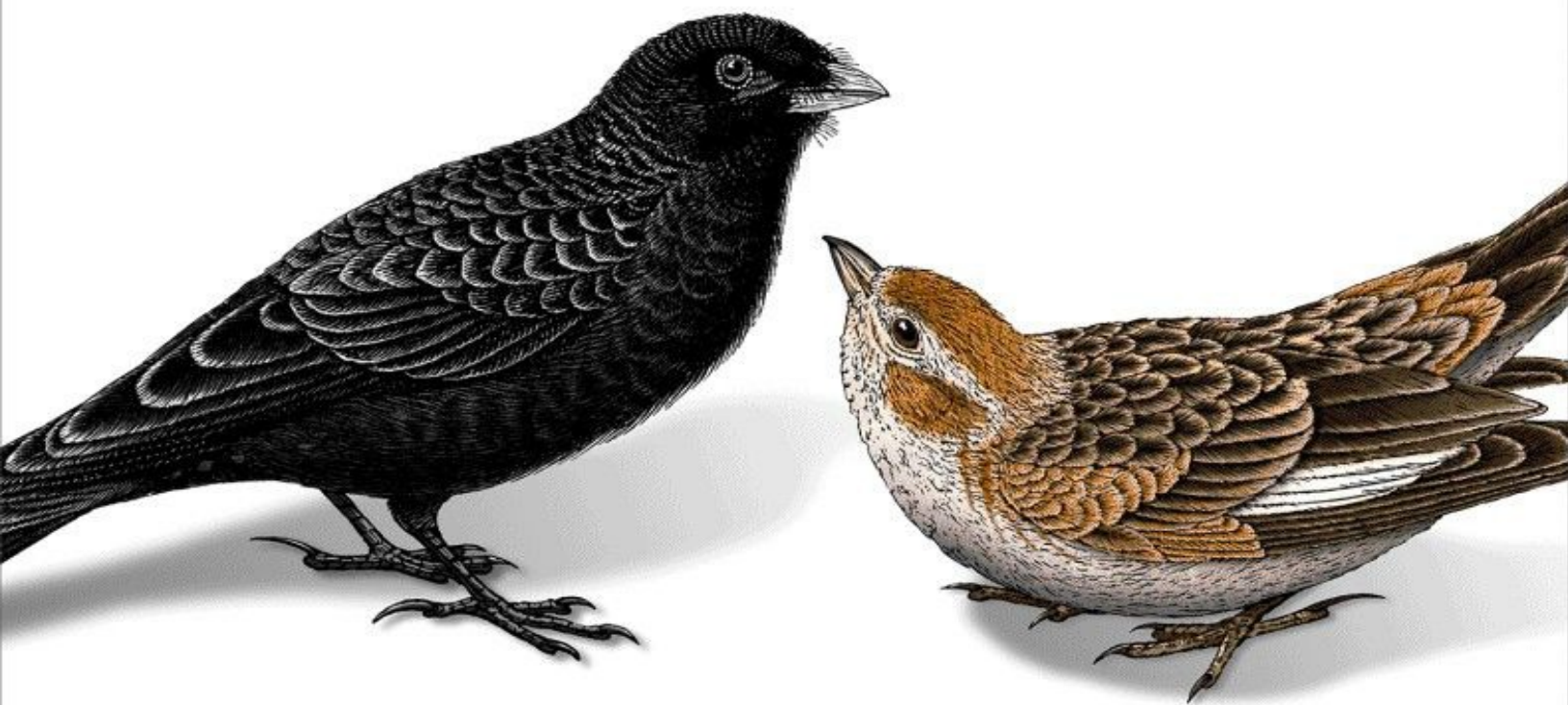


O'REILLY®

Programação web com Node e Express

Beneficiando-se da stack JavaScript



novatec

Ethan Brown

Authorized Portuguese translation of the English edition of Web Development with Node and Express 2E, ISBN 9781492053514 © 2020 Ethan Brown. 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 Web Development with Node and Express 2E, ISBN 9781492053514 © 2020 Ethan Brown. 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. [2020].

Copyright ©2020 da Novatec Editora Ltda.

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 Aldir Coelho Corrêa da Silva

Revisão gramatical: Tássia Carvalho

Editoração eletrônica: Carolina Kuwabata

ISBN do ebook: 978-65-86057-09-6

ISBN do impresso: 978-65-86057-08-9

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

Dedico este livro à minha família:

Meu pai, Tom, que me fez gostar de engenharia; minha mãe, Ann, que me fez gostar de escrever; e minha irmã, Meris, que tem sido uma companheira leal.

Prefácio

Para quem é este livro

Este livro é para programadores que queiram criar aplicações web (sites tradicionais; aplicações de página única (single-page applications) com o React, Angular ou Vue; APIs REST; ou algo intermediário) usando JavaScript, Node e Express. Um dos aspectos interessantes do desenvolvimento com o Node é que ele tem atraído um público de programadores totalmente novo. A acessibilidade e a flexibilidade do JavaScript têm conquistado programadores autodidatas do mundo todo. Em nenhum momento na história da ciência da computação foi tão acessível programar. O número e a qualidade dos recursos online que ensinam programação (e ajudam a obter ajuda quando não sabemos o que fazer) são surpreendentes e inspiradores. Logo, saúdo os novos programadores (e possivelmente autodidatas).

É claro que também existem programadores como eu, que já estão por aí há algum tempo. Como muitos programadores da minha época, comecei com assembler e BASIC e passei por Pascal, C++, Perl, Java, PHP, Ruby, C, C# e JavaScript. Na universidade, conheci linguagens mais relacionadas a nichos como ML, LISP e PROLOG. Várias delas são linguagens pelas quais tenho muito apreço, mas não considero nenhuma tão promissora quanto JavaScript. Portanto, também estou escrevendo este livro para programadores como eu, que têm muita experiência e talvez uma visão mais filosófica de tecnologias específicas.

Nenhuma experiência com o Node é necessária, mas é preciso ter alguma experiência com JavaScript. Se você é novo em programação, recomendo o site *Codecademy* (<http://bit.ly/2KfDqkQ>). Se é um programador intermediário ou experiente, recomendo meu livro *Learning JavaScript, 3rd Edition* (O'Reilly). Os exemplos deste livro podem ser usados com qualquer sistema no qual o Node funcione (o que abrange o Windows, macOS e Linux,

entre outros). Eles se destinam a usuários de linha de comando (terminal), logo, é preciso ter alguma familiaridade com o terminal de seu sistema.

