generalizar o nosso código da seguinte maneira:

```
redisClient.mget(["awesome", "cool"], function (err, results) {  
  if (err !== null) {  
    console.log("ERROR: " + err);  
    return;  
  }  
  
  counts.awesome = parseInt(results[0], 10) || 0;  
  counts.cool = parseInt(results[1], 10) || 0;  
}
```

Ao usar `mget`, devemos ser capazes de tornar esse código genérico, de modo que ele funcione para todas as palavras que estivermos monitorando.

O Redis é ótimo para dados simples que possam ser armazenados como strings. Isso inclui dados armazenados como objetos JSON. Entretanto, se quisermos ter um pouco mais de controle sobre os nossos dados JSON, é melhor utilizar um tipo de armazenamento de dados criado com o JSON em mente. O MongoDB é um exemplo perfeito de um armazenamento de dados desse tipo.

MongoDB

O MongoDB (ou Mongo para ser mais conciso) é um banco de dados que permite armazenar dados em disco, porém não em formato relacional. Em vez disso, o Mongo é um banco de dados orientado a documentos que, conceitualmente, permite armazenar

objetos organizados como *coleções* em formato JSON. (T tecnicamente, o MongoDB armazena seus dados em formato BSON, porém, de nosso ponto de vista, podemos pensar nele como JSON.) Além disso, o MongoDB permite interagirmos com ele totalmente em JavaScript!

O Mongo pode ser usado para necessidades mais complexas de armazenamento de dados, por exemplo, para armazenar contas de usuários ou comentários de postagens de blogs. Ele pode até mesmo ser utilizado para armazenar dados binários como imagens! Em nosso caso, o Mongo é perfeito para armazenar os objetos do Amazeriffic referentes aos itens da lista de tarefas (to-do), de modo independente do servidor.

Interagindo com o cliente de linha de comando do MongoDB

Assim como o Redis, o Mongo disponibiliza um cliente de linha de comando que nos permite interagir diretamente com os dados armazenados. O cliente do Mongo pode ser inicializado ao digitar **mongo** na linha de comando de seu computador guest:

```
vagrant $ mongo
MongoDB shell version: 2.4.7
connecting to: test
>
```



Você poderá ver alguns warnings (avisos) quando o Mongo for inicializado da primeira vez, mas isso é totalmente normal.

Uma diferença imediata entre o Redis e o Mongo é que podemos interagir com esse último utilizando o JavaScript! Por exemplo, a seguir, criamos uma variável chamado `card` e armazenamos um objeto nessa variável:

```
> var card = { "rank": "ace", "suit": "clubs" };
> card
{ "rank" : "ace", "suit" : "clubs" }
>
```

Da mesma maneira, podemos criar e manipular arrays. Observe que, no exemplo a seguir, não completamos a nossa instrução JavaScript até o final de cada linha. Quando teclamos Enter, o Mongo responde com três pontos, o que nos permite saber que a instrução anterior estava incompleta. O Mongo executará automaticamente a primeira instrução JavaScript completa:

```
> var clubs = [];
> ["two", "three", "four", "five"].forEach(
... function (rank) {
... clubs.push( { "rank":rank, "suit":"clubs" } )
... });
> clubs
[
  {
    "rank" : "two",
    "suit" : "clubs"
  },
  {
    "rank" : "three",
    "suit" : "clubs"
  },
  {
    "rank" : "four",
    "suit" : "clubs"
  },
  {
    "rank" : "five",
    "suit" : "clubs"
  }
]
```

Em outras palavras, o cliente de linha de comando do Mongo funciona de forma um pouco parecida com o JavaScript console do Chrome. No entanto essas semelhanças terminam quando começamos a armazenar dados. O Mongo organiza os dados na forma de *documentos* e podemos pensar neles como se fossem objetos JSON. Ele armazena *coleções* de documentos em *bancos de dados*. Podemos ver os bancos de dados em nosso MongoDB

por meio do comando `show dbs`:

```
> show dbs
local 0.03125GB
```

O banco de dados local está sempre presente. Podemos mudar para um banco de dados diferente por meio do comando `use`:

```
> use test
switched to db test
```

Após termos selecionado um banco de dados a ser utilizado, podemos acessá-lo por meio do objeto `db`. Podemos salvar objetos em uma coleção chamando a função `save` nessa coleção. Se a coleção ainda não existir, o Mongo a criará para nós. Neste caso, salvaremos a carta que criamos anteriormente em nossa coleção:

```
> show collections;
> db.cards.save(card);
> show collections;
cards
system.indexes
```

Você verá que a coleção `cards` não existia até salvarmos um objeto nela. Podemos usar a função `find` de coleção, sem argumentos, para ver quais documentos estão armazenados:

```
> db.cards.find();
{ "_id" : ObjectId("526ddeea7ba2be67c95558d8"), "rank": "ace", "suit": "clubs" }
```

Além das propriedades `rank` e `suit`, uma carta também tem um `_id` associado a ela. Na maioria das vezes, todo documento em uma coleção MongoDB possui um valor desses associado a ele.

Além de salvar um único item, também podemos adicionar vários documentos à coleção em uma só chamada a `save`. Neste caso, faremos isso usando o array `clubs` criado anteriormente:

```
> db.cards.save(clubs);
> db.cards.find();
{ "_id" : ObjectId("526ddeea7ba2be67c95558d8"), "rank": "ace", "suit": "clubs" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558d9"), "rank": "two", "suit": "clubs" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558da"), "rank": "three", "suit": "clubs" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558db"), "rank": "four", "suit": "clubs" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558dc"), "rank": "five", "suit": "clubs" }
```

Também podemos adicionar mais objetos ao db:

```
> hearts = [];
> ["two", "three", "four", "five"].
... forEach(function (rank)
... { hearts.push( { "rank":rank, "suit":"hearts" } )
... });
> db.cards.save(hearts);
> db.cards.find();
{ "_id" : ObjectId("526ddeea7ba2be67c95558d8"), "rank": "ace", "suit": "clubs" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558d9"), "rank": "two", "suit": "clubs" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558da"), "rank": "three", "suit": "clubs" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558db"), "rank": "four", "suit": "clubs" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558dc"), "rank": "five", "suit": "clubs" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558de"), "rank": "two", "suit": "hearts" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558df"), "rank": "three", "suit": "hearts" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558e0"), "rank": "four", "suit": "hearts" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558e1"), "rank": "five", "suit": "hearts" }
```

Quando tivermos documentos variados o suficiente em nossa coleção, poderemos acessá-los criando *queries* a partir de objetos JSON que representam as propriedades dos documentos que queremos obter. Por exemplo, podemos obter todos os documentos relacionados às cartas cujo valor seja igual a dois e armazená-los em uma variável chamada `twos`:

```
> var twos = db.cards.find({"rank":"two"});
> twos
{ "_id" : ObjectId("526ddeea7ba2be67c95558d9"), "rank": "two", "suit": "clubs" }
{ "_id" : ObjectId("526ddf0f7ba2be67c95558de"), "rank": "two", "suit": "hearts" }
```

Ou podemos selecionar todos os ases:

```
> var aces = db.cards.find({"rank":"ace"});
> aces
{ "_id" : ObjectId("526ddeea7ba2be67c95558d8"), "rank": "ace", "suit": "clubs" }
```

Também podemos remover os elementos da coleção ao chamar o método `remove` e enviar uma query:

```
> db.cards.remove({"rank":"two"});
> db.cards.find();
{ "_id" : ObjectId("526ddeea7ba2be67c95558da"), "rank": "three", "suit": "clubs" }
{ "_id" : ObjectId("526ddeea7ba2be67c95558db"), "rank": "four", "suit": "clubs" }
```

```
{ "_id" : ObjectId("526ddeea7ba2be67c95558dc"), "rank": "five", "suit": "clubs" }  
{ "_id" : ObjectId("526ddf0f7ba2be67c95558df"), "rank": "three", "suit": "hearts" }  
{ "_id" : ObjectId("526ddf0f7ba2be67c95558e0"), "rank": "four", "suit": "hearts" }  
{ "_id" : ObjectId("526ddf0f7ba2be67c95558e1"), "rank": "five", "suit": "hearts" }
```

Ou podemos remover todos os documentos da coleção se chamarmos `remove` com uma query vazia:

```
> db.cards.remove();  
> db.cards.find();  
>
```

De modo semelhante ao Redis, o MongoDB disponibiliza um tutorial interativo (<http://try.mongodb.org/>), conforme mostrado na figura 7.2, e você pode experimentar usá-lo em seu navegador web. Sugiro que você trabalhe com esse tutorial para aprender um pouco mais sobre as funcionalidades do Mongo e os tipos de queries que estão disponíveis.

No final das contas, porém, não iremos utilizar muitos dos comandos default do Mongo no Node.js. Em vez disso, iremos modelar os nossos dados na forma de objetos utilizando um módulo do Node.js chamado Mongoose.

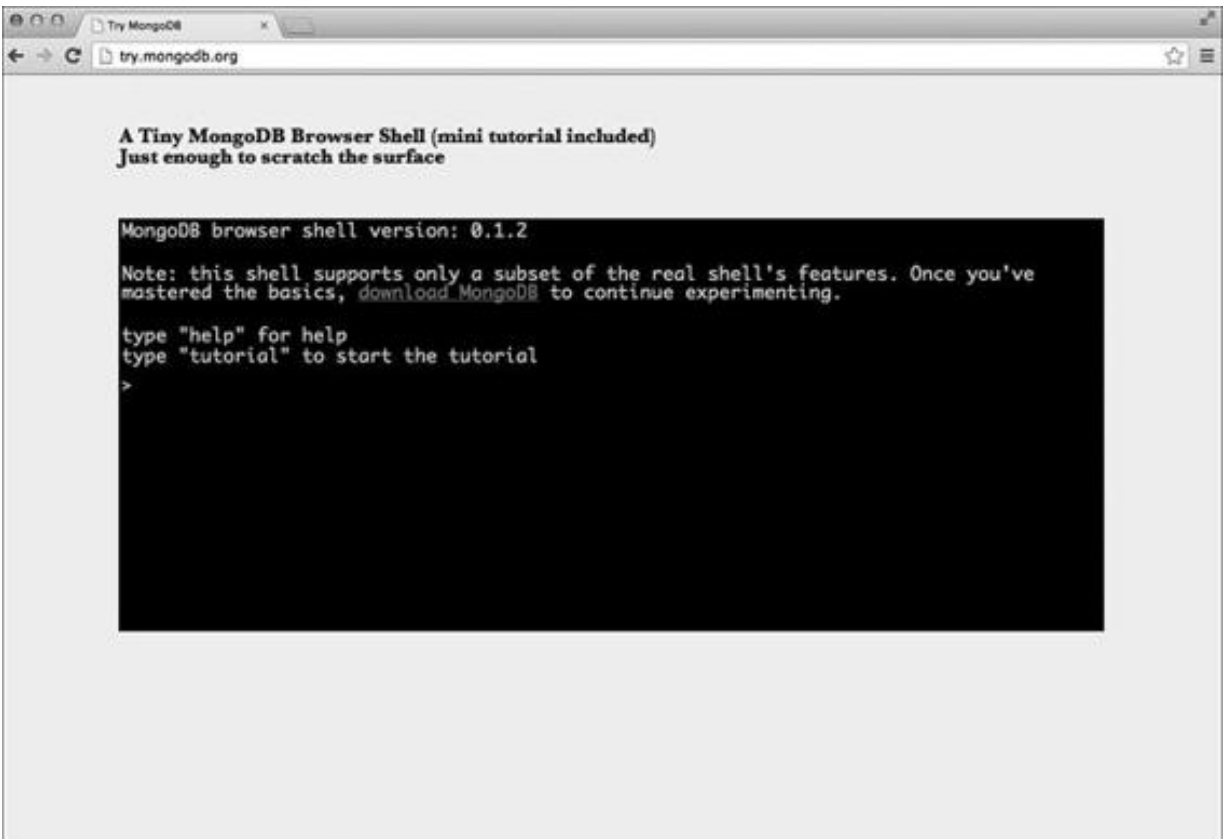


Figura 7.2 – A página inicial do tutorial interativo do MongoDB.

Modelando dados com o Mongoose

O Mongoose é um módulo do Node.js que tem dois propósitos principais. Em primeiro lugar, ele funciona como um cliente para o MongoDB, da mesma maneira que o módulo `node-redis` serve como cliente para o Redis. No entanto o Mongoose também serve como uma ferramenta para *modelagem de dados*, o que nos permite representar documentos como objetos em nossos programas. Nesta seção, aprenderemos o básico sobre modelagem de dados e usaremos o Mongoose para criar um modelo de dados para os Todos do Amazeriffic.

Modelos

Um *modelo de dados* nada mais é do que a representação de uma coleção de documentos na forma de um objeto em um sistema de armazenamento de dados. Além de especificar os campos

presentes em todos os documentos de uma coleção, os modelos adicionam operações de banco de dados do MongoDB, como *save* e *find*, aos objetos associados.

No Mongoose, um modelo de dados é constituído de um *esquema* (schema) que descreve a estrutura de todos os objetos que são do mesmo tipo. Por exemplo, suponha que queremos criar um modelo de dados para uma coleção de cartas de baralho. Devemos começar pela especificação do esquema para uma carta – ou seja, explicitamente declarando que toda carta possui um valor e um naipe. Em nosso arquivo JavaScript, isso se parece com algo do tipo:

```
var CardSchema = mongoose.Schema({
  "rank" : String,
  "suit" : String
});
```

Após criar o esquema, definir um modelo é muito fácil. Por convenção, utilizamos uma letra maiúscula para os objetos que representem modelos de dados:

```
var Card = mongoose.model("Card", CardSchema);
```

Os esquemas podem se tornar mais complexos. Por exemplo, podemos criar um esquema que contenha datas e comentários para postagens de blogs. Neste exemplo, o atributo `comments` representa não apenas uma string, mas um array delas:

```
var BlogPostSchema = mongoose.Schema({
  title: String,
  body : String,
  date : Date,
  comments : [ String ]
});
```

O que um modelo faz por nós? Após termos um modelo, podemos criar muito facilmente um objeto que seja do tipo desse modelo utilizando o operador `new` do JavaScript. Por exemplo, esta linha de código cria o ás de espadas e o armazena em uma variável chamada `c1`:

```
var c1 = new Card({"rank":"ace", "suit":"spades"});
```

Ótimo, mas não poderíamos ter feito isso de forma igualmente fácil com o código a seguir?

```
var c2 = { "rank":"ace", "suit":"spades" };
```

A diferença é que o objeto Mongoose nos permite interagir com o banco de dados por meio de algumas funções prontas!

```
// salva esta carta em nosso sistema de armazenamento de dados
c1.save(function (err) {
  if (err !== null) {
    // o objeto não foi salvo!
    console.log(err);
  } else {
    console.log("the object was saved!");
  }
});
```

Também podemos interagir diretamente com o modelo para extrair itens do banco de dados usando a função `find`, que faz parte do modelo de dados. Assim como ocorre com a função `find` no cliente interativo do MongoDB, essa função recebe uma query arbitrária. A diferença é que ela está restrita aos tipos definidos pelo modelo:

```
Card.find({}, function (err, cards) {
  if (err !== null) {
    console.log("ERROR: " + err);
    // retorna da função
    return;
  }
  // se chegamos aqui, é porque não houve erros
  cards.forEach(function (card) {
    // este código exibirá todas as cartas do banco de dados
    console.log (card.rank + " of " + card.suit);
  });
});
```

Também podemos atualizar os elementos ao encontrar aqueles que são apropriados (por meio de seu `_id` ou de outra query) e salvá-los novamente. Por exemplo, suponha que queremos alterar todas as cartas que tenham o naipe igual a copas para o naipe de espadas:


```

Card.find({"suit" : " hearts"}, function (err, cards) {
  cards.forEach(function (card) {
    // atualiza a carta para espadas
    card.suit = "spades";

    // salva a carta atualizada
    card.save(function (err) {
      if (err) {
        // o objeto não foi salvo
        console.log(err);
      }
    });
  });
});

```

Por fim, podemos remover elementos do banco de dados ao chamar a função `remove` no modelo de dados:

```

Card.remove({ "rank": "ace", "suit": "spades" }, function (err) {
  if (err !== null) {
    // o objeto não foi removido com sucesso!
    console.log(err);
  }
});

```

A essa altura, vimos exemplos das quatro principais operações associadas ao armazenamento de dados utilizando o Mongoose: criar, ler, atualizar e apagar. Mas, por enquanto, ainda não vimos realmente o Mongoose em uma aplicação! Veremos isso agora.

Armazenando os Todos do Amazeriffic

Para começar, podemos copiar nossa versão atual do Amazeriffic com o cliente e o servidor para o nosso diretório *Chapter7*. Será necessário definir um arquivo *package.json* que inclua o Mongoose como uma dependência:

```

{
  "name": "amazeriffic",
  "description": "The best to-do list app in the history of the world",
  "version": "0.0.1",
  "dependencies": {
    "mongoose": "3.6.x"
  }
}

```

```
}  
}
```

Feito isso, podemos executar `npm install` para instalar o módulo `mongoose`. Então podemos adicionar um código em `server.js` para que o módulo seja importado:

```
var express = require("express"),  
    http = require("http"),  
    // importa a biblioteca mongoose  
    mongoose = require("mongoose"),  
    app = express();  
app.use(express.static(__dirname + "/client"));  
app.use(express.urlencoded());  
  
// faz a conexão com os dados do amazeriffic armazenados no mongo  
mongoose.connect('mongodb://localhost/amazeriffic');
```

A seguir, definiremos o nosso esquema e, em seguida, o nosso modelo para os itens de nossa lista de tarefas no código do servidor:

```
// Este é o nosso modelo mongoose para as tarefas da lista  
var ToDoSchema = mongoose.Schema({  
  description: String,  
  tags: [ String ]  
});  
  
var ToDo = mongoose.model("ToDo", ToDoSchema);  
  
// agora ficamos aguardando as solicitações  
http.createServer(app).listen(3000);
```

Podemos atualizar a nossa rota `get` de tarefas para obter os itens da lista de tarefas a partir do banco de dados e retorná-los:

```
app.get("/todos.json", function (req, res) {  
  ToDo.find({}, function (err, toDos) {  
    // não se esqueça de verificar se houve erros!  
    res.json(toDos);  
  });  
});
```

Por fim, atualizaremos a nossa rota `post` para adicionar um elemento ao banco de dados. Isso é um pouco mais interessante, pois, para manter a compatibilidade com o cliente que foi modificado no capítulo 2, devemos retornar uma lista completa de itens da lista de

tarefas:

```
app.post("/todos", function (req, res) {
  console.log(req.body);
  var newToDo = new ToDo({"description":req.body.description,
    "tags":req.body.tags});
  newToDo.save(function (err, result) {
    if (err !== null) {
      console.log(err);
      res.send("ERROR");
    } else {
      // nosso cliente espera que *todos* os itens da lista de tarefas
      // sejam retornados, portanto precisamos de uma solicitação
      // adicional para manter a compatibilidade
      ToDo.find({}, function (err, result) {
        if (err !== null) {
          // o elemento não foi salvo!
          res.send("ERROR");
        }
        res.json(result);
      });
    }
  });
});
```

Agora podemos executar o nosso servidor e nos conectar por meio do navegador. Tenha em mente que não inicializamos nossa coleção com nenhum `ToDo`, portanto será necessário adicionar elementos antes que possamos ver qualquer item da lista de tarefas aparecer em nossa aplicação.

Resumo

Neste capítulo, aprendemos a armazenar dados de maneira independente de nossa aplicação. Vimos duas soluções para armazenamento de dados que podem ser consideradas exemplos de bancos de dados NoSQL: o Redis e o MongoDB.

O Redis armazena dados no formato chave-valor. Ele oferece uma solução rápida e flexível para armazenar dados simples. O módulo

`node-redis` nos permite interagir com o Redis de maneira quase idêntica ao modo como interagimos com ele a partir da linha de comando.

O MongoDB é uma solução mais robusta para armazenamento de dados, em que os bancos de dados são organizados em coleções. Como usuários do banco de dados, podemos pensar nos dados como se estivessem armazenados na forma de objetos JavaScript. Além do mais, a interface de linha de comando do Mongo pode ser utilizada como um interpretador JavaScript básico, que permite interagir com nossas coleções.

O Mongoose é um módulo do Node.js que possibilita a interação com o MongoDB, mas também permite criar modelos de dados. Além de especificar como devem ser todos os elementos de uma coleção de dados, os modelos permitem interagir com o banco de dados por meio de objetos em nosso programa.

Práticas e leituras adicionais

API para pôquer

Se você vem acompanhando as seções práticas dos capítulos anteriores, você deve ter criado uma API para pôquer que aceite mãos de pôquer e retorne o tipo da mão (por exemplo, um par, um full house). Seria interessante acrescentar um código ao seu servidor Express que armazene o resultado de todas as mãos e os seus tipos no Redis ou no Mongo. Em outras palavras, faça com que sua aplicação registre todas as mãos válidas, submetidas à sua API, juntamente com o resultado. Não registre os itens que resultarem em erro.

Se fizer isso, você poderá definir uma rota `get` que responda com um objeto JSON e que inclua todas as mãos enviadas. Você pode até mesmo fazer com que a rota `get` retorne somente as cinco mãos mais recentes.

Se quiser tornar esse exemplo *realmente* interessante, procure

encontrar uma biblioteca de imagens de cartas de baralho e faça o seu cliente mostrar as imagens das cartas da mão que foi enviada, em vez de mostrar os objetos JSON ou uma descrição textual. Para isso, você pode criar um subdiretório *images* na parte de sua aplicação referente ao cliente e armazenar as imagens nesse local.

Outras referências a bancos de dados

Banco de dados é um assunto muito amplo, que merece diversos livros contendo materiais a esse respeito. Se você quiser aprender mais sobre o desenvolvimento de aplicações web e sobre programação em geral, é muito importante que, em algum momento, você adquira mais conhecimentos sobre bancos de dados relacionais e SQL.

O Coursera (<http://www.coursera.org/>) é um ótimo local para ter aulas online gratuitas sobre assuntos como bancos de dados. Com efeito, o site disponibiliza um curso de banco de dados que você pode fazer por conta própria, ministrado por Jennifer Widom, da Stanford University!

Se você preferir os livros, Jennifer Widom é coautora (juntamente com Hector Garcia-Molina e Jeffrey Ullman) do livro *Database Systems: The Complete Book* (Prentice Hall, 2008). Definitivamente, o livro segue uma abordagem mais acadêmica para o assunto, mas acho que é uma introdução excelente e fácil de ler.

Outro ótimo livro que discute diferentes tipos de banco de dados (incluindo os bancos de dados NoSQL) é o *Databases in Seven Weeks* de Eric Redmond e Jim Wilson (Pragmatic Bookshelf, 2012). Além de incluir o Mongo e o Redis, o livro também descreve opções populares como o PostgreSQL e o Riak. Ele faz um ótimo trabalho de descrição dos prós e dos contras de cada banco de dados, além de apresentar vários casos de uso.

CAPÍTULO 8

Plataforma

A essa altura, sabemos criar uma aplicação web de modo muito básico, com tecnologias utilizadas do lado do cliente e do servidor, porém nossa aplicação não será de muita utilidade se estiver executando somente em nosso computador. A próxima peça do quebra-cabeça é compartilhar a nossa aplicação fazendo com que ela seja executada na web.

No passado, fazer isso exigia um volume incrível de trabalho – era preciso comprar espaço em servidores com um endereço IP fixo, instalar e configurar os softwares necessários, comprar um nome de domínio e fazer esse nome apontar para a nossa aplicação. Felizmente, os tempos mudaram e, atualmente, há uma classe de serviços de hospedagem chamada *PaaS* (Platforms-as-a-Service, ou Plataformas como serviço) que cuida do trabalho pesado para nós.

Você já deve ter ouvido falar do termo *computação em nuvem* (cloud computing); esse conceito associado está baseado na ideia de que os detalhes de baixo nível relativos à manutenção de software e ao processamento podem e devem ser transferidos dos computadores locais para a web. Com um PaaS, não precisamos conhecer nenhum detalhe do gerenciamento e da configuração de um servidor web e podemos nos concentrar mais somente em fazer nossas aplicações funcionarem corretamente. Em outras palavras, um PaaS é o tipo de tecnologia que o modelo de computação em nuvem torna possível.

Neste capítulo, aprenderemos a fazer a implantação de nossas aplicações web em um PaaS de código aberto chamado Cloud

Foundry. Embora o foco vá ser no Cloud Foundry, a maioria dos conceitos que aprenderemos podem ser generalizados para outros serviços PaaS, como o Heroku ou o Nodejitsu.

O Cloud Foundry

O Cloud Foundry é um PaaS de código aberto originalmente desenvolvido pela VMware. Você pode ler mais a respeito do serviço em sua página inicial, localizada em [<http://run.pivotal.io>] (<http://run.pivotal.io>), a qual está sendo mostrada na figura 8.1. O Cloud Foundry oferece um trial de 60 dias que pode ser usado para fazer a implantação de algumas das aplicações presentes neste livro.



Figura 8.1 – Página inicial do Cloud Foundry (run.pivotal.io).

Criando uma conta

Para começar, é necessário criar uma conta em [<http://run.pivotal.io>] (<http://run.pivotal.io>). A versão trial de 60 dias pode ser acessada ao clicar no link que está no canto superior à direita e digitar o seu

endereço de email. Você receberá uma resposta contendo informações sobre como criar a sua conta.

Preparando a sua aplicação para a implantação

Para efetuar a implantação de suas aplicações, será necessária uma ferramenta chamada *cf*. A instalação dessa ferramenta em seu computador local exige que você primeiro instale a linguagem Ruby e o gerenciador de pacote RubyGems. Incluí também a aplicação *cf* como parte do projeto *node-dev-bootstrap*, portanto você poderá efetuar a implantação diretamente a partir de seu computador guest, sem a necessidade de nenhuma instalação adicional.

Para começar, crie um diretório *Chapter8* efetuando novamente a clonagem do projeto *node-dev-bootstrap*, como foi feito nos dois capítulos anteriores. Feito isso, inicie o seu computador guest e conecte-se a ele usando o SSH.

Começaremos fazendo a implantação do programa *server.js* básico que acompanha o *node-dev-bootstrap*, porém, antes disso, precisamos realizar duas tarefas. Em primeiro lugar, devemos adicionar um arquivo *package.json* porque o Cloud Foundry espera que esse arquivo esteja presente em todas as aplicações a serem implantadas. Como a nossa aplicação servidora default não depende de nenhum módulo externo do Node.js, o arquivo é bem compacto:

```
{
  "name": "sp_example",
  "description": "My first Cloud Foundry app!"
}
```

A seguir, precisaremos fazer uma pequena modificação no arquivo *server.js* default, que está incluído no projeto *node-dev-bootstrap*. Essa mudança está relacionada ao número da porta na qual o nosso servidor está ouvindo: por razões técnicas, o Cloud Foundry deve atribuir um número de porta que o nosso código deve ouvir, porém,

normalmente, não saberemos que porta é essa até que o nosso código esteja executando. Felizmente, o Cloud Foundry disponibiliza o número da porta para nós em uma *variável de ambiente* chamada `PORT`. Podemos acessá-la por meio da variável `process.env.PORT` em nosso programa `Node.js`:

```
var http = require("http"),
    // se a variável de ambiente PORT estiver definida, fique ouvindo nessa
    // porta; do contrário, fique ouvindo na porta 3000
    port = process.env.PORT || 3000;
var server = http.createServer(function (req, res) {
  res.writeHead(200, {"Content-Type": "text/plain"});
  res.end("Hello from Cloud Foundry!");
});
server.listen(port);
console.log("Server listening on port " + port);
```

Novamente, vemos o *idiom* `||` apresentado no capítulo 7. Basicamente, esse código diz o seguinte: se `process.env.PORT` estiver definido, utilize-o; caso contrário, utilize a porta 3000. Isso nos permite definir o número de porta de modo que a nossa aplicação funcione corretamente tanto em nosso computador `guest` quanto se estiver no Cloud Foundry.