O mais importante neste livro é que ele é para programadores empolgados. Empolgados com o futuro da internet e querendo fazer parte dele. Empolgados para aprender coisas novas, técnicas recentes e novas maneiras de considerar o desenvolvimento web. Se você, caro leitor, não é alguém empolgado, espero que tenha se tornado ao chegar ao fim do livro....

Notas sobre a segunda edição

Foi ótimo escrever a primeira edição deste livro e me sinto até hoje feliz com o aconselhamento prático que consegui introduzir nela e a calorosa resposta de meus leitores. A primeira edição foi publicada no mesmo momento em que o Express 4.0 foi lançado a partir da versão beta, e, embora ele ainda esteja na versão 4.x, o middleware e as ferramentas que o acompanham passaram por mudanças *massivas*. Além disso, o JavaScript também evoluiu, e até mesmo a maneira como as aplicações web são projetadas passou por uma mudança imensa (distanciando-se da pura renderização no lado do servidor e indo mais na direção das aplicações de página única [SPAs – single page applications]). Embora muitos dos princípios da primeira edição ainda sejam úteis e válidos, as técnicas e ferramentas específicas são quase em sua totalidade diferentes. Já estava passando da hora de haver uma nova edição. Devido à preponderância dos SPAs, o foco desta nova edição também mudou para dar mais ênfase ao Express como servidor de APIs e arquivos estáticos, e inclui um exemplo de SPA.

Como este livro foi organizado

O Capítulo 1 e o Capítulo 2 o apresentarão ao Node, ao Express e a algumas ferramentas que você usará no decorrer da leitura. No Capítulo 3 e no Capítulo 4, você começará a usar o Express e a construir o esqueleto de um site que será usado como exemplo pelo resto do livro.

O Capítulo 5 discute a execução de testes e a garantia da qualidade (QA), e o Capítulo 6 aborda algumas das estruturas mais importantes do Node e como

elas são estendidas e usadas pelo Express. O Capítulo 7 abrange o templating (com o uso do Handlebars), que forma a base da construção de sites úteis com o Express. O Capítulo 8 e o Capítulo 9 abordam os cookies, as sessões, e os manipuladores de formulários, mostrando tudo que você precisa saber para construir sites funcionais básicos com o Express.

O Capítulo 10 detalha o middleware, um conceito essencial no Express. O Capítulo 11 explica como usar o middleware para enviar emails a partir do servidor e discute questões de segurança e layout inerentes aos emails.

O Capítulo 12 antecipa algumas preocupações de produção. Ainda que nesse estágio do livro não tenhamos todas as informações necessárias para construir um site pronto para a produção, considerar esse ambiente nesse momento pode evitar problemas maiores no futuro.

O Capítulo 13 é sobre persistência, com ênfase no MongoDB (um dos principais bancos de dados de documentos) e no PostgreSQL (um sistema open-source popular de gerenciamento de banco de dados relacional).

O Capítulo 14 mostra os detalhes do roteamento com o Express (como as URLs são mapeadas para o conteúdo), e o Capítulo 15 entra no assunto da criação de APIs. O Capítulo 17 aborda os detalhes de como é servido conteúdo estático, com ênfase na maximização do desempenho.

O Capítulo 18 discute a segurança: como construir autenticação e autorização em seu aplicativo (com ênfase no uso de um provedor de autenticação de terceiros) e como executar seu site usando HTTPS.

O Capítulo 19 explica como fazer a integração com serviços de terceiros. Os exemplos usados são o Twitter, o Google Maps e o US National Weather Service¹.

O Capítulo 16 pega o que aprendemos sobre o Express e usa essas informações para refatorar o exemplo que perpassa o livro como um SPA, com o Express como servidor back-end fornecendo a API que criamos no Capítulo 15.

O Capítulo 20 e o Capítulo 21 o prepararão para o grande dia: o lançamento de seu site. Eles abordarão a depuração, para que você remova qualquer defeito existente antes do lançamento, e o processo de colocá-lo online. O

Capítulo 22 fala sobre a próxima fase importante (e com frequência negligenciada): a manutenção.

O livro termina com o Capítulo 23, que mostra recursos adicionais, caso você queira aumentar seu conhecimento sobre o Node e o Express, e aonde poderá ir para obter ajuda.

Exemplo de site

A partir do Capítulo 3, um exemplo recorrente será usado no livro: o site da Meadowlark Travel. Redigi a primeira edição assim que voltei de uma viagem a Lisboa, e estava com o assunto viagens em mente, logo, o exemplo de site que escolhi é o de uma agência de viagens fictícia no Oregon, o estado em que nasci (a cotovia ocidental [Western Meadowlark] é o pássaro representativo do Oregon). A Meadowlark Travel permite aos viajantes entrar em contato com “guias turísticos amadores” locais e é associada a empresas que oferecem aluguel de bicicletas e scooters e passeios locais, com ênfase no ecoturismo.

Como qualquer exemplo pedagógico, o site da Meadowlark Travel é fictício, mas mostra muitos dos desafios que os sites do mundo real enfrentam: integração com componentes de terceiros, geolocalização, ecommerce, desempenho e segurança.

Como o ponto de vista deste livro é o da infraestrutura de back-end, o exemplo de site não estará completo; ele serve simplesmente como um exemplo fictício de um site do mundo real para fornecer profundidade e contexto às situações. Possivelmente você estará trabalhando em seu próprio site e poderá usar o exemplo da Meadowlark Travel como template.

Convenções usadas neste livro

As convenções tipográficas a seguir são usadas no livro:

Itálico

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

Largura constante

Usada para listagens de programas, assim como dentro de parágrafos para indicar elementos de programa como nomes de variáveis ou funções, bancos de dados, tipos de dados, variáveis de ambiente, instruções e palavras-chave.

Largura constante em negrito

Mostra comandos ou demais textos que tenham de ser digitados literalmente pelo usuário.

Largura constante em itálico

Mostra um texto que tenha de ser substituído por valores fornecidos pelo usuário ou determinados pelo contexto.



Este elemento significa uma dica ou 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 <https://github.com/EthanRBrown/web-development-with-node-and-express-2e>.

Este livro está aqui para ajudá-lo a fazer seu trabalho. Se o código de exemplo for útil, você poderá usá-lo em seus programas e em sua documentação. Não é necessário 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. No entanto, vender ou distribuir um CD-ROM de exemplos de livros da O'Reilly requer. 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.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: “*Web Development with Node and Express, Second Edition* by Ethan Brown (O’Reilly). Copyright 2019 Ethan Brown, 978-1-492-05351-4”.

Se 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.

Como entrar em contato

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: novatec@novatec.com.br.