Fazendo a implantação de nossa aplicação

A esta altura, devemos ser capazes de executar a aplicação em nossa máquina virtual e acessá-la a partir do navegador, como fizemos anteriormente. E agora que criamos um arquivo *package.json* e configuramos nossa aplicação para que fique ouvindo na porta correta, também estamos prontos para efetuar a sua implantação no Cloud Foundry.

Como mencionado anteriormente, isso exige o uso do programa `cf`, que já está instalado em nosso computador `guest`. Então, para começar, vá até o diretório que contém o arquivo *server.js* em seu computador `guest`.

A API da plataforma Cloud Foundry está localizada em

api.run.pivotal.io. Portanto começamos informando ao `cf` o local em que a plataforma Cloud Foundry alvo está localizada utilizando o subcomando `target`:

```
vagrant $ cf target api.run.pivotal.io  
Setting target to https://api.run.pivotal.io... OK
```

Em seguida, faremos login com o subcomando `login` e digitaremos nossas credenciais do modo como as definimos quando criamos a nossa conta no Cloud Foundry. Após efetuarmos o login, o `cf` perguntará pelo espaço de implantação que queremos utilizar. Normalmente, eu uso o espaço de *desenvolvimento* (development) quando estou fazendo testes:

```
vagrant $ cf login  
target: https://api.run.pivotal.io  
Email> me@semmy.me  
Password> *****  
Authenticating... OK  
1: development  
2: production  
3: staging  
Space> 1  
Switching to space development... OK
```

Se tudo correr bem, o nosso próximo passo será enviar nossa aplicação para o Cloud Foundry! Como podemos fazer isso? É tão simples quanto usar o subcomando `push`. Juntamente com esse subcomando, será necessário incluir o comando que inicia a nossa aplicação:

```
vagrant $ cf push --command "node server.js"
```

O subcomando `push` nos coloca em um diálogo com o Cloud Foundry. Ele nos fará algumas perguntas a respeito de nossa aplicação e do ambiente que utilizaremos. Aqui está o meu diálogo com o Cloud Foundry – o seu deverá ser semelhante a este exemplo:

```
Name> sp_example  
Instances> 1
```

1: 128M
2: 256M
3: 512M
4: 1G
Memory Limit> **256M**
Creating sp_example... OK
1: sp_example
2: none
Subdomain> **sp_example**
1: cfapps.io
2: none
Domain> **cfapps.io**
Creating route sp_example.cfapps.io... OK
Binding sp_example.cfapps.io to sp_example... OK
Create services for application?> **n**
Save configuration?> **n**
Uploading sp_example... OK
Preparing to start sp_example... OK
-----> Downloaded app package (4.0K)
-----> Resolving engine versions
WARNING: No version of Node.js specified in package.json, see:
<https://devcenter.heroku.com/articles/nodejs-versions>
Using Node.js version: 0.10.21
Using npm version: 1.2.30
-----> Fetching Node.js binaries
-----> Vendoring node into slug
-----> Installing dependencies with npm
npm WARN package.json sp_example@ No repository field.
npm WARN package.json sp_example@ No readme data.
Dependencies installed
-----> Building runtime environment
-----> Uploading droplet (15M)
Checking status of app 'sp_example'...
1 of 1 instances running (1 running)
Push successful! App 'sp_example' available at sp_example.cfapps.io



O nome de sua aplicação deve ser único entre todos os nomes desse domínio. Isso significa que, se você der um nome como **example** à sua aplicação, é provável que você vá obter um erro ao tentar efetuar a sua implantação. Eu

procuro evitar esse erro ao adicionar minhas iniciais e um underscore no início do nome que vou utilizar. Isso nem sempre funciona, portanto pode ser que sejam necessárias outras soluções para gerar nomes exclusivos para as aplicações.

Se tudo correr bem, sua aplicação estará executando na web agora! Podemos confirmar isso ao abrir um navegador web e acessar o URL que o `cf` nos disponibilizou (no meu caso, é *http://sp_example.cfapps.io*). Feito isso, você deverá ver uma resposta de sua aplicação.

Obtendo informações sobre suas aplicações

Agora que sua aplicação está pronta e executando, você pode usar outros subcomandos do `cf` para obter informações sobre o status de suas aplicações. Por exemplo, utilize o subcomando `apps` para obter uma lista de todas as suas aplicações no Cloud Foundry e os seus respectivos status:

```
vagrant $ cf apps
name status usage url
sp_example running 1 x 256M sp_example.cfapps.io
```

Um dos principais problemas de executar sua aplicação em um PaaS é que você não consegue ver os resultados de suas instruções `console.log` tão facilmente quanto o faz quando a sua aplicação é executada localmente. Isso pode ser um grande problema se o seu programa falhar e você tiver de determinar o motivo. Felizmente, o Cloud Foundry disponibiliza o subcomando `logs`, que pode ser usado nas aplicações em execução para que você possa visualizar os logs de seus programas:

```
vagrant $ cf logs sp_example
Getting logs for sp_example #0... OK

Reading logs/env.log... OK
TMPDIR=/home/vcap/tmp
VCAP_APP_PORT=61749
USER=vcap
VCAP_APPLICATION= { ... }
PATH=/home/vcap/app/bin:/home/vcap/app/node_modules/.bin:/bin:/usr/bin
PWD=/home/vcap
```

```
VCAP_SERVICES={}
SHLVL=1
HOME=/home/vcap/app
PORT=61749
VCAP_APP_HOST=0.0.0.0
MEMORY_LIMIT=256m
_=/usr/bin/env
Reading logs/staging_task.log... OK
-----> Downloaded app package (4.0K)
-----> Resolving engine versions

    WARNING: No version of Node.js specified in package.json, see:
    https://devcenter.heroku.com/articles/nodejs-versions

    Using Node.js version: 0.10.21
    Using npm version: 1.2.30
-----> Fetching Node.js binaries
-----> Vendoring node into slug
-----> Installing dependencies with npm
    npm WARN package.json sp_example@ No repository field.
    npm WARN package.json sp_example@ No readme data.
    Dependencies installed
-----> Building runtime environment
Reading logs/stderr.log... OK
Reading logs/stdout.log... OK
Server listening on port 61749
```

Você verá que o Cloud Foundry exibe o conteúdo de quatro logs armazenados. O primeiro é *env.log*, que contém todas as variáveis de ambiente que podem ser acessadas por meio da variável `process.env` em seu programa. O segundo é *staging_task.log*, que registra tudo o que acontece quando o programa é iniciado (você perceberá que o seu conteúdo é o mesmo exibido quando `cf push` foi executado pela primeira vez). Os últimos dois logs são *stderr.log* e *stdout.log*. Você verá que *stdout.log* inclui a instrução *console.log* utilizada em seu programa. Se você utilizar *console.err*, sua mensagem aparecerá em *stderr.log*.

Atualizando a sua aplicação

Você pode facilmente enviar uma versão mais nova de sua aplicação para o Cloud Foundry efetuando novamente um push. Modifique *server.json* para que ele retorne um pouco mais de informações:

```
var http = require("http"),
    port = process.env.PORT || 3000;

var server = http.createServer(function (req, res) {
  res.writeHead(200, {"Content-Type": "text/plain"});
  res.write("The server is running on port " + port);
  res.end("Hello from Cloud Foundry!");
});

server.listen(port);
console.log("Server listening on port " + port);
```

Feita essa alteração, você poderá executar o subcomando `push` de novo, juntamente com o nome da aplicação que você quer atualizar. As novas alterações serão enviadas sem que seja necessário responder a todas as perguntas novamente:

```
vagrant $ cf push sp_example
Save configuration?> n

Uploading sp_example... OK
Stopping sp_example... OK

Preparing to start sp_example... OK
-----> Downloaded app package (4.0K)
-----> Downloaded app buildpack cache (4.0K)
-----> Resolving engine versions

WARNING: No version of Node.js specified in package.json, see:
https://devcenter.heroku.com/articles/nodejs-versions

Using Node.js version: 0.10.21
Using npm version: 1.2.30
-----> Fetching Node.js binaries
-----> Vendoring node into slug
-----> Installing dependencies with npm
    npm WARN package.json sp_example@ No repository field.
    npm WARN package.json sp_example@ No readme data.
    Dependencies installed
-----> Building runtime environment
-----> Uploading droplet (15M)
```

Checking status of app 'sp_example'...

1 of 1 instances running (1 running)

Push successful! App 'sp_example' available at sp_example.cfapps.io

Apagando aplicações do Cloud Foundry

Ocasionalmente, vamos querer apagar aplicações do Cloud Foundry, particularmente quando estivermos apenas fazendo testes. Para isso, podemos utilizar o subcomando `delete`:

```
vagrant $ cf delete sp_example
```

```
Really delete sp_example?> y
```

```
Deleting sp_example... OK
```

As dependências e o package.json

Nos exemplos anteriores, nossas aplicações apresentavam dependências externas, por exemplo, em relação aos módulos `express`, `redis`, `mongoose` e `ntwitter`. A utilização de módulos básicos que não se conectam a serviços externos (como o `express` e o `ntwitter`) é muito simples. Como, normalmente, o nosso diretório `node_modules` não é enviado, só precisamos garantir que todas as nossas dependências estejam listadas em nosso arquivo `package.json`.

Por exemplo, considere uma de nossas primeiras aplicações Express. Após algumas modificações pequenas para que ela ouça a porta correta no Cloud Foundry, a aplicação terá o seguinte aspecto:

```
var express = require("express"),
    http = require("http"),
    app = express(),
    port = process.env.PORT || 3000;;

http.createServer(app).listen(port);
console.log("Express is listening on port " + port);

app.get("/hello", function (req, res) {
  res.send("Hello World!");
});

app.get("/goodbye", function (req, res) {
  res.send("Goodbye World!");
});
```

```
});
```

Será necessário incluir nossa dependência do módulo `Express` no arquivo *package.json*, o que pode ser feito exatamente da mesma maneira que fizemos em nosso exemplo anterior de *package.json*:

```
{
  "name": "sp_express",
  "description": "a sample Express app",
  "dependencies": {
    "express": "3.4.x"
  }
}
```

Você verá que eu especifiquei que nossa aplicação depende do módulo `Express`, em especial de qualquer número de versão que comece com 3.4 (o x é um caractere curinga). Isso informa ao Cloud Foundry qual versão deve ser instalada para que a nossa aplicação execute corretamente. Após termos incluído as dependências em nosso arquivo *package.json*, podemos enviar a aplicação ao Cloud Foundry utilizando o mesmo comando usado anteriormente:

```
vagrant $ ~/app$ cf push --command "node server.js"
Name> sp_expressexample
```

Feito isso, podemos acessar http://sp_expressexample.cfapps.io/hello ou http://sp_expressexample.cfapps.io/goodbye para ver nossa aplicação responder!

Fazer a nossa aplicação do Twitter ou do Amazeriffic funcionar é um pouco mais desafiador porque elas dependem de outros serviços de armazenamento de dados – especificamente do Redis e do MongoDB. Isso significa que será necessário criar os serviços e, em seguida, configurar a nossa aplicação para utilizá-los.

Associando o Redis à sua aplicação

Quando executamos nossa aplicação na máquina virtual, os serviços como o Redis e o MongoDB executam localmente. Isso muda um pouco quando executamos nossa aplicação em um PaaS.