Temos uma página web para este livro na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português

<https://novatec.com.br/livros/programacao-web-com-node-express>

- Página da edição original em inglês

https://oreil.ly/web_dev_node_express_2e

- Página com material suplementar (exemplos de códigos, exercícios etc.).

<https://github.com/cloudnativedevops/demo>

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

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

Agradecimentos

Muitas pessoas que fazem parte da minha vida ajudaram a tornar este livro realidade; essa empreitada não teria sido possível sem a influência de todos que cruzaram minha trajetória e me transformaram em quem sou hoje.

Gostaria de começar agradecendo a todos os funcionários da Pop Art: além de o tempo que passei na Pop Art ter reacendido minha paixão por engenharia, aprendi muita coisa com todas as pessoas de lá, e, sem seu

suporte, este livro não existiria. Sou grato a Steve Rosenbaum por criar um local de trabalho inspirador, e a Del Olds por me receber, me fazer sentir bem-vindo e ser um líder honrado. Obrigado a Paul Inman por seu firme apoio e pela atitude estimulante no que diz respeito à engenharia, e a Tony Alferez por seu suporte caloroso e por me ajudar a encontrar tempo para escrever sem afetar a Pop Art. Para concluir, obrigado a todos os grandes engenheiros com quem trabalhei e que me ajudaram em minha preparação: John Skelton, Dylan Hallstrom, Greg Yung, Quinn Michaels, CJ Stritzel, Colwyn Fritze-Moor, Diana Holland, Sam Wilskey, Cory Buckley e Damion Moyer.

Tenho uma dívida de gratidão muito grande com minha equipe atual na Value Management Strategies, Inc. Aprendi muito sobre o lado empresarial dos programas com Robert Stewart e Greg Brink, e sobre comunicação, coesão e eficácia em equipe com Ashley Carson (obrigado por seu apoio resolutivo, Scratch Chromatic). Terry Hays, Cheryl Kramer e Eric Trimble, obrigado por seu trabalho árduo e suporte! Obrigado também a Damon Yeutter, Tyler Brenton e Brad Wells por seu trabalho crítico sobre análise de requisitos e gerenciamento de projeto. E o mais importante, obrigado aos talentosos e dedicados desenvolvedores que trabalharam – incansavelmente – comigo na VMS: Adam Smith, Shane Ryan, Jeremy Loss, Dan Mace, Michael Meow, Julianne Soifer, Matt Nakatani e Jake Feldmann.

Obrigado a todos os meus colegas de banda na Escola do Rock! Que jornada louca tem sido e que válvula de escape jovial e criativa temos. Agradeço especialmente aos instrutores que compartilham sua paixão e conhecimento musicais: Josh Thomas, Amanda Sloane, Dave Coniglio, Dan Lee, Derek Blackstone e Cory West. Obrigado a todos por me darem a oportunidade de ser uma estrela do rock!

Zach Mason, obrigado por ser uma inspiração para mim. Este livro não é *Os Cantos Perdidos da Odisseia*, mas é *meu*, e não sei se eu seria tão corajoso sem seu exemplo.

Elizabeth e Ezra, obrigado pelos presentes que me deram. Amarei os dois para sempre.

Devo tudo à minha família. Não poderia ter uma educação melhor e mais

amorosa do que a que eles me deram, e também vejo seus excepcionais cuidados refletidos em minha irmã.

Muito obrigado a Simon St. Laurent por me dar essa oportunidade e a Angela Rufino (segunda edição) e Brian Anderson (primeira edição) por sua edição segura e encorajadora. Obrigado a todos na O'Reilly por sua dedicação e paixão. Obrigado a Alejandra Olvera-Novack, Chetan Karande, Brian Sletten, Tamas Piros, Jennifer Pierce, Mike Wilson, Ray Villalobos e Eric Elliot por suas revisões técnicas perfeitas e construtivas.

Katy Roberts e Hanna Nelson deram feedbacks e conselhos inestimáveis sobre minha proposta “não solicitada” que tornou este livro possível. Muito obrigado às duas! Obrigado a Chris Cowell-Shah por seu excelente feedback sobre o capítulo de garantia da qualidade.

Por fim, obrigado aos meus queridos amigos, sem os quais certamente eu teria enlouquecido: Byron Clayton, Mark Booth, Katy Roberts e Kimberly Christensen. Amo todos vocês.

¹ N.T.: Serviço de previsão do tempo do governo americano.

Introduzindo o Express

A revolução do JavaScript

Antes de introduzir o principal assunto deste livro, é importante fornecer algum contexto histórico, e isso significa falar sobre JavaScript e Node. A época do JavaScript é agora. De seu modesto começo como linguagem de script para o cliente, ele não só se tornou totalmente onipresente no lado do cliente (client-side), como sua aceitação como linguagem para o lado do servidor (server-side) finalmente também ocorreu, graças ao Node.

A promessa de um stack de tecnologias todo em JavaScript é clara: deixar de lado a mudança de contexto! Você não terá mais de mudar o modo de pensar de JavaScript para PHP, C#, Ruby ou Python (ou qualquer outra linguagem do lado do servidor). Além disso, ela permite que os engenheiros front-end deem o salto para a programação no lado do servidor. Não quero dizer que a programação no lado do servidor tenha ligação apenas com a linguagem; há muito mais a se aprender. Com JavaScript, no entanto, pelo menos a linguagem não será uma barreira.

Este livro é para todos que veem como o stack de tecnologias JavaScript é promissor. Talvez você seja um engenheiro front-end querendo estender sua experiência ao desenvolvimento back-end. Ou pode ser um desenvolvedor back-end experiente como eu que esteja considerando o JavaScript como uma alternativa viável às linguagens específicas do lado do servidor.

Se você tem atuado como engenheiro de software há tanto tempo quanto eu, já viu muitas linguagens, frameworks e APIs popularizarem-se. Alguns foram adotados e outros caíram no esquecimento. Provavelmente você se orgulha de sua habilidade em aprender rapidamente novas linguagens e sistemas. Cada vez que depara com uma nova linguagem, ela parece um pouco mais familiar:

você reconhece algo de uma linguagem que aprendeu na universidade ou alguma outra coisa vista no emprego que tinha anos atrás. Certamente é confortável ter esse tipo de visão, mas também é exaustivo. Às vezes queremos apenas *fazer algo até o fim*, sem ter de aprender uma tecnologia totalmente nova ou relembrar habilidades que não usamos há meses ou anos.