Às vezes, os serviços são executados no mesmo host, porém, em outras ocasiões, eles são executados em outros serviços de hosting. De qualquer modo, você deve começar pela configuração do serviço que você quer que seja executado em suas interações com o cf. Nesta seção, configuraremos o Redis no Cloud Foundry e depois faremos a nossa aplicação de contagem de tuítes se conectar a ele. Começaremos copiando a nossa aplicação do Twitter do capítulo 7 para o nosso diretório *Chapter8/app*. Certifique-se de que o seu arquivo *package.json* exista e inclua as dependências do *redis* e do *ntwitter*. O meu arquivo tem o seguinte aspecto:

```
{
  "name": "tweet_counter",
  "description": "tweet counter example for learning web app development",
  "dependencies": {
    "ntwitter": "0.5.x",
    "redis": "0.9.x"
  }
}
```

Também devemos atualizar *server.js* para que ele fique ouvindo a porta especificada em `process.env.PORT`. Feito isso, podemos tentar enviar a nossa aplicação ao Cloud Foundry! Vou me adiantar e mencionar com antecedência que haverá falhas, porém isso nos dará a oportunidade de configurar o nosso serviço Redis:

```
vagrant $ cf push --command "node server.js"
```

```
Name> sp_tweetcounter
```

```
Instances> 1
```

```
1: 128M
```

```
2: 256M
```

```
3: 512M
```

```
4: 1G
```

```
Memory Limit> 256M
```

```
Creating sp_tweetcounter... OK
```

```
1: sp_tweetcounter
```

```
2: none
```

```
Subdomain> sp_tweetcounter
```

1: cfapps.io

2: none

Domain> **cfapps.io**

Binding sp_tweetcounter.cfapps.io to sp_tweetcounter... OK

Create services for application?> **y**

1: blazemeter n/a, via blazemeter

2: cleardb n/a, via cleardb

3: cloudamqp n/a, via cloudamqp

4: elephantsql n/a, via elephantsql

5: loadimpact n/a, via loadimpact

6: mongolab n/a, via mongolab

7: newrelic n/a, via newrelic

8: rediscloud n/a, via garantiadata

9: sendgrid n/a, via sendgrid

10: treasuredata n/a, via treasuredata

11: user-provided , via

Observe que informamos o Cloud Foundry de que gostaríamos de criar um serviço para a nossa aplicação:

What kind?> **8**

Name?> **rediscloud-dfc38**

1: 20mb: Lifetime Free Tier

Which plan?> **1**

Creating service rediscloud-dfc38... OK

Binding rediscloud-dfc38 to sp_tweetcounter... OK

Create another service?> **n**

Bind other services to application?> **n**

Save configuration?> **n**

Encerre o diálogo e você perceberá que o envio falhou. Isso ocorre porque ainda não dissemos à nossa aplicação de que modo ela deve se conectar ao serviço Redis externo. Os logs devem dar uma pista para nós. Com efeito, ao executar `cf logs sp_tweetcounter`, devemos ver algo como o que está a seguir em meu *logs/stderr.log*:

Reading logs/stderr.log... OK

events.js:72

throw er; // Unhandled 'error' event

^

```
Error: Redis connection to 127.0.0.1:6379 failed - connect ECONNREFUSED
  at RedisClient.on_error (/home/vcap/app/node_modules/redis/index.js:189:24)
  at Socket.<anonymous> (/home/vcap/app/node_modules/redis/index.js:95:14)
  at Socket.EventEmitter.emit (events.js:95:17)
  at net.js:441:14
  at process._tickCallback (node.js:415:13)
```

Em nossa aplicação, não enviamos nenhum argumento ao Redis, portanto ele tenta se conectar ao servidor local (127.0.0.1) utilizando também o número de porta default (6379). Devemos informá-la de que ela deve se conectar ao serviço Redis da nuvem no host e na porta disponibilizado pelo Cloud Foundry. Em que local o Cloud Foundry nos disponibiliza essas informações? Assim como ocorre com o número da porta, ele nos passa as informações sobre os serviços associados por meio da variável `process.env`!

Se você já digitou `cf logs sp_tweetcounter`, será possível fazer rolagens para ver as variáveis de ambiente e observar o conteúdo da variável `VCAP_SERVICES`. Nos logs, ela corresponde a uma string JSON longa, portanto eu a formatei aqui para que ela se tornasse um pouco mais legível:

```
VCAP_SERVICES = {
  "rediscloud-n/a": [{
    "name": "rediscloud-dfc38",
    "label": "rediscloud-n/a",
    "tags": ["redis", "key-value"],
    "plan": "20mb",
    "credentials": {
      "port": "18496",
      "hostname": "pub-redis-18496.us-east-1-4.2.ec2.garantiadata.com",
      "password": "REDACTED"
    }
  ]
}
```

Esses dados incluem todas as informações de que precisamos para efetuar a conexão com um serviço Redis remoto, incluindo um URL, uma porta e uma senha. Se quiséssemos, poderíamos facilmente codificar essas credenciais do Redis diretamente em nosso

programa. É provável, porém, que o melhor seja fazer isso por meio de linhas de programação, da mesma maneira que definimos o número da porta.



Uma pequena complicação é que a variável de ambiente `VCAP_SERVICES` está armazenada na forma de uma string, portanto devemos utilizar o comando `JSON.parse` do mesmo modo que fizemos no capítulo 5 para convertê-la em um objeto que possa ser acessado como qualquer outro objeto JavaScript.

Em suma, podemos fazer o Redis funcionar ao modificar a primeira parte de nosso módulo *tweet_counter.js* para que se pareça com:

```
var ntwitter = require("ntwitter"),
    redis = require("redis"), // faz o require do módulo redis
    credentials = require("./credentials.json"),
    redisClient,
    counts = {},
    twitter,
    services,
    redisCredentials;

// cria o nosso cliente twitter
twitter = ntwitter(credentials);

// configura os nossos serviços caso a variável de ambiente exista
if (process.env.VCAP_SERVICES) {
    // faz o parse da string JSON
    services = JSON.parse(process.env.VCAP_SERVICES);
    redisCredentials = services["rediscloud-n/a"][0].credentials;
} else {
    // do contrário, definiremos valores default
    redisCredentials = {
        "hostname": "127.0.0.1",
        "port": "6379",
        "password": null
    };
}

// cria um client para se conectar ao Redis
client = redis.createClient(redisCredentials.port, redisCredentials.hostname);

// faz a autenticação
client.auth(redisCredentials.password);
```

Nesse trecho de código, verificamos se a variável de ambiente `VCAP_SERVICES` existe e, em caso afirmativo, fazemos o parse da

string associada a ela para gerar um objeto JavaScript a partir da representação JSON. A seguir, extraímos as credenciais associadas à propriedade `rediscloud-n/a` do objeto `services`. Esse objeto é um array (porque nossa aplicação poderia estar associada a várias instâncias de Redis), portanto acessamos o primeiro elemento do array.

Se a variável de ambiente `VCAP_SERVICES` não estiver definida, um objeto `redisCredentials` que contém os valores default será criado. Depois disso, iremos nos conectar ao Redis especificando a porta e o nome do host e, em seguida, enviando a senha. Se estivermos conectados com a instância local do Redis em nossa máquina virtual, enviaremos `null` para a senha, pois ele não exige autenticação.

Se já tivermos enviado a aplicação uma vez e tiver havido falha, podemos simplesmente enviá-la novamente para atualizá-la. Desta vez, farei o envio utilizando o nome que eu dera anteriormente. Caso eu me esqueça desse nome, sempre poderei utilizar `cf apps` para consultar a lista de minhas aplicações:

```
vagrant $ cf apps
Getting applications in development... OK

name status usage url
sp_example running 1 x 256M sp_example.cfapps.io
sp_express running 1 x 256M sp_express.cfapps.io
sp_tweetcounter running 1 x 256M sp_tweetcounter.cfapps.io
vagrant $ cf push sp_tweetcounter
```

Se tudo estiver definido corretamente, você verá a aplicação ser enviada com sucesso e deverá ser capaz de acessá-la por meio do URL que o Cloud Foundry disponibilizar a você!

Associando o MongoDB à sua aplicação

Ter o MongoDB pronto para executar no Cloud Foundry é quase tão fácil quanto fazer o Redis executar. Vamos começar copiando nossa aplicação Amazeriffic de *Chapter7* para o nosso diretório corrente. Serão necessárias algumas modificações que seguem o mesmo

padrão usado anteriormente.

Em primeiro lugar, iremos alterar a aplicação para que ela fique ouvindo a porta especificada em `process.env.PORT`, caso esse valor exista. O código para isso é idêntico ao dos exemplos da seção anterior.

Depois disso, será preciso obter as nossas credenciais do MongoDB a partir de `process.env.VCAP_SERVICES`. Esse código será muito semelhante ao código usado para o Redis. A principal diferença é que as nossas credenciais do MongoDB estão todas contidas em uma única string – o `uri`:

```
// não se esqueça de declarar mongoUrl em algum lugar antes deste ponto
// define os nossos serviços
if (process.env.VCAP_SERVICES) {
  services = JSON.parse(process.env.VCAP_SERVICES);
  mongoUrl = services["mongolab-n/a"][0].credentials.uri;
} else {
  // utilizaremos isto quando não estivermos executando no Cloud Foundry
  mongoUrl = "mongodb://localhost/amazeriffic"
}
```

Depois que esse código estiver funcionando, podemos enviar a nossa aplicação para o Cloud Foundry, como fizemos anteriormente. Uma pequena alteração diz respeito à necessidade de configurar o serviço MongoLab:

```
Create services for application?> y
1: blazemeter n/a, via blazemeter
2: cleardb n/a, via cleardb
3: cloudamqp n/a, via cloudamqp
4: elephantsql n/a, via elephantsql
5: loadimpact n/a, via loadimpact
6: mongolab n/a, via mongolab
7: newrelic n/a, via newrelic
8: rediscloud n/a, via garantiadata
9: sendgrid n/a, via sendgrid
10: treasuredata n/a, via treasuredata
11: user-provided , via
What kind?> 6
```

```
Name?> mongolab-8a0f4
```

```
1: sandbox: 0.5 GB
```

```
Which plan?> 1
```

```
Creating service mongolab-8a0f4... OK
```

Após o serviço Mongo ter sido criado, basta associá-lo à nossa aplicação, como fizemos com o Redis. Desde que o nosso código esteja preparado para se conectar com um serviço remoto por meio do URL disponibilizado pela variável de ambiente `VCAP_SERVICES`, nosso programa deverá ser iniciado exatamente como esperamos.

Resumo

Neste capítulo, aprendemos a usar o Cloud Foundry para disponibilizar nossas aplicações na web. O Cloud Foundry é um exemplo de um PaaS (*Platform as a Service*, ou Plataforma como serviço). Um PaaS é uma tecnologia de computação em nuvem que faz a abstração da configuração, da administração e da hospedagem do servidor, geralmente por meio de um programa de linha de comando ou de uma interface web.

Para que nossas aplicações sejam executadas no Cloud Foundry (ou em qualquer outro PaaS), normalmente devemos fazer pequenas modificações no código para que ele se comporte de modo diferente, de acordo com o fato de a aplicação estar sendo executada localmente em nosso ambiente de desenvolvimento ou em produção nos servidores do Cloud Foundry.

O Cloud Foundry também disponibiliza serviços externos como o Redis e o MongoDB. Para que nossas aplicações executem com esses serviços, é necessário *criar* o serviço usando o programa `cf` e, em seguida, *associá-lo* à nossa aplicação.

Práticas e leituras adicionais

API para pôquer

Nas seções práticas dos capítulos anteriores, criamos uma API

simples de pôquer que determinava como eram as mãos de pôquer. No último capítulo, adicionamos um componente de banco de dados a essa aplicação. Se ela estiver funcionando, tente modificá-la para que a sua implantação seja feita no Cloud Foundry.

Outras plataformas

Não há escassez de plataformas para nuvens hoje em dia. Uma das plataformas mais populares é o Heroku (<http://heroku.com/>). Ele permite criar uma conta gratuitamente, sem a necessidade de um cartão de crédito. Entretanto, se quiser adicionar o Redis ou o Mongo, você será solicitado a fornecer um, embora associações gratuitas para ambos os serviços estejam disponíveis.

Se quiser praticar mais, procure ler a documentação para implantação do Node.js no Heroku e faça a implantação de uma ou mais aplicações suas nesse serviço. Também sugiro que você experimente usar outras opções, incluindo o Nodejitsu (<http://nodejitsu.com/>) e o Windows Azure da Microsoft (<http://windowsazure.com/>). Todos eles diferem um pouco, portanto comparar e contrastar os recursos de cada um é um ótimo exercício de aprendizagem.

CAPÍTULO 9

Aplicação

Se você chegou até este ponto em sua jornada, então deverá ser capaz de fazer tanto o cliente quanto o servidor funcionarem usando HTML, CSS e JavaScript. E se tiver uma ideia para uma aplicação web, você poderá muito bem usar o que aprendeu para escrever e reunir algum código para que ela faça o que deve fazer.

Em sua maior parte, este capítulo explica por que você não deve fazer isso, ao menos não por enquanto. David Parnas certa ocasião disse que “um programador ruim pode criar facilmente dois novos empregos por ano”. Embora eu ache que isso seja verdade, em minha experiência, programadores “ruins” não são ruins por natureza; eles só não têm a experiência e a sabedoria para pensar na organização e na facilidade de manutenção de suas bases de código. O código (especialmente em JavaScript) pode tornar-se rapidamente confuso e impossível de manter nas mãos de um programador inexperiente.

Conforme mencionei no prefácio, como iniciante, é totalmente aceitável sair por aí escrevendo códigos enquanto você estiver aprendendo, porém, à medida que continuar, você logo perceberá que essa não é a melhor maneira de lidar com problemas maiores e mais complexos de engenharia. Em outras palavras, ainda há muito que aprender (e sempre haverá).

Por outro lado, os desenvolvedores de software têm criado aplicações web baseadas em banco de dados há vários anos, portanto é praticamente inevitável que alguns padrões básicos tenham surgido e que sirvam de base para um melhor design de código. Com efeito, frameworks completos de desenvolvimento

como o Ruby on Rails foram criados para *forçar* (ou pelo menos para *incentivar fortemente*) os desenvolvedores a criar aplicações que utilizem alguns desses padrões. Enquanto terminamos a nossa jornada pelo básico, faremos uma revisão do Amazeriffic e procuraremos entender de que modo podemos facilitar a manutenção do nosso código atual. Também veremos como uma parte de nosso código pode ser modificada para que se enquadre nesses padrões.

Efetuando a refatoração de nosso cliente

Começaremos pelo código de nosso cliente. Em minha experiência, o cliente é um dos lugares mais fáceis de a manutenção de código se tornar mais complicada. Acho que isso pode ser atribuído ao fato de que o código do lado do cliente normalmente está associado a ações visuais aparentemente simples e, desse modo, passa a *impressão* de ser bem fácil de ser implementado. Por exemplo, prover um comportamento específico quando um usuário executar uma ação trivial, como clicar em um botão, nos dá a impressão de que essa *deve* ser uma tarefa simples de codificação. Mas nem sempre é o caso.

Em nosso exemplo, criar uma interface de usuário utilizando abas nos dá a impressão de ser uma tarefa relativamente simples. E, em nosso caso, foi. Mas podemos facilitar bastante a manutenção se fizermos uma pequena refatoração e seguirmos alguns conselhos básicos.