Inicialmente o JavaScript pode parecer um vencedor improvável. Acredite, sou da mesma opinião. Se você me dissesse em 2007 que eu não só escolheria JavaScript como minha linguagem preferida, mas também escreveria um livro sobre ela, teria lhe falado que estava louco. Tenho todos os preconceitos usuais contra JavaScript: achava que era uma linguagem “não séria” (toy language), algo para amadores e diletantes usarem e abusarem. Para ser sincero, o JavaScript diminuiu os obstáculos para os amadores, e há muitos códigos JavaScript questionáveis por aí, o que não contribuiu para a reputação da linguagem. Invertendo um ditado popular, eu diria: “Odeie o jogador e não o jogo”.

É inadequado as pessoas terem esse preconceito contra JavaScript; ele impediu que elas percebessem como a linguagem é poderosa, flexível e elegante. Só agora muitas delas estão levando o JavaScript a sério, ainda que a linguagem como a conhecemos exista desde 1996 (mas muitos de seus recursos interessantes foram adicionados em 2005).

Se você comprou este livro, provavelmente não tem preconceito: porque, como eu, o superou ou porque nunca o teve. De qualquer forma, você tem sorte, e não vejo a hora de apresentá-lo ao Express, uma tecnologia tornada possível graças a uma linguagem encantadora e surpreendente.

Em 2009, anos após as pessoas começarem a perceber o poder e a expressividade do JavaScript como linguagem de script de navegador, Ryan Dahl viu seu potencial como linguagem do lado do servidor, e assim nasceu o Node.js. Essa foi uma época fértil para a tecnologia da internet. O Ruby (e o Ruby on Rails) pegou algumas boas ideias da ciência da computação acadêmica, combinou-as com novas ideias que ele próprio lançou, e mostrou ao mundo uma maneira mais rápida de construir sites e aplicações web. A Microsoft, em um corajoso esforço para tornar-se relevante na era da internet, fez coisas maravilhosas com a plataforma .NET e aprendeu com os erros não

só do Ruby e do JavaScript, mas também de Java, pegando ao mesmo tempo ideias emprestadas das salas das universidades.

Atualmente, os desenvolvedores web têm liberdade para usar os recursos mais recentes da linguagem JavaScript sem medo de deixar de lado usuários que tenham navegadores mais antigos, graças a tecnologias de transcompilação como o Babel. O webpack tornou-se a solução onipresente para o gerenciamento de dependências e a garantia do desempenho, e frameworks como o React, Angular e Vue estão mudando a maneira de as pessoas trabalharem no desenvolvimento web, relegando as bibliotecas declarativas de manipulação do Document Object Model (DOM) ao passado.

É uma época empolgante para nos envolvermos com a tecnologia da internet. Existem novas ideias interessantes (ou antigas ideias interessantes revitalizadas) em todos os locais. Há muitos anos não víamos um espírito de inovação e motivação tão grande.

Introduzindo o Express

O site do Express o descreve como um “framework minimalista e flexível para aplicações web em Node.js que fornece um conjunto de recursos robusto para aplicações web e móveis”. No entanto, o que isso significa? Dividiremos essa descrição:

Mínimo

Esse é um dos aspectos mais interessantes do Express. Quase sempre, os desenvolvedores de frameworks se esquecem de que “menos é mais”. A filosofia do Express é a de fornecer uma camada *mínima* entre nosso cérebro e o servidor. Isso não significa não ser robusto ou ter um número insuficiente de recursos úteis. Significa que ele interfere menos, permitindo-nos expressar nossas ideias plenamente, mas fornecendo ao mesmo tempo algo útil. O Express oferece um framework mínimo, e você pode adicionar diferentes partes de sua funcionalidade conforme desejado, substituindo o que não atender às suas necessidades. É um grande alívio. Muitos frameworks nos dão *tudo*, deixando-nos com um projeto inchado, misterioso e complexo antes de escrevermos uma única linha de código.

Com frequência, a primeira tarefa é perder tempo removendo funcionalidades desnecessárias ou substituindo a funcionalidade que não atende aos requisitos. O Express toma o rumo oposto, permitindo a inclusão do que é necessário quando é necessário.

Flexível

No fim das contas, o que o Express faz é muito simples: aceita requisições HTTP de um cliente (que pode ser um navegador, um dispositivo móvel, outro servidor, uma aplicação desktop... qualquer coisa que fale HTTP) e retorna uma resposta HTTP. Esse padrão básico descreve quase tudo que se conecta à internet, tornando o Express extremamente flexível em suas aplicações.

Framework de aplicação web

Uma descrição mais precisa seria “framework da parte do servidor (server-side) de uma aplicação web”. Hoje, quando pensamos em “framework de aplicação web”, geralmente nos lembramos de um framework de aplicação de página única (SPA) como o React, Angular ou Vue. No entanto, exceto por algumas aplicações autônomas, a maioria das aplicações web precisa compartilhar dados e se integrar com outros serviços. Elas costumam fazê-lo usando uma web API, que pode ser considerada o componente do lado do servidor de um framework de aplicação web. Porém, também é possível (e às vezes desejável) construir uma aplicação inteira com renderização somente no lado do servidor, caso em que o Express pode ser considerado o framework completo da aplicação web!

Além das características do Express mencionadas explicitamente em sua descrição, eu adicionaria mais duas:

Rápido

Quando o Express se tornou o framework de aplicações web mais procurado para o desenvolvimento com Node.js, ele chamou muito a atenção de empresas grandes que estavam executando sites de alto desempenho e tráfego intenso. Isso gerou uma grande pressão sobre a equipe para se concentrar no desempenho, e agora o Express oferece alto

desempenho para sites de tráfego pesado.

Não tendencioso

Uma das marcas registradas do ecossistema JavaScript é seu tamanho e diversidade. Embora com frequência o Express seja a parte principal do desenvolvimento web com o Node.js, há centenas (se não milhares) de pacotes de comunidades que podem participar de uma aplicação Express. A equipe do Express reconheceu a diversidade do ecossistema e respondeu fornecendo um sistema de middleware extremamente flexível que facilita o uso de componentes de nossa preferência na criação da aplicação. Durante o desenvolvimento do Express, podemos notar que ele substituiu componentes “internos” em favor de um middleware configurável.

Mencionei que o Express é a “parte do lado do servidor” de um framework de aplicações web...então, devemos considerar o relacionamento entre aplicações do lado servidor (server-side) e do lado do cliente (client-side).

Aplicações do lado do servidor e do lado do cliente

Uma *aplicação do lado do servidor* é aquela em que as páginas são renderizadas no servidor (como HTML, CSS, imagens e outros arquivos multimídia, e JavaScript) e enviadas para o cliente. Ao contrário, uma *aplicação do lado do cliente* renderiza grande parte de sua interface de usuário a partir de um pacote de aplicação inicial que é enviado apenas uma vez. Isto é, quando o navegador recebe o HTML inicial (em geral muito pequeno), ele usa JavaScript para modificar o DOM dinamicamente e não precisa depender do servidor para exibir novas páginas (embora dados brutos ainda venham do servidor).