Generalizando conceitos significativos

Sempre procure generalizar o máximo possível os conceitos que estão significativamente relacionados. Como já mencionei algumas vezes neste livro, quando as entidades em seu programa estiverem relacionadas, é melhor que elas estejam concretamente relacionadas por meio de construções reais no programa. Nossas abas não seguem esse conselho: elas têm muito em comum, porém

a lógica e a estrutura estão espalhadas em vários locais. Por exemplo, nós definimos o nome da aba no HTML:

```
<div class="tabs">
  <a href=""><span class="active">Newest</span></a>
  <a href=""><span>Oldest</span></a>
  <a href=""><span>Tags</span></a>
  <a href=""><span>Add</span></a>
</div>
```

E no JavaScript, utilizamos a localização da aba no DOM para determinar a ação a ser tomada:

```
if ($element.parent().is(":nth-child(1)")) {
  // gera o conteúdo da aba 'Newest'
} else if ($element.parent().is(":nth-child(2)")) {
  // gera o conteúdo da aba 'Oldest'
} else if ($element.parent().is(":nth-child(3)")) {
  // gera o conteúdo da aba 'Tags'
} else if ($element.parent().is(":nth-child(4)")) {
  // gera o conteúdo da aba 'Adds'
}
```

Observe que jamais associamos realmente um nome de aba a uma ação. Esse é um exemplo de duas entidades de programação que estão relacionadas, porém somente na mente do programador – e não em alguma construção presente no código. Um sintoma bem básico disso é que, sempre que acrescentarmos uma aba à nossa interface de usuário, será preciso modificar tanto o HTML quanto o JavaScript. No capítulo 5, salientei que esse fato torna o nosso código suscetível a erros.

Como isso pode ser melhorado? Um curso de ação consiste em generalizar uma aba para que ela seja um objeto, exatamente como fizemos no exemplo da carta no capítulo 5. A diferença é que o objeto aba terá uma string associada ao seu nome da aba e uma função associada à ação, a qual criará o conteúdo da aba. Por exemplo, podemos generalizar a nossa aba Newest na forma de um objeto desta maneira:

```
var newestTab = {
```

```

// o nome desta aba
"name":"Newest",

// a função que gera o conteúdo desta aba
"content": function () {
    var $content;

    // gera o conteúdo da aba "Newest" e o retorna
    $content = $("<ul>");
    for (i = toDos.length-1; i >= 0; i--) {
        $content.append($("<li>").text(toDos[i]));
    }

    return $content;
}
}

```

O fato é que, depois que tivermos essa estrutura, a maior parte do problema terá sido resolvida. Nossa interface de usuário é constituída de um conjunto de abas, portanto, em vez de ter o nome da aba e a ação correspondente armazenados em locais separados de nosso código, podemos criar um array de objetos aba que os mantêm reunidos:

```

var main = function (todoObjects) {
    "use strict";

    var toDos,
        tabs;

    toDos = todoObjects.map(function (todo) {
        return todo.description;
    });

    // cria um array vazio de abas
    tabs = [];

    // adiciona a aba 'Newest'
    tabs.push({
        "name":"Newest",
        "content":function () {
            // cria $content para a aba 'Newest'
            return $content;
        }
    });

    // adiciona a aba 'Oldest'

```

```

tabs.push({
  "name": "Oldest",
  "content": function () {
    // cria $content para a aba 'Oldest'
    return $content;
  }
});

// adiciona a aba 'Tags'
tabs.push({
  "name": "Tags",
  "content": function () {
    // cria $content para a aba 'Tags'
    return $content;
  }
});

// adiciona a aba 'Add'
tabs.push({
  "name": "Add",
  "content": function () {
    // cria $content para a aba 'Add'
    return $content;
  }
});
};

```

Depois que tivermos o array que armazena os objetos aba, podemos simplificar enormemente a nossa abordagem de criação da UI. Começaremos pela total remoção das abas do HTML (mas deixaremos a div):

```

<div class="tabs">
  <!-- é aqui que nossas abas costumavam ser definidas -->
  <!-- nós as criaremos dinamicamente no JavaScript -->
</div>

```

A seguir, podemos percorrer o nosso array `tabs` usando um laço. Para cada item, definiremos o nosso handler de cliques e adicionaremos a aba ao DOM:

```

tabs.forEach(function (tab) {
  var $aElement = $("<a>").attr("href", ""),
    $spanElement = $("<span>").text(tab.name);

```

```

$element.append($spanElement);
$spanElement.on("click", function () {
    var $content;

    $(".tabs a span").removeClass("active");
    $spanElement.addClass("active");
    $(".main .content").empty();

    // neste ponto, obtemos o conteúdo da
    // função definida no objeto aba
    $content = tab.content();

    $(".main .content").append($content);
    return false;
});
});

```

Incluindo o AJAX em nossas abas

Outro problema que nossa aplicação apresenta no momento é que, se alguém acessá-la por meio de outro navegador e adicionar um item à nossa lista de tarefas, não veremos esse novo item se clicarmos em uma aba diferente. Na realidade, será necessário recarregar toda a página para vermos a atualização.

Isso ocorre porque nossa aplicação está armazenando e carregando os itens da lista de tarefas quando a página é inicialmente carregada e não os altera até a página ser carregada novamente. Para resolver esse problema, faremos com que cada aba faça uma solicitação AJAX quando for carregada. Isso pode ser feito por meio do encapsulamento de nossa ação em chamadas à função `get` da jQuery:

```

tabs.push({
    "name": "Newest",
    "content": function () {
        $.get("todos.json", function (todoObjects) {
            // cria $content para a aba 'Newest'

            // não podemos mais dar um 'return' em $content
        });
    }
});

```



Outra solução é fazer o nosso programa efetuar atualizações em tempo real no cliente, o que significa que, quando outro usuário atualizar a página que estivermos olhando, o servidor deverá enviar os dados novos para nós. Isso está um pouco além do escopo deste livro, porém pode ser feito por meio de um módulo do Node.js chamado *socket.io*.

Observe que isso muda o comportamento de nossas funções porque agora faremos uma solicitação assíncrona dentro de nossa chamada à função. Isso quer dizer que esse código não retornará mais o conteúdo a quem chamou de forma adequada, pois teremos de esperar até que a chamada AJAX se complete:

```
$content = tab.content();  
$("#main .content").append($content);
```

Temos algumas opções para corrigir esse problema. A solução mais simples é transferir a atualização do DOM para a própria função `content`:

```
tabs.push({  
  "name": "Newest",  
  "content": function () {  
    $.get("todos.json", function (todoObjects) {  
      // cria $content para a aba 'Newest'  
      // atualiza o DOM aqui  
      $("#main .content").append($content);  
    });  
  }  
});
```

Não gosto dessa abordagem por dois motivos. O primeiro está relacionado a uma razão de caráter estético: a função `content` deve criar e retornar o conteúdo da aba – ela não deve afetar o DOM. Se fosse assim, devíamos chamá-la de `getContentAndUpdateTheDOM`.

O segundo motivo provavelmente é um pouco mais importante: se quisermos fazer mais do que simplesmente atualizar o DOM no final, será necessário adicionar essa lógica a toda função `content` de todas as abas.

Uma solução para ambos os problemas consiste em implementar a abordagem de continuidade que usamos anteriormente para as

operações assíncronas. Vamos deixar que a função que faz a chamada inclua uma callback e chamá-la dentro de nossa função `content`:

```
// criamos nossa função content
// de modo que ela aceite uma callback
tabs.push({
  "name": "Newest",
  "content": function (callback) {
    $.get("todos.json", function (todoObjects) {
      // cria $content para a aba 'Newest'
      // chama a callback com $content
      callback($content);
    });
  }
});
// ...

// agora, na função que faz a chamada, enviamos uma callback
tab.content(function ($content) {
  $("main .content").append($content);
});
```

Essa solução provavelmente é a abordagem mais comum que você encontrará na comunidade Node.js, porém outras abordagens estão se tornando populares, incluindo o Promises e o Reactive JavaScript. Se suas operações assíncronas começarem a ficar complicadas e você se vir em (algo que é comumente chamado de) um *inferno de callbacks*, pode ser que você queira explorar essas opções.

Livrando-se dos hacks

Agora que incluímos o *AJAX* em todas as abas, podemos nos livrar de um hack de compatibilidade que permanece presente no código. Anteriormente, fizemos o nosso servidor retornar toda a lista de objetos `ToDo` porque era isso o que o nosso cliente esperava sempre que uma adição era efetuada. Agora, em vez de fazer isso, voltaremos para a aba *Newest* sempre que um objeto `ToDo` for

adicionado.

O código de tratamento do botão da aba de adição de um novo item no momento tem o seguinte aspecto:

```
$button.on("click", function () {  
    var description = $input.val(),  
        tags = $tagInput.val().split(","),  
        newToDo = {"description":description, "tags":tags};  
  
    $.post("todos", newToDo, function (result) {  
        // aqui está um pouco de código remanescente de quando  
        // estávamos mantendo todos os itens da lista de tarefas no cliente  
        toDoObjects = result;  
  
        // atualiza os ToDos do cliente  
        toDos = toDoObjects.map(function (toDo) {  
            return toDo.description;  
        });  
  
        $input.val("");  
        $tagInput.val("");  
    });  
});
```

No momento, não precisamos mais manter os `ToDos` no cliente, portanto podemos nos livrar da maior parte desse código. Com efeito, tudo o que realmente precisamos fazer depois que o novo `ToDo` for enviado com sucesso é limpar as caixas de entrada e, em seguida, fazer o *redirecionamento* para a aba `Newest`. Nossa solicitação `AJAX` fará isso e os resultados serão ordenados de acordo com o item mais recente.

Podemos utilizar a função `trigger` da `jQuery` para disparar o evento clique na aba `Newest`, e nosso código, no final, terá o seguinte aspecto:

```
$button.on("click", function () {  
    var description = $input.val(),  
        tags = $tagInput.val().split(","),  
        newToDo = {"description":description, "tags":tags};  
  
    $.post("todos", newToDo, function (result) {  
        // limpa as nossas caixas de entrada  
        $input.val("");  
    });  
});
```

```
    $tagInput.val("");  
    // dispara 'click' na aba Newest  
    $(".tabs a:first span").trigger("click");  
  });  
});
```

Essa pequena alteração também permite simplificar enormemente o código do lado do servidor, que cria um novo objeto `ToDo` em um post:

```
app.post("/todos", function (req, res) {  
  var newToDo = new ToDo({"description":req.body.description,  
    "tags":req.body.tags});  
  newToDo.save(function (err, result) {  
    console.log(result);  
    if (err !== null) {  
      // o elemento não foi salvo!  
      console.log(err);  
      res.json(err);  
    } else {  
      res.json(result);  
    }  
  });  
});
```

Tratando erros do AJAX

Boa parte dos problemas relacionados ao tratamento de erros em todo o nosso código até agora foi ignorada, porém, ao começar a criar aplicações maiores, você perceberá que pensar nos erros é essencial. Há ocasiões em que o código de seu cliente será carregado, porém o servidor estará indisponível (ou causará falhas inadvertidamente). O que acontecerá nesse caso?

Em várias situações, nossas condições de erro terão como consequência o fato de nossas callbacks não serem chamadas corretamente. Quando isso acontecer, vai parecer que a aplicação não está respondendo. Pior ainda, o nosso código poderá responder adicionando `ToDo`s no cliente, porém eles não serão adicionados corretamente no sistema de armazenamento de dados. Isso pode

fazer com que nossos usuários percam dados que eles acharam que foram adicionados com sucesso.

Felizmente, a jQuery permite facilmente que levemos esse cenário em consideração ao utilizarmos a função `fail`, que pode ser encadeada a partir de uma chamada AJAX. Esse é um exemplo de uma API do tipo Promise, mencionada na seção anterior, porém não ofereceremos muitos detalhes a respeito do que isso quer dizer. Acho que a melhor maneira de lidar com essa situação é seguir o padrão que vimos tanto no módulo Mongoose quanto no Redis para o Node.js. Deixaremos que o código que efetuar a chamada envie uma callback que aceite um erro e os dados propriamente ditos, e deixaremos a callback lidar com o erro, caso ele não esteja definido com `null`:

```
// criamos a nossa função content
// de modo que ela aceite uma callback
tabs.push({
  "name": "Newest",
  "content": function (callback) {
    $.get("todos.json", function (todoObjects) {
      // cria $content para a aba 'Newest'

      // chama a callback com o erro definido com null e $content
      // como o segundo argumento
      callback(null, $content);
    }).fail(function (jqXHR, textStatus, error) {
      // nesse caso, enviaremos o erro juntamente
      // com null para $content
      callback(error, null);
    });
  }
});
```

A callback da função `fail` aceita três argumentos. O erro é o argumento no qual estamos mais interessados e é o que passaremos para a callback da função que fez a chamada.

Na callback da função que fez a chamada, trataremos o erro exatamente como fizemos nos exemplos com o Redis e o Mongoose:

```
tab.content(function (err, $content) {  
  if (err !== null) {  
    alert("Whoops, there was a problem with your request: " + err);  
  } else {  
    $("main .content").append($content);  
  }  
});
```

Podemos testar esse comportamento alterando a rota em nossa chamada a `get` para uma rota inexistente (diferente de *todos.json*). A mensagem de erro nesse caso deverá ser igual a “Not Found” (Não encontrado).

Efetuando a refatoração de nosso servidor

Vimos algumas dicas sobre como refatorar o código de nosso cliente. Podemos aplicar facilmente essas mesmas dicas, porém há algumas considerações adicionais a serem feitas no servidor. Nesta seção, aprenderemos a organizar o código de nosso servidor utilizando o padrão de projeto *Model-View-Controller* (Modelo-Visão-Controlador).

Organização do código

Nesse momento, todo o código do lado do servidor está em um único arquivo: *server.js*. Isso não é complicado demais para uma aplicação pequena como a nossa, porém, à medida que nossa aplicação crescer e incluir outras entidades diferentes dos objetos `ToDo`, esse código poderá se tornar rapidamente difícil de administrar. Portanto vamos efetuar algumas separações.