Antes de 1999, as aplicações do lado do servidor eram o padrão. Na verdade, o termo *aplicação web* foi introduzido oficialmente naquele ano. Considero o período que fica aproximadamente entre 1999 e 2012 como a era Web 2.0, durante a qual as tecnologias e técnicas que acabariam se tornando aplicações do lado do cliente estavam sendo desenvolvidas. Por volta de 2012, com a sólida adoção dos smartphones, era prática comum enviar o menor número de informações possível pela rede, uma prática que favoreceu as aplicações do

lado do cliente.

As aplicações do lado do servidor costumam ser chamadas de *renderizadas no lado do servidor* (SSR, server-side rendered) e as aplicações do lado do cliente são chamadas de *aplicações de página única* (SPAs, single-page applications). As aplicações do lado do cliente são totalmente concebidas em frameworks como o React, Angular e Vue. Sempre considere “de página única” um termo incorreto porque – do ponto de vista do usuário – na verdade pode haver muitas páginas. A única diferença é se a página será enviada pelo servidor ou renderizada dinamicamente no cliente.

Há uma grande indefinição nas fronteiras existentes entre as aplicações do lado do servidor (server-side) e as do lado do cliente (client-side). Várias aplicações do lado do cliente têm de dois a três pacotes HTML que podem ser enviados para o cliente (por exemplo, a interface pública e a de quem fez login ou uma interface comum e uma administrativa). Além disso, as SPAs costumam ser combinadas com as SSRs para melhorar o desempenho de carregamento da primeira página e ajudar na otimização do mecanismo de busca (SEO, search engine optimization).

Em geral, se o servidor envia um pequeno número de arquivos HTML (provavelmente de um a três) e o usuário tem uma rica experiência multiview baseada na manipulação dinâmica do DOM, consideramos isso como renderização no lado do cliente. Os dados (quase sempre no formato JSON) e os arquivos multimídia de diferentes views costumam vir da rede.

É claro que para o Express não importa muito se você está criando uma aplicação do lado do servidor ou do lado cliente; ele atende às duas necessidades. Não faz diferença para o Express se você está servindo um único pacote HTML ou uma centena.

Embora as SPAs tenham definitivamente “vencido” como a arquitetura de aplicação web predominante, este livro começa com exemplos de aplicações do lado do servidor. Elas ainda são relevantes e a diferença conceitual entre servir um ou muitos pacotes HTML é pequena. Há um exemplo de SPA no Capítulo 16.

Uma breve história do Express

O criador do Express, TJ Holowaychuk, o descreve como um framework para aplicações web inspirado pelo Sinatra, um framework baseado no Ruby. Não surpreende que o Express seja derivado de um framework baseado em Ruby: o Ruby gerou ótimas abordagens para o desenvolvimento web, visando torná-lo mais rápido, mais eficiente e de manutenção mais fácil.

Assim como o Express foi baseado no Sinatra, ele também está profundamente ligado ao Connect, uma biblioteca de “plugins” para o Node. O Connect cunhou o termo *middleware* para descrever módulos plugáveis do Node que podem manipular requisições web em vários níveis. Em 2014, na versão 4.0, o Express removeu sua dependência do Connect, mas ainda deve a ele seu conceito de middleware.



O Express passou por uma reformulação bastante significativa entre as versões 2.x e 3.0, e depois novamente entre as versões 3.x e 4.0. Este livro enfoca a versão 4.0.

Node: Um novo tipo de servidor web

De certa forma, o Node tem muitas coisas em comum com outros servidores web populares, como o Internet Information Services (IIS) da Microsoft ou o Apache. Porém, o mais interessante está nas diferenças, então, começaremos por aí.

Como no Express, a abordagem do Node para os servidores web é mínima. Ao contrário do IIS ou Apache, que uma pessoa pode demorar muitos anos para dominar, o Node é fácil de instalar e configurar. Isso não significa que ajustar servidores Node para fornecerem um nível máximo de desempenho no ambiente de produção seja uma questão trivial; apenas as opções de configuração são mais simples e diretas.

Outra grande diferença entre o Node e os servidores web mais tradicionais é que ele é single threaded. À primeira vista, isso pode parecer um retrocesso. Na verdade, é um golpe de mestre. A arquitetura single threaded simplifica muito a tarefa de criar aplicativos web, e, se você precisar do desempenho de um aplicativo multithreaded, pode simplesmente criar mais instâncias do Node, e terá os benefícios do multithreading para o desempenho. Isso pode

soar ilusório para o leitor astuto. Afinal, o multithreading por intermédio do paralelismo de servidor (e não do paralelismo de aplicativo) apenas transfere a complexidade em vez de eliminá-la, certo? Talvez, mas, pela minha experiência, ele transferiu a complexidade para onde ela deveria estar. Além disso, com a crescente popularidade da computação em nuvem e o tratamento dos servidores como commodities genéricas, essa abordagem faz muito sentido. O IIS e o Apache são realmente poderosos, e foram projetados para extrair até a última gota de desempenho dos potentes equipamentos de hardware atuais. Contudo, isso tem um custo: é preciso ter um conhecimento de configuração e ajuste considerável para chegar a esse nível de desempenho.

No que diz respeito à maneira como os aplicativos são criados, os aplicativos Node são mais parecidos com os aplicativos PHP ou Ruby do que com os aplicativos .NET ou Java. Embora o engine JavaScript que o Node usa (o V8 do Google) compile JavaScript para código de máquina nativo (semelhante ao C ou C++), isso é feito tão transparentemente¹ que, do ponto de vista do usuário, ele se comporta como uma linguagem puramente interpretada. Não haver uma etapa de compilação separada reduz os problemas de manutenção e implantação: tudo que você tem de fazer é atualizar um arquivo JavaScript e suas alterações ficarão automaticamente disponíveis.

Outro benefício interessante dos aplicativos Node é que o Node é incrivelmente independente de plataforma. Ele não é a primeira ou a única tecnologia desse tipo, mas a independência de plataforma é na verdade mais um espectro do que uma proposição binária. Por exemplo, você pode executar aplicativos .NET em um servidor Linux graças ao Mono, mas é uma tarefa difícil devido à documentação irregular e às incompatibilidades de sistema. Da mesma forma, você pode executar aplicativos PHP em um servidor Windows, mas geralmente isso não é tão fácil de configurar como seria em uma máquina Linux. O Node, por outro lado, é fácil de configurar em todos os principais sistemas operacionais (Windows, macOS e Linux) e permite a colaboração sem complicações. Entre as equipes de design de sites, o uso de uma combinação de PCs e Macs é muito comum. Certas plataformas, como a .NET, introduzem desafios para desenvolvedores e projetistas front-end, que com frequência usam Macs, o que gera um grande

impacto sobre a colaboração e a eficiência. A ideia de poder criar um servidor funcional em qualquer sistema operacional em minutos (ou até mesmo segundos!) é um sonho tornado realidade.

Ecossistema do Node

É claro que o Node pode ser considerado o coração do stack de tecnologias JavaScript. Ele é o software que permite que o JavaScript seja executado no servidor, desacoplado de um navegador, o que por sua vez permite que frameworks escritos em JavaScript (como o Express) sejam usados. Outro componente importante é o banco de dados, que abordaremos com mais detalhes no Capítulo 13. Todos os aplicativos web, exceto os mais simples, precisam de um banco de dados, e alguns bancos de dados estão mais alinhados com o ecossistema do Node do que outros.

Não é novidade que há interfaces disponíveis para todos os maiores bancos de dados relacionais (MySQL, MariaDB, PostgreSQL, Oracle, SQL Server); seria tolo negligenciar esses monólitos estabelecidos. No entanto, o advento do desenvolvimento com o Node revitalizou uma nova abordagem para o armazenamento em bancos de dados: os chamados bancos de dados NoSQL. Nem sempre é útil definir alguma coisa pelo que ela *não* é, logo, gostaria de adicionar que esses bancos de dados NoSQL poderiam ser mais apropriadamente chamados de “bancos de dados de documentos” ou “bancos de dados de pares chave/valor”. Eles fornecem uma abordagem conceitualmente mais simples para o armazenamento de dados. Existem muitos, mas o MongoDB é um dos pioneiros, e é o banco de dados NoSQL que usaremos neste livro.

Já que a construção de um site funcional depende de várias tecnologias, acrônimos foram criados para descrever o “stack” no qual um site se baseia. Por exemplo, a combinação de Linux, Apache, MySQL e PHP é chamada de stack *LAMP*. Valeri Karpov, engenheiro do MongoDB, cunhou o acrônimo *MEAN*: Mongo, Express, Angular e Node. Embora certamente seja fácil de lembrar, é limitado: há tantas opções para bancos de dados e frameworks de aplicações que “MEAN” não captura a diversidade do ecossistema (e também deixa de fora o que considero um componente importante: os engines de

renderização).

Cunhar um acrônimo inclusivo é um exercício interessante. É claro que o componente indispensável é o Node. Embora haja outros contêineres JavaScript do lado do servidor, o Node está emergindo como o predominante. O Express também não é o único framework de aplicativos web disponível, mas chega próximo do Node em sua predominância. Os dois outros componentes que geralmente são essenciais para o desenvolvimento de aplicativos web são um servidor de banco de dados e um engine de renderização (seja um engine de templating como o Handlebars ou um framework de SPA como o React). No que diz respeito a esses dois últimos componentes, não há tantos pioneiros óbvios, e é nesses casos que acho um desserviço ser restritivo.

O que une essas tecnologias é o JavaScript, logo, em um esforço para ser inclusivo, usarei o termo *stack JavaScript*. Para os fins deste livro, isso inclui o Node, o Express e o MongoDB (também há um exemplo de banco de dados relacional no Capítulo 13).

Licenciamento

Ao desenvolver aplicativos Node, talvez você tenha de prestar mais atenção no licenciamento do que jamais precisou (eu certamente tive). Um dos encantos do ecossistema Node é o vasto conjunto de pacotes disponíveis. No entanto, cada um tem o próprio licenciamento, e o pior, cada pacote pode depender de outros pacotes, o que significa que pode ser complicado saber que licença é usada pelas diversas partes do aplicativo que você criou.

Porém, tenho boas novas. Uma das licenças mais populares dos pacotes Node é a MIT, que é descomplicadamente permissiva, permitindo-nos fazer *quase* tudo, inclusive usar o pacote em software proprietário. Contudo, não presumo que todos os pacotes que você está usando são controlados pela licença MIT.



Há vários pacotes disponíveis no npm que tentam descobrir as licenças de cada dependência do projeto. Procure no npm por `nlf` ou `license-report`.

Embora a MIT seja a licença mais fácil de encontrar, você também pode deparar com as seguintes licenças:

GNU General Public License (GPL)

A GPL é uma licença open source popular inteligentemente criada para manter o software livre. Ou seja, se você usar um código licenciado pela GPL em seu projeto, este *também* terá de ser licenciado pela GPL. É claro que isso significa que o projeto não poderá ser proprietário.

Apache 2.0

Como a MIT, essa licença permite usar uma licença diferente para o projeto, inclusive licença de software proprietário. No entanto, você deve incluir uma observação com os componentes que usam a licença Apache 2.0.

Berkeley Software Distribution (BSD)

Semelhante à licença Apache, essa licença permite usar a licença que quisermos para o projeto, contanto que seja incluída uma observação com os componentes licenciados pela BSD.



O software pode ter *licença dual* (ter dois tipos de licenças diferentes). Uma razão comum para isso ocorrer é permitirmos que o software seja usado tanto em projetos com licença GPL quanto em projetos com licenciamento mais permissivo. (Para ser usado em software GPL, um componente tem de ter a licença GPL). Esse é um esquema de licenciamento que emprego com frequência em meus projetos: licença dual com GPL e MIT.

Por fim, se você pretende criar os próprios pacotes, seja um bom cidadão selecionando uma licença para ele e documentando-o corretamente. Não há nada mais frustrante para um desenvolvedor do que usar o pacote de alguém e ter de examinar detalhadamente o código-fonte para determinar o licenciamento, ou pior, descobrir que ele não tem licença alguma.

Conclusão

Espero que este capítulo o tenha ajudado a entender melhor o que é o Express e como ele se encaixa no ecossistema maior do Node e JavaScript, e deixado um pouco mais claro o relacionamento entre as aplicações web do lado do

servidor e as do lado do cliente.

Se você continua confuso sobre o que é realmente o Express, não se preocupe: às vezes é muito mais fácil simplesmente começar a usar algo para depois o entender, e este livro o ajudará a começar a construir aplicações web com o Express. No entanto, antes de usar o Express, conheceremos o Node no próximo capítulo, uma informação importante para entendermos como o Express funciona.

¹ O que é com frequência chamado de compilação *just in time* (JIT).