Separando as responsabilidades: os modelos

Começaremos removendo a definição do modelo do Mongoose que está em *server.js* e a reorganizaremos em um módulo Node.js independente. Gosto de colocar as definições dos modelos em seu próprio diretório porque, à medida que minha aplicação crescer, a quantidade de modelos criados também aumentará. Sendo assim,

criaremos um arquivo chamado *todo.js* em um diretório de nome *models*, que fica dentro de nosso diretório *Amazeriffic*. Nesse arquivo, definiremos o meu modelo exatamente como foi feito anteriormente e o exportaremos:

```
var mongoose = require("mongoose");  
  
// Este é o nosso modelo do mongoose para as tarefas da lista  
var TodoSchema = mongoose.Schema({  
  description: String,  
  tags: [ String ]  
});  
  
var Todo = mongoose.model("ToDo", TodoSchema);  
  
module.exports = Todo;
```

Agora podemos fazer o `require` do módulo em *server.js* e remover o código de `ToDo` que está no arquivo:

```
var express = require("express"),  
    http = require("http"),  
    mongoose = require("mongoose"),  
    // importa o nosso modelo ToDo  
    ToDo = require("../models/todo.js"),  
    app = express();
```

Se executarmos o nosso servidor agora, tudo deverá funcionar exatamente como antes.

Separando as responsabilidades: os controladores

Transferir os modelos para o seu próprio diretório nos permite manter mais facilmente uma separação clara das responsabilidades em nosso programa. Sabemos que, quando for necessário alterar o modo como os nossos dados `ToDo` são armazenados no banco de dados, poderemos alterar o arquivo *todo.js* que está no diretório *models*. De modo semelhante, quando for necessário alterar o modo como o nosso programa responde às solicitações do cliente, iremos alterar um arquivo *controlador* associado.

Atualmente, quando nossas solicitações `get` e `post` chegam ao nosso servidor Express, respondemos com funções anônimas. Vamos dar nomes a essas funções anônimas e transferi-las para um módulo

separado. Esse módulo será constituído de um único objeto controlador que tem um conjunto de *ações* a serem disparadas por meio de nossas rotas Express. No caso de nossos objetos `ToDo`, criaremos duas ações: `index` e `create`.

Para isso, criaremos um diretório chamado *controllers*, que ficará ao lado de nosso diretório *models*, e, nesse diretório, criaremos um arquivo chamado *todos_controller.js*. Nesse módulo, importaremos o nosso módulo `ToDo` e criaremos um objeto `ToDoController`. Associaremos funções que fazem o mesmo que as funções anônimas faziam em nosso arquivo *server.js*:

```
// observe que devemos especificar um diretório
// acima para encontrar o nosso diretório models
var ToDo = require("../models/todo.js"),
    ToDoController = {};

ToDoController.index = function (req, res) {
  ToDo.find({}, function (err, todos) {
    res.json(todos);
  });
};

ToDoController.create = function (req, res) {
  var newToDo = new ToDo({ "description": req.body.description,
    "tags": req.body.tags });
  newToDo.save(function (err, result) {
    console.log(result);
    if (err !== null) {
      // o elemento não foi salvo!
      console.log(err);
      res.json(500, err);
    } else {
      res.json(200, result);
    }
  });
};

module.exports = ToDoController;
```

Do mesmo modo que fizemos com o nosso modelo, importaremos esse módulo no código de *server.js* utilizando `require`. Feito isso,

atualizaremos as ações de nossa rota para que apontem para essas funções, em vez de ter funções anônimas. Observe que, como não havia mais necessidade de acessar o modelo `ToDo` em `server.js`, removemos a instrução `require` relacionada ao modelo:

```
var express = require("express"),
    http = require("http"),
    mongoose = require("mongoose"),
    // importa o nosso controlador de Todos
    TodosController = require("./controllers/todos_controller.js"),
    app = express();

// demais código para configuração/Cloud Foundry/mongoose devem ser
// inseridos aqui ...

// rotas
app.get("/todos.json", TodosController.index);
app.post("/todos", TodosController.create);
```

Agora o nosso código está um pouco mais organizado e mais fácil de ser mantido porque separamos as responsabilidades. O nosso arquivo `server.js`, em sua maior parte, cuida da configuração básica do servidor e do roteamento, o nosso controlador cuida das ações que devem ocorrer quando uma solicitação chegar e o nosso modelo cuida das questões que envolvem o banco de dados. Essa separação de responsabilidades torna muito mais fácil a manutenção de nosso código à medida que ele crescer, além de facilitar bastante o mapeamento das solicitações HTTP de nosso cliente para as ações em nosso servidor.

Verbos HTTP, CRUD e REST

No capítulo 6, discutimos brevemente o protocolo HTTP. Em seções subsequentes, aprendemos que podemos fazer dois tipos de solicitação HTTP: solicitações GET e solicitações POST. Essas solicitações correspondem às rotas `get` e `post` em nosso servidor Express que, por sua vez, correspondem às ações definidas em nosso controlador. O fato é que o HTTP também tem dois outros verbos que ainda não utilizamos: PUT e DELETE. Esses quatro

verbos mapeiam-se muito bem às operações CRUD para armazenamento de dados, sobre as quais aprendemos no capítulo 7 (veja a tabela 9.1)!

Esse mapeamento permite criar APIs que oferecem uma interface clara e simples aos recursos disponíveis em nosso servidor. As APIs que se comportam dessa maneira normalmente são chamadas de APIs web *RESTful*. REST quer dizer *Representational State Transfer* (Transferência de Estado Representativo) e – falando de modo bem geral – é uma ideia que descreve de que modo os recursos em servidores web devem ser expostos por meio do protocolo HTTP.

Tabela 9.1 – Mapeamentos entre HTTP/CRUD/controlador

HTTP	CRUD	Ação	Comportamento
POST	Create (Criar)	create	Cria um novo objeto e retorna o seu ID
GET	Read (Ler)	show	Retorna um objeto que tem um determinado ID
PUT	Update (Atualizar)	update	Atualiza um objeto que tem um determinado ID
DELETE	Delete (Apagar)	destroy	Apaga um objeto que tem um determinado ID

Definindo rotas de acordo com o ID

Um recurso interessante oferecido pelo Express é a capacidade de criar variáveis em nossas rotas. Isso permite criar uma única regra que responda a todo um conjunto de solicitações. Por exemplo, suponha que queremos obter um único item da lista de tarefas utilizando o ID associado pelo MongoDB:

```
// rotas
app.get("/todos.json", TodosController.index);

// rotas CRUD básicas
app.get("/todos/:id", TodosController.show);
app.post("/todos", TodosController.create);
```

Nesse caso, criamos a nossa rota `get` que faz o mapeamento para um único item da lista de tarefas. O mapeamento é feito para a

função `show` de nosso controlador. Observe que a rota utiliza dois-pontos (`:`) para criar uma variável. Isso permite que a rota responda a *qualquer* rota que iniciar com `/todos/`. Portanto, se configurarmos o nosso navegador com `/todos/helloworld`, a solicitação será enviada para a ação `show` de nosso controlador. É a ação do controlador que será responsável por encontrar o elemento que tenha um ID igual a `helloworld`.

Como podemos acessar a variável enviada ao controlador? O fato é que o objeto referente à solicitação mantém essas informações em sua propriedade `params`. Nosso código para a ação pode ter a aspecto a seguir, em que efetuamos uma query no modelo e retornamos a resposta. Se o ID não for encontrado, retornaremos uma string que contém “Not Found” (Não encontrado):

```
TodosController.show = function (req, res) {  
  // este é o id que é enviado ao URL  
  var id = req.params.id;  
  // encontra o item Todo com o id associado  
  Todo.find({"_id":id}, function (err, todo) {  
    if (err !== null) {  
      res.json(err);  
    } else {  
      if (todo.length > 0) {  
        res.json(todo[0]);  
      } else {  
        res.send("Not Found");  
      }  
    }  
  });  
};
```

Supondo que já tenhamos inserido alguns itens na lista de tarefas em nosso banco de dados, podemos verificar se esse código está funcionando se iniciarmos o cliente do MongoDB a partir da linha de comando e obtivermos um ID:

```
vagrant $ mongo  
MongoDB shell version: 2.4.7  
connecting to: test
```

```

> show dbs
amazeriffic 0.0625GB
local 0.03125GB
> use amazeriffic
switched to db amazeriffic
> show collections
system.indexes
todos
> db.todos.find()
{ "description" : "first", "_id" : ObjectId("5275643e0cff128714000001"), ... }
{ "description" : "second", "_id" : ObjectId("52756de289f2f5f014000001"), ... }
{ "description" : "test", "_id" : ObjectId("5275722a8d735d0015000001"), ... }
{ "description" : "hello", "_id" : ObjectId("5275cbdc408d04c150000001"), ... }

```

Agora que podemos ver o campo `_id`, podemos executar o servidor, abrir o Chrome e experimentar digitar algo como *<http://localhost:3000/5275643e0cff128714000001>* na barra de endereço. Se tudo funcionar corretamente, devemos ver o JSON retornado pelo nosso servidor!



Se você digitar um ID de objeto inválido (o que significa uma string que não é constituída de 24 números ou letras de a até f), o código anterior irá gerar um erro que informa que um ID inválido foi utilizado. Não há problemas nisso, por enquanto – faremos a correção em “Práticas e leituras adicionais”.

Utilizando a jQuery em solicitações put e delete

Além de efetuar solicitações `get` e `post`, a jQuery é capaz de fazer solicitações `put` e `delete` por meio da função `$.ajax` genérica. Com efeito, as solicitações `get` e `post` também podem ser feitas desta maneira:

```

// exemplo de PUT com a jQuery
// Neste caso, atualizaremos a descrição do
// objeto ToDo cujo id é igual a 1234
$.ajax({
  "url" : "todos/1234",
  "type": "PUT",
  "data": {"description": "this is the new description"},
}).done(function (response) {

```

```

    // sucesso!
  }).fail(function (err) {
    // erro!
  });
// exemplo de DELETE com a jQuery
// Vamos apagar o objeto ToDo cujo id é igual a 1234
$.ajax({
  "url" : "todos/1234",
  "type": "DELETE",
}).done(function (response) {
  // sucesso!
}).fail(function (err) {
  // erro!
});

```

Podemos associar facilmente essas ações a eventos de cliques de botão para atualizar ou para apagar objetos em nosso banco de dados a partir do cliente. Para isso, estamos supondo que temos as rotas associadas expostas no servidor.

Códigos de resposta HTTP

Além de o HTTP incluir uma série de verbos que são muito bem mapeados às operações CRUD, ele também define um conjunto de códigos-padrão de resposta que representa as respostas possíveis a uma solicitação HTTP. Você já deve ter tentado acessar um site ou uma página inexistente e recebido o infame erro 404, que significa que a página não foi encontrada. O fato é que 404 é um dos códigos de resposta definidos pelo protocolo HTTP.

Outros códigos de resposta HTTP incluem 200, que significa que a solicitação foi bem-sucedida, e 500, que representa um erro interno genérico do servidor. O Express permite enviar esses valores de volta a um cliente, juntamente com o valor da resposta. Portanto, para tornar nossa rota `show` um pouco mais robusta, podemos incluir os códigos de resposta HTTP apropriados, juntamente com os dados que estivermos enviando:

```

TodosController.show = function (req, res) {

```

```

// este é o id que é enviado ao URL
var id = req.params.id;

ToDo.find({"_id":id}, function (err, todo) {
  if (err !== null) {
    // retornamos um erro interno de servidor
    res.json(500, err);
  } else {
    if (todo.length > 0) {
      // retornamos sucesso!
      res.json(200, todo[0]);
    } else {
      // não encontramos o item da lista de tarefas com esse id
      res.send(404);
    }
  }
});
};

```

Se o servidor for executado agora e um id de objeto inválido for digitado, você verá que o navegador Chrome saberá responder automaticamente com uma mensagem “Not Found” (Não encontrado) quando um código de resposta 404 for recebido por meio do HTTP.

Model-View-Controller

Isso nos leva a um dos mais importantes conceitos em todo o desenvolvimento de aplicações web – o padrão de projeto MVC (Model-View-Controller, ou Modelo-Visão-Controlador). Essa é uma abordagem utilizada na arquitetura de aplicações que, em sua maior parte, determina o design de uma aplicação baseada em banco de dados e tornou-se um padrão de mercado para a criação de aplicações web. O padrão é tão amplamente aceito como a melhor maneira de criar aplicações web que passou a ser incluído nos frameworks mais populares para aplicações web.

Já efetuamos a refatoração do código do Amazeriffic para que se enquadrasse nesse padrão, porém é uma boa ideia dar um passo para trás e compreender a responsabilidade de cada um dos

componentes e como as partes da aplicação se encaixam.

O *controlador* (controller) normalmente é a parte mais simples entre as três. Quando um navegador web faz uma solicitação HTTP ao servidor, o roteador entrega a solicitação a uma ação do controlador associado. O controlador traduz a solicitação em uma ação que normalmente coordena uma ação do banco de dados por meio do modelo e, em seguida, envia (ou renderiza) a resposta por meio de uma visão (view). Isso pode ser visto na ação `show` – o controlador encontra o item da lista de tarefas que tem o ID solicitado (se houver) e gera o objeto relativo à visão (nesse caso, o JSON), a ser enviado na resposta.

O *modelo* (model) é a abstração, na forma de objetos, dos elementos de nosso banco de dados. Felizmente, a maioria dos frameworks inclui um modelador de objetos como o Mongoose. No Rails, o modelador default chama-se *Active Record* e, de modo geral, se comporta do mesmo modo que o Mongoose. Em nosso caso, o modelo `ToDo` é totalmente constituído de uma definição de esquema, porém, normalmente, nossa definição de modelo pode incluir muito mais do que isso. Por exemplo, ela pode definir relacionamentos com outros modelos e pode especificar funções que serão disparadas quando determinados aspectos do modelo forem alterados.

Por fim, temos o componente referente à *visão* (view). Em nosso caso, a visão pode ser pensada como o HTML e o CSS do lado do cliente. O controlador simplesmente envia os dados como JSON ao cliente e esse decide em que local deve colocá-los. No entanto as visões podem facilmente se tornar muito mais interessantes. Por exemplo, a maioria dos frameworks MVC inclui templates do lado do servidor que permitem ao controlador criar o HTML no momento da solicitação. No mundo JavaScript/Node.js, o *Jade* e o *EJS* são dois engines de templating HTML comumente utilizados.

Falando de modo mais geral, a aplicação funciona basicamente desta maneira: o cliente solicita um recurso por meio de um verbo

HTTP e uma rota. O *roteador* (em nosso caso, o *server.js*) decide qual é o controlador e a ação para os quais a solicitação será enviada. O controlador então usa a solicitação para interagir com o modelo de alguma maneira e decide como a visão deve ser construída. Feito isso, ele envia uma resposta à solicitação. A figura 9.1 sintetiza tudo isso!

Vamos ver tudo isso em ação. Adicionaremos usuários à nossa aplicação para que cada um possa ter o seu próprio conjunto de Todos.

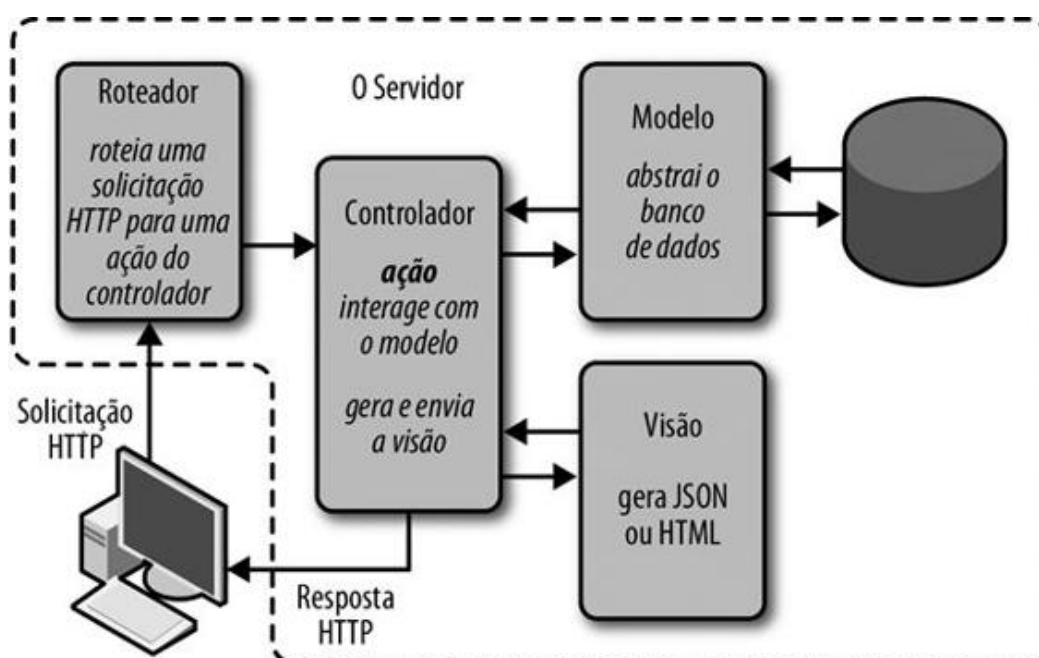


Figura 9.1 – Padrão Model-View-Controller (Modelo-Visão-Controlador).

Adicionando usuários ao Amazeriffic

A essa altura, o Amazeriffic tem exatamente uma entidade: os objetos `ToDo`. Não haverá problemas se nós formos os únicos a querer manter um registro dos objetos `ToDo`, porém jamais teremos um negócio de 1 bilhão de dólares dessa maneira. Então o que podemos fazer se quisermos que uma quantidade arbitrária de pessoas utilize nossa aplicação para manter suas listas de tarefas? O fato é que precisamos adicionar outra entidade à nossa aplicação:

os usuários.

Criando o modelo para os usuários

Vamos começar pela criação de um modelo para os nossos usuários. O modelo `User` será composto de uma string que representa o nome do usuário e o ID de objeto default do MongoDB. Associaremos um ID de usuário aos objetos `ToDo` que pertençam a esse usuário e configuraremos rotas que retornem somente os objetos `ToDo` associados a ele.

Ao utilizar o Mongoose, definir um modelo simples como esse é realmente muito fácil. Vamos criar um arquivo chamado *user.js* em nosso diretório *models*. Nesse arquivo, criaremos um esquema e um modelo Mongoose. E, como antes, exportaremos o modelo Mongoose para o programa Node.js, que o utilizará por meio de uma instrução `require`:

```
var mongoose = require("mongoose");  
// Este é o nosso modelo mongoose para os usuários  
var UserSchema = mongoose.Schema({  
  username: String,  
});  
var User = mongoose.model("User", UserSchema);  
module.exports = User;
```

Pode ser que faça sentido incluir também um array de objetos `ToDo`, porém vamos manter a simplicidade por enquanto. Logo mais, definiremos um relacionamento entre os objetos `ToDo` e os objetos `User`.

Criando o controlador para os usuários

A seguir, criaremos os *stubs* do controlador para os nossos usuários. Isso significa que criaremos funções placeholder vazias que serão posteriormente completadas à medida que precisarmos delas. Definiremos uma ação para cada operação CRUD, juntamente com uma ação `index` que retornará uma lista de todas os

objetos User:

```
var User = require("../models/user.js"),
    mongoose = require("mongoose");

var UsersController = {};

UsersController.index = function (req, res) {
  console.log("index action called");
  res.send(200);
};

// Mostra um usuário
UsersController.show = function (req, res) {
  console.log("show action called");
  res.send(200);
};

// Cria um novo usuário
UsersController.create = function (req, res) {
  console.log("create action called");
  res.send(200);
};

// Atualiza um usuário existente
UsersController.update = function (req, res) {
  console.log("update action called");
  res.send(200);
};

// Apaga um usuário existente
UsersController.destroy = function (req, res) {
  console.log("destroy action called");
  res.send(200);
};

module.exports = UsersController;
```

Espero que você possa tornar essas ações genéricas com base no exemplo com `ToDoController`. Em “Práticas e leituras adicionais”, você criará uma interface de usuário que permitirá interagir com o modelo `User` da mesma maneira que interagimos anteriormente com o modelo `ToDo`.

Por enquanto, porém, criaremos um usuário de exemplo com o qual trabalharemos nas seções seguintes, por meio da adição do código

a seguir no início do controlador de usuários:

```
// Este código verifica se um usuário já existe
User.find({}, function (err, result) {
  if (err !== null) {
    console.log("SOMETHING WENT HORRIBLY WRONG");
    console.log(err);
  } else if (result.length === 0) {
    console.log("Creating Example User...");
    var exampleUser = new User({"username": "semmy"});
    exampleUser.save(function (err, result) {
      if (err) {
        console.log(err);
      } else {
        console.log("Saved Example User");
      }
    });
  }
});
```

Utilizaremos esse usuário de exemplo para testar nossas rotas. Você pode criar outros usuários de teste ou default ao seguir esse mesmo formato básico.

Definindo as rotas

Nesse exemplo, é nas rotas que a situação começa a ficar interessante. Inicialmente, podemos definir o nosso mapeamento básico de HTTP/ação para o nosso modelo `User` como fizemos anteriormente. Abra o arquivo `server.js` e acrescente as rotas básicas a seguir, que mapeiam as solicitações HTTP às ações do controlador:

```
app.get("/users.json", usersController.index);
app.post("/users", usersController.create);
app.get("/users/:username", usersController.show);
app.put("/users/:username", usersController.update);
app.del("/users/:username", usersController.destroy);
```



Observe que o Express utiliza `del` em vez de `delete` para solicitações HTTP DELETE. Isso ocorre porque `delete` tem um significado diferente no JavaScript.

Agora queremos configurar a nossa aplicação para que tenhamos as rotas e os comportamentos mostrados na tabela 9.2.

Tabela 9.2 – Rotas/Comportamentos

Verbo	Rota	Comportamento
GET	/users/semmy/	Mostra minha página do Amazeriffic
GET	/users/semmy/todos.json	Obtém todos os meus ToDos na forma de um array
POST	/users/semmy/todos	Cria um novo ToDo para mim
PUT	/users/semmy/todos/[id]	Atualiza o meu ToDo cujo ID está especificado
DELETE	/users/semmy/todos/[id]	Apaga o meu ToDo cujo ID está especificado

Você verá que as rotas se comportam quase exatamente do mesmo modo que as rotas `ToDo` existentes, exceto pelo fato de elas estarem associadas a um determinado `username`. Como podemos fazer isso com o código existente? Simplesmente iremos configurar outro conjunto de rotas que inclua o prefixo `users/:username` e as apontaremos para as ações de `ToDoController`:

```
app.get("/users/:username/todos.json", toDosController.index);
app.post("/users/:username/todos", toDosController.create);
app.put("/users/:username/todos/:id", toDosController.update);
app.del("/users/:username/todos/:id", toDosController.destroy);
```

Agora que temos um roteamento básico definido, podemos começar a integrar nossas alterações. Em primeiro lugar, queremos enviar o nosso arquivo *index.html* do lado do cliente quando solicitarmos a página do usuário, em vez de criar uma página diferente. Podemos fazer isso facilmente modificando a nossa ação `show`. Queremos que a nossa ação `show` envie uma `view` para esse usuário. Felizmente, a visão do usuário será igual à visão `default`, portanto podemos simplesmente enviar o nosso arquivo *index.html* utilizando a função `sendfile` de resposta do Express:

```
UsersController.show = function (req, res) {
  console.log("show action called");
  // envia o arquivo HTML básico que representa a visão
```

```
    res.sendFile("./client/index.html");
  };
```

Se executarmos o servidor agora e acessarmos *localhost:3000/users/semmy/*, deveremos ver a mesma interface que vimos anteriormente, incluindo qualquer objeto `ToDo` que estiver em nosso sistema de armazenamento de dados no momento. Isso ocorre porque ainda não fizemos a restrição ao subconjunto de `ToDos` que pertencem a um usuário.

No entanto, temos um problema. Abra o seu navegador e acesse *localhost:3000/users/hello/*. Veremos o mesmo e *hello* não é realmente um usuário! Queremos retornar 404 quando a rota apontar para um usuário inválido. Isso pode ser feito por meio de uma query no modelo de usuários usando o nome do usuário enviado como parâmetro. Desse modo, nossa ação `show` será alterada para:

```
UserController.show = function (req, res) {
  console.log("show action called");
  User.find({"username": req.params.username}, function (err, result) {
    if (err) {
      console.log(err);
      res.send(500, err);
    } else if (result.length !== 0) {
      // encontramos um usuário
      res.sendFile("./client/index.html");
    } else {
      // não há nenhum usuário com esse nome,
      // portanto enviamos 404
      res.send(404);
    }
  });
};
```

Agora devemos obter respostas para essa rota somente quando um usuário válido for solicitado.

Aperfeiçoando as ações de nosso controlador de ToDos

Antes de prosseguir, precisamos ter uma maneira de associar o nosso modelo `User` ao modelo `ToDo`. Para isso, modificaremos o nosso modelo `ToDo` para que cada objeto desse tenha um proprietário. O proprietário será representado por um `ObjectId` na coleção de `ToDo`s que irá referenciar um elemento correspondente a um usuário na coleção de `User`s. Em termos clássicos de banco de dados, basicamente, isso é o mesmo que adicionar uma *chave estrangeira* que relacione uma determinada tabela a outra tabela diferente.

Para configurar isso, modificaremos o modelo e o esquema de `ToDo` para que incluam a referência ao modelo `User`:

```
var mongoose = require("mongoose"),
    ToDoSchema,
    ObjectId = mongoose.Schema.Types.ObjectId;

ToDoSchema = mongoose.Schema({
  description: String,
  tags: [ String ],
  owner : { type: ObjectId, ref: "User" }
});

module.exports.ToDo = mongoose.model("ToDo", ToDoSchema);
```

Ao criarmos um objeto `ToDo`, também incluiremos um proprietário ou o valor `null`, caso o proprietário não exista (no caso em que adicionarmos um `ToDo` a partir da rota default, por exemplo).