CAPÍTULO 2

Introdução ao Node

Se você não tem nenhuma experiência com o Node, este capítulo o ajudará. Entender o Express e saber para que ele é útil requer um conhecimento básico do Node. Se você já tem experiência na construção de aplicativos web com o Node, pode pular este capítulo. Nele, construiremos um servidor web muito básico com o Node; no próximo capítulo, veremos como fazer o mesmo com o Express.

Obtendo o Node

Não poderia ser mais fácil instalar o Node em seu sistema. A equipe do Node se esforçou para assegurar que o processo de instalação seja simples e direto em todas as principais plataformas.

Acesse a *homepage do Node* (<http://nodejs.org>). Clique no grande botão verde que tem um número de versão seguido por “LTS (Recommended for Most Users)”. LTS é a abreviação de *Long-Term Support*, e essa versão é um pouco mais estável que a chamada de Current, que contém recursos mais recentes e melhorias no desempenho.

Para o Windows e macOS, é baixado um instalador que nos guia pelo processo. Para o Linux, provavelmente tudo ficará pronto com mais rapidez se você *usar um gerenciador de pacotes* (<http://bit.ly/36UYMxI>).



Se você é usuário do Linux e deseja utilizar um gerenciador de pacotes, certifique-se de seguir as instruções da página web mencionada. Várias distribuições Linux instalam uma versão muito antiga do Node quando não adicionamos o repositório de pacotes apropriado.

Também é possível baixar um instalador autônomo

(<https://nodejs.org/en/download>), que pode ser útil se você estiver distribuindo o Node para sua empresa.

Usando o terminal

Sou fã confesso do poder e da produtividade de usar um terminal (também chamado de *console* ou *prompt de comando*). No decorrer deste livro, todos os exemplos assumirão que você está usando um terminal. Se não gosta de seu terminal, recomendo que passe algum tempo se familiarizando com outro de sua preferência. Muitos dos utilitários do livro têm interfaces de GUI correspondentes, logo, se estiver determinado a não usar um terminal, há opções, mas você terá de encontrar seu próprio caminho.

Se você está no macOS ou Linux, há vários shells (o interpretador de comandos de terminal) respeitáveis para escolher. Sem dúvida o mais popular é o Bash, embora o zsh tenha seus adeptos. A principal razão para eu tender para o Bash (além da longa familiaridade) é sua onipresença. Sente-se na frente de qualquer computador baseado em Unix e verá que 99% das vezes o shell padrão é o Bash.

Se é usuário do Windows, as coisas não serão tão fáceis. A Microsoft nunca esteve particularmente interessada em fornecer uma experiência agradável com terminais, logo, você terá um pouco mais de trabalho. Felizmente, o Git inclui um shell “Git bash”, que fornece uma experiência de terminal Unix-like (ele tem apenas um pequeno subconjunto dos utilitários de linha de comando Unix que estão normalmente disponíveis, mas é um subconjunto útil). Embora o Git bash ofereça um shell Bash mínimo, ele continua usando a aplicação de console interna do Windows, o que leva à frustração (até mesmo funcionalidades simples como redimensionar uma janela do console, selecionar texto, cortar e colar são indiretas e complicadas). Portanto, recomendo instalar um terminal mais sofisticado como o *ConsoleZ* (<https://github.com/cbucher/console>) ou o *ConEmu* (<https://conemu.github.io>). Para usuários avançados do Windows – principalmente para desenvolvedores .NET ou administradores experientes de redes ou sistemas Windows – há outra opção: o PowerShell da própria Microsoft. O PowerShell está à altura de seu nome: as pessoas fazem coisas

extraordinárias com ele e um usuário hábil poderia fazer um guru da linha de comando Unix ter de se esforçar para ganhar a vida. No entanto, se você se alterna entre o macOS/Linux e o Windows, continuo recomendando o Git bash pela consistência que ele fornece.

Se você está usando o Windows 10 ou posterior, já pode instalar o Ubuntu Linux diretamente no Windows! Não se trata de dual-boot ou virtualização, e sim de um ótimo trabalho por parte da equipe open source da Microsoft para trazer a experiência Linux para o Windows. É possível instalar o Ubuntu no Windows por intermédio da *Microsoft App Store* (<http://bit.ly/2KcSfEI>).

Uma última opção para usuários do Windows é a virtualização. Com o poder e a arquitetura dos computadores modernos, o desempenho das máquinas virtuais (VMs, virtual machines) é praticamente indistinguível do das máquinas reais. Tive muita sorte com o *VirtualBox* gratuito da Oracle (<https://www.virtualbox.org/>).

Para concluir, independentemente do sistema usado, há ótimos ambientes de desenvolvimento baseado em nuvem, como o *Cloud9* (<https://aws.amazon.com/cloud9/>) (agora um produto da AWS). O Cloud9 cria um novo ambiente de desenvolvimento que facilita muito aprender a usar o Node rapidamente.

Quando você decidir que shell prefere, recomendo que passe algum tempo estudando os aspectos básicos. Há muitos tutoriais interessantes na internet (*The Bash Guide* (<https://guide.bash.academy>) é um ótimo ponto de partida), e você evitará dores de cabeça posteriormente aprendendo um pouco agora. No mínimo, precisa saber como navegar pelos diretórios; copiar, mover e excluir arquivos; e sair de um programa de linha de comando (geralmente com Ctrl-C). Se quiser se tornar um especialista em terminais, aprenda como procurar texto em arquivos, buscar arquivos e diretórios, encadear comandos (a velha “filosofia Unix”) e redirecionar a saída.



Em muitos sistemas Unix-like, a combinação de teclas Ctrl-S tem um significado especial: ela “congela” o terminal (era usada para pausar uma saída em sua rápida rolagem). Já que esse é um atalho muito comum para Salvar, é fácil pressionarmos sem pensar, o que leva a uma situação confusa para a maioria das pessoas (isso ocorre comigo com mais

frequência do que gosto de admitir). Para descongelar o terminal, basta pressionar Ctrl-Q. Logo, se não souber o que fazer com um terminal que travou repentinamente, tente pressionar Ctrl-Q e veja se ele volta ao normal.

Editores

Poucos tópicos motivam debates tão acalorados entre os programadores como a escolha de um editor, e por uma boa razão: o editor será sua principal ferramenta. Meu editor preferido é o vi (ou um editor que tenha um modo vi).¹ O vi não é para qualquer um (meus colaboradores sempre reviram os olhos em desaprovação quando lhes digo como seria fácil fazer o que eles estão fazendo usando o vi), mas encontrar um editor poderoso e aprender a usá-lo aumentará significativamente sua produtividade e, ousado dizer, sua satisfação. Uma das razões para eu gostar especialmente do vi (embora provavelmente não seja a mais importante) é que, como o Bash, ele é onipresente. Se você tem acesso a um sistema Unix, encontrará o vi nele. Os editores mais populares têm um “modo vi” que permite usar seus comandos de teclado. Quando nos acostumamos com ele, é difícil pensar em usar outra coisa. Inicialmente é difícil usar o vi, mas o que ele oferece vale o esforço.