A seguir, podemos começar a trabalhar nas ações do controlador de `ToDo` para que elas levem em consideração a possibilidade de que podem ser chamadas por meio de uma rota associada a um nome de usuário. Para começar, modificaremos a nossa ação `index` para que ela responda com os `ToDo`s de um usuário, caso esse esteja definido:

```
ToDosController.index = function (req, res) {
  var username = req.params.username || null,
      respondWithTodos;

  // uma função auxiliar que obtém os Todos
  // de acordo com uma query
  respondWithTodos = function (query) {
```

```

    Todo.find(query, function (err, todos) {
      if (err !== null) {
        res.json(500,err);
      } else {
        res.json(200,todos);
      }
    });
  };

  if (username !== null) {
    // obtém os Todos associados a um nome de usuário
    User.find({"username":username}, function (err, result) {
      if (err !== null) {
        res.json(500, err);
      } else if (result.length === 0) {
        // nenhum usuário com esse id foi encontrado!
        res.send(404);
      } else {
        // responde com os objetos Todo desse usuário
        respondWithTodos({ "owner" : result[0].id });
      }
    });
  } else {
    // responde com todos os Todos
    respondWithTodos({});
  }
};

```

Por fim, modificaremos nossa ação `create` para que ela adicione o usuário ao novo `Todo` se o usuário estiver definido:

```

TodosController.create = function (req, res) {
  var username = req.params.username || null,
      newTodo = new Todo({"description":req.body.description,
                          "tags":req.body.tags});

  User.find({"username":username}, function (err, result) {
    if (err) {
      res.send(500);
    } else {
      if (result.length === 0) {
        // o usuário não foi encontrado, portanto
        // simplesmente criamos um Todo sem proprietário

```

```

        newToDo.owner = null;
    } else {
        // um usuário foi encontrado, portanto
        // definimos o proprietário deste ToDo
        // com o id do usuário
        newToDo.owner = result[0]._id;
    }
    newToDo.save(function (err, result) {
        if (err !== null) {
            res.json(500, err);
        } else {
            res.json(200, result);
        }
    });
}
});
};

```

Nesse caso, você verá que estamos definindo `owner` com `null` caso o usuário não exista, e definimos `owner` com o `_id` associado ao usuário, caso contrário. Agora, quando executarmos o servidor, devemos ser capazes de acessar uma rota para cada usuário de teste criado e criar `ToDo`s associados a esse usuário em suas rotas. E, se acessarmos a página principal `localhost:3000`, devemos ver os `ToDo`s de todos os usuários.

Resumo

Criar uma aplicação web não é difícil depois que você souber como todas as partes funcionam. Por outro lado, criar uma aplicação web que possa ser *mantida* exige um pouco de raciocínio prévio e planejamento. A facilidade de manutenção de código em aplicações web está fortemente relacionada à ideia de separação das responsabilidades de determinados aspectos da aplicação. Em outras palavras, um programa deve ser constituído de várias partes pequenas que executem uma só tarefa e a faça muito bem. A interação entre essas partes deve ser minimizada o máximo possível.

Neste capítulo, vimos algumas dicas básicas sobre como manter o código do lado do cliente de forma clara e simples. Um dos aspectos mais importantes sobre esse assunto é que aprendemos a efetuar a generalização de conceitos significativos. Isso está intimamente relacionado à ideia de *programação orientada a objetos*.

Do lado do servidor, há um *padrão de projeto* que determina a estrutura do código. O padrão de projeto *Model-View-Controller* (Modelo-Visão-Controlador) corresponde a uma prática aceita para a organização de aplicações cliente/servidor de modo a facilitar a manutenção. A prática é tão amplamente aceita que frameworks inteiros foram desenvolvidos visando a forçar os desenvolvedores a criar aplicações que utilizem esse padrão. Entre esses frameworks está o Ruby on Rails.

No padrão MVC, o *modelo* faz a abstração do banco de dados por meio da utilização de algum tipo de ferramenta de modelagem de dados. Em nossos exemplos, a *visão* corresponde simplesmente à parte de nossa aplicação relacionada ao cliente (o HTML, o CSS e o JavaScript do lado do cliente). O *controlador* mapeia solicitações da visão às ações do modelo e responde com dados que podem ser utilizados para atualizar a visão.

Outra prática comum consiste em organizar a parte de uma aplicação web referente ao lado do servidor na forma de um web service *RESTful*. Um web service RESTful expõe os recursos do programa (por exemplo, os modelos de dados) ao lado cliente da aplicação por meio de URLs simples. O cliente faz uma solicitação usando um URL específico, alguns dados e um verbo HTTP, que o *roteador* da aplicação mapeará para a ação de um controlador.

Práticas e leituras adicionais

Apagando os ToDos

Uma questão que ainda não resolvemos é a capacidade de remover ToDos da aplicação. Preencha a ação `destroy` do controlador de `ToDo`

para que ele remova o `ToDo` contendo o ID especificado. A seguir, vamos acrescentar a capacidade de remover o `ToDo` da UI.

Como isso pode ser feito? Se tivermos as nossas rotas e as ações definidas corretamente, estamos retornando o objeto `ToDo` completo a partir da rota `todos.json`. Então estamos utilizando somente o campo `description` de nosso objeto para criar a UI. Também queremos usar o campo `_id` para criar um link de remoção.

Por exemplo, suponha que o código da aba que lista os nossos `ToDo`s tenha o seguinte aspecto:

```
"content":function (callback) {
  $.get("todos.json", function (todoObjects) {
    var $content = $("<ul>");
    // cria $content para a aba 'Oldest'
    todoObjects.forEach(function (todo) {
      $content.append($("<li>").text(todo.description));
    });
    // chama a callback com $content
    callback($content);
  });
}
```

Nesse caso, estamos adicionado um elemento `li` para cada objeto `ToDo`. Em vez de somente adicionar o texto ao `li`, queremos incluir um elemento `a` que faça a remoção do item. Por exemplo, queremos que o elemento resultante do DOM tenha um HTML que se pareça com:

```
<li>This is a ToDo item <a href="todos/5275643e0cff128714000001">remove</a></li>
```

Podemos fazer isso manipulando um pouco o nosso objeto `$content`:

```
var $todoListItem = $("<li>").text(todo.description),
    $todoRemoveLink = $("<a>").attr("href", "todos/"+todo._id);

// adiciona a âncora para remoção
$todoListItem.append($todoRemoveLink)
$content.append($todoListItem);
```

A seguir, podemos modificar o código de modo a associar um handler de clique ao link de remoção:


```

var $todoListItem = $("<li>").text(todo.description),
    $todoRemoveLink = $("<a>").attr("href", "todos/"+todo._id);
$todoRemoveLink.on("click", function () {
    $.ajax({
        // chama a solicitação HTTP delete no objeto
    }).done(function () {
        // depois de removermos o item de nossa aplicação com sucesso,
        // podemos remover esse item da lista do DOM
    });
    // retorna false para não seguirmos o link
    return false;
});
// adiciona a âncora para remoção
$todoListItem.append($todoRemoveLink)
$content.append($todoListItem);

```

É um pouco complicado fazer esse código funcionar corretamente, portanto planeje investir algum tempo nisso!

Adicionando uma página de administração de usuários

Até agora, não temos nenhuma maneira de adicionar ou de remover usuários de nossa aplicação. Criaremos uma página de administração de usuários que liste todos os usuários, juntamente com um campo de entrada que permita adicionar um usuário à aplicação. Isso exigirá o acréscimo de uma página adicional em nosso cliente, além de um arquivo JavaScript a mais para cuidar da UI dessa página. Por exemplo, você pode criar um arquivo *users.html* que importe um arquivo *users.js* em nosso diretório *javascripts*. É provável que você vá reutilizar a maior parte do CSS existente em sua aplicação, portanto não há problemas se você simplesmente fizer o link do arquivo *style.css* que está em seu diretório *stylesheets*. Nesse arquivo, é provável que você vá adicionar alguns elementos personalizados para efetuar uma estilização.

Depois que tudo isso estiver funcionando, pode ser interessante acrescentar um botão para cada usuário, que permita removê-lo da aplicação. Isso, é claro, fará a ação `destroy` ser disparada no modelo de usuários. Além de remover o usuário da coleção de usuários, você também deve remover todos os itens da lista de tarefas que estejam associados a esse usuário nessa ação.

É um pouco mais complicado do que o problema da remoção de usuários, portanto tente fazer esse último funcionar antes!

Visões que utilizam EJS e Jade

Um tópico importante que deixei de lado nesse tratamento do MVC e do Node.js refere-se aos engines de templating. Isso não aconteceu porque eu não ache que eles sejam úteis ou importantes, mas porque eu estava fazendo um esforço para manter o conteúdo um pouco mais administrável e oferecer uma oportunidade a você para trabalhar com chamadas AJAX do lado do cliente.

Apesar do que foi dito, sugiro que você invista algum tempo para ler tanto sobre o Jade (<http://jade-lang.com/>) quanto sobre o EJS (<http://embeddedjs.com/>). Eles adotam abordagens muito diferentes para gerar o HTML dinamicamente do lado do servidor, e ambos se integram muito bem com o Express.

Crie outra aplicação

Criamos uma aplicação simples que administra uma lista de tarefas a fazer (to-do list). Tente criar outra aplicação a partir do zero. Procure generalizar as ideias aprendidas e crie uma aplicação diferente. Se você consultar outros tutoriais, com frequência, verá exemplos que envolvem plataformas de blogging ou clonagens do Twitter. São projetos simples, fáceis de administrar, que ajudarão a consolidar os conhecimentos adquiridos neste livro.

Ruby On Rails

Um dos objetivos principais deste livro é prover conhecimentos suficientes aos leitores para que comecem a aprender Ruby on Rails, um dos frameworks web mais populares que existem. A essa altura, eu espero que você tenha um conhecimento básico suficiente para utilizar o Rails Tutorial (<http://ruby.railstutorial.org/>) de Michael Hartl. Nesse tutorial, ele faz uma descrição de como utilizar um ambiente orientado a testes para criar uma aplicação Rails simples. Muitos dos conceitos apresentados aqui devem se aplicar a esse tutorial.

Sobre o autor

Semmy Purewal dedicou aproximadamente uma década de sua vida dando aulas de ciência da computação e trabalhando como consultor autônomo de JavaScript. Durante esse período, ele trabalhou com um grupo diversificado de clientes compostos de startups, organizações sem fins lucrativos e laboratórios de pesquisa. Atualmente, sua atividade principal é como engenheiro de software em San José, Califórnia.

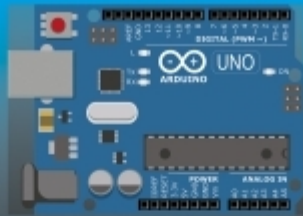
Colofão

Os animais na capa de *Desenvolvimento rápido de aplicações web* são as ovelhas alemãs Heidschnucke cinza (*Graue Gehörnte Heidschnucke*): são animais de pelos longos e cauda curta, nativos do hemisfério norte, da Escócia até a Sibéria e, especificamente, do norte da Alemanha, em uma área especial chamada Luneburg Heath. Algumas raças são classificadas como espécies domésticas ameaçadas de extinção e, desse modo, sua preservação é subsidiada na União Europeia.

O pelo dessa ovelhas é cinzento e muito longo, porém espesso demais para a maioria das aplicações têxteis; a lã serve apenas para tecelagens mais grosseiras, por exemplo, em carpetes. Ambos os sexos são dotados de chifres que se curvam sobre si mesmos e diz-se que a sua carne tem sabor de caça, o que os torna populares

para o consumo.

A quantidade dessas ovelhas diminuiu na virada do século XX e tem estado em declínio desde então. Nos anos 90, porém, muitos pequenos produtores rurais em toda a Alemanha começaram a criar essa raça de ovelhas e as salvaram da extinção. Com efeito, esses animais são tão celebrados que o “Moorland Sheep Day” ocorre todos os anos na segunda quinta-feira de julho em um pequeno vilarejo na Alemanha. Machos jovens são apresentados aos criadores e aos visitantes, e os melhores animais são premiados.



Conectando o Arduino à web

Desenvolvimento de frontend usando JavaScript

—
Indira Knight

novatec

apress®

Conectando o Arduino à web

Knight, Indira

9788575227138

288 páginas

[Compre agora e leia](#)

Crie interfaces físicas para interagir com a internet e com páginas web. Com Arduino e JavaScript você pode criar displays físicos interativos e ter dispositivos conectados que enviam ou recebem dados da web. Você tirará proveito dos processos necessários para configurar componentes eletrônicos, coletar dados e criar páginas web capazes de interagir com esses componentes. Por meio de exercícios, projetos e explicações, este livro permitirá que você tenha habilidades para fazer desenvolvimento web de frontend e lidar com componentes eletrônicos, a fim de criar interfaces físicas conectadas e implementar visualizações atraentes usando uma variedade de bibliotecas JavaScript. No final do livro você terá desenvolvido protótipos interativos totalmente funcionais, capazes de enviar e receber dados de uma interface física. Acima de tudo, este livro apresenta uma amostra do que é possível fazer e possibilita ter o conhecimento necessário para você criar suas próprias interfaces físicas conectadas e levar a web aos seus projetos eletrônicos. Aqui você aprenderá a: construir um painel de controle para a Internet das Coisas (Internet of Things), o qual se atualizará de acordo com componentes eletrônicos conectados a um

Arduino; usar componentes para interagir com displays 3D online; criar páginas web com HTML e CSS; configurar um servidor Node.js; usar Websockets para processar dados ao vivo; interagir com SVG (Scalable Vector Graphics). A quem este livro se destina Especialistas em tecnologia, desenvolvedores e entusiastas que queiram ampliar suas habilidades, ser capazes de desenvolver protótipos físicos com dispositivos conectados e que tenham interesse em começar a trabalhar com a IoT. Além disso, também se destina a pessoas empolgadas com a possibilidade de conectar o mundo físico à web.

[Compre agora e leia](#)

O'REILLY®

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações
nativas de nuvem



novatec

Bilgin Ibryam
Roland Huß

Padrões para Kubernetes

Ibryam, Bilgin

9788575228159

272 páginas

[Compre agora e leia](#)

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões:

- Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres.
- Padrões comportamentais, que exploram conceitos mais

específicos para administrar contêineres e interações com a plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

[Compre agora e leia](#)

CANDLESTICK

Um método para ampliar lucros na Bolsa de Valores



novatec

Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

[Compre agora e leia](#)

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos.

Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

[Compre agora e leia](#)



AVALIANDO
EMPRESAS

INVESTINDO EM AÇÕES

A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS

novatec

CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

[Compre agora e leia](#)

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

[Compre agora e leia](#)

Marcos Abe

MANUAL DE ANÁLISE TÉCNICA

ESSÊNCIA E ESTRATÉGIAS AVANÇADAS

TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE

novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

[Compre agora e leia](#)

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá:

- os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado;
- identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos;
- estruturar e proteger operações por meio do gerenciamento de capital.

Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.

[Compre agora e leia](#)