Se, como eu, você acha que vale a pena se familiarizar com um editor que esteja disponível em qualquer local, sua outra opção é o Emacs. O Emacs e eu nunca nos demos muito bem (e geralmente ou você é uma pessoa do tipo que usa o Emacs ou do tipo que usa o vi), mas respeito o poder e a flexibilidade que ele fornece. Se a abordagem de edição modal não lhe agrada, recomendo que teste o Emacs.

Embora conhecer um editor de console (como o vi ou o Emacs) seja incrivelmente útil, você pode querer usar um editor mais moderno. Uma opção popular é o *Visual Studio Code* (<https://code.visualstudio.com/>) (não deve ser confundido com o Visual Studio sem o “Code”). Endosso com veemência o uso do Visual Studio Code; trata-se de um editor bem projetado, rápido e eficiente que é perfeitamente adequado para o desenvolvimento com Node e JavaScript. Outra opção popular é o *Atom* (<https://atom.io>), que também é estimado na comunidade JavaScript. Esses dois editores estão

disponíveis gratuitamente no Windows, macOS e Linux (e ambos têm modos vi!).

Agora que temos uma boa ferramenta para editar código, voltaremos nossa atenção para o npm, que nos ajudará a obter pacotes que outras pessoas criaram para podemos nos beneficiar da grande e ativa comunidade JavaScript.

npm

O npm é o onipresente gerenciador de pacotes Node (e é com ele que obteremos e instalaremos o Express). Como ocorre na estranha tradição do PHP, GNU, WINE e outros, *npm* não é um acrônimo (e é por isso que não está em maiúsculas); é uma abreviação recursiva para “npm não é um acrônimo”².

De modo geral, as duas principais responsabilidades de um gerenciador de pacotes são instalar pacotes e gerenciar dependências. O npm é um gerenciador de pacotes veloz, adequado e descomplicado, e acho que é em grande parte responsável pelo rápido crescimento e pela diversidade do ecossistema Node.



Há um popular gerenciador de pacotes concorrente chamado Yam que usa o mesmo banco de dados de pacotes do npm; utilizaremos o Yam no Capítulo 16.

O npm é instalado quando instalamos o Node, logo, se você seguiu as etapas listadas anteriormente, já deve tê-lo. Então, ao trabalho!

O comando que você mais usará com o npm é `install` (o que já era de se esperar). Por exemplo, para instalar o nodemon (um utilitário popular que reinicia automaticamente um programa Node quando fazemos alterações no código-fonte), você emitiria o comando a seguir (no console):

```
npm install -g nodemon
```

A flag `-g` solicita ao npm que instale o pacote *globalmente*, o que significa que ele estará disponível globalmente no sistema. Essa diferença ficará mais clara quando abordarmos os arquivos *package.json*. Por enquanto, a regra prática é a de que os utilitários JavaScript (como o nodemon) devem ser

instalados globalmente, ao contrário de pacotes que sejam específicos de seu projeto ou aplicativo web.



Ao contrário de linguagens como Python – que passou por uma alteração maior da versão 2.0 para a 3.0, precisando de uma maneira de facilitar a alternância entre diferentes ambientes – a plataforma Node é tão nova que provavelmente você estará executando sua versão mais recente. No entanto, se precisar dar suporte a várias versões do Node, use o *nvm* (<https://github.com/creationix/nvm>) ou o *n* (<https://github.com/tj/n>), que permitem a alternância entre os ambientes. Você pode descobrir que versão do Node está instalada em seu computador digitando `node --version`.

Um servidor web simples com o Node

Se você já construiu um site HTML estático ou vem da experiência com PHP ou ASP, provavelmente está acostumado com a ideia do servidor web (que poderia ser o Apache ou o IIS) disponibilizando seus arquivos estáticos de modo que um navegador possa visualizá-los pela rede. Por exemplo, se você criar o arquivo *about.html* e inseri-lo no diretório apropriado, poderá navegar para `http://localhost/about.html`. Dependendo da configuração de seu servidor web, você pode até mesmo omitir *.html*, mas o relacionamento entre a URL e o nome do arquivo é claro: o servidor web saberá onde o arquivo está no computador e o enviará para o navegador.



localhost, como o nome sugere, é o computador em que você está. É um alias comum para o endereço Ipv4 de loopback 127.0.0.1 ou o endereço IPv6 de loopback ::1. Com frequência, o que vemos é 127.0.0.1, mas usarei *localhost* neste livro. Se você estiver em um computador remoto (usando o SSH, por exemplo), lembre-se de que navegar para *localhost* não estabelecerá uma conexão com esse computador.

O Node oferece um paradigma diferente do de um servidor web tradicional: o aplicativo criado é o servidor web. O Node fornece apenas o framework para a construção do servidor.

Você poderia dizer “mas eu não quero criar um servidor web!”. É uma

resposta natural: você quer criar um aplicativo, e não um servidor web. No entanto, o Node simplifica a criação desse servidor (em apenas algumas linhas), e o controle que ganhamos sobre o aplicativo mais do que vale a pena.

Então, mãos à obra. Você instalou o Node, se familiarizou com o terminal e agora está pronto para prosseguir.

Hello World

Sempre achei inadequado o exemplo canônico de introdução à programação ser a insípida mensagem “Hello world”. Porém, a essa altura parece quase um sacrilégio desrespeitar tradição tão arraigada, logo, começaremos dessa forma e depois passaremos para algo mais interessante.

Em seu editor favorito, crie um arquivo chamado *helloworld.js* (*ch02/00-helloworld.js* no repositório fornecido):

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('Hello world!')
})

server.listen(port, () => console.log(`server started on port ${port}; `
  + 'press Ctrl-C to terminate....'))
```



Dependendo de quando e onde você aprendeu JavaScript, pode estranhar a falta de ponto e vírgula nesse exemplo. Eu costumava ser um ferrenho defensor do ponto e vírgula, e relutantemente parei de usá-lo, à medida que me envolvia com o desenvolvimento no React, onde é comum omiti-lo. Após algum tempo, me dei conta da situação e comecei a ponderar por que ficava tão empolgado com o ponto e vírgula! Agora sou um membro convicto da equipe do “não ao ponto e vírgula” e os exemplos deste livro refletem isso. É uma escolha pessoal e você pode usar o ponto e vírgula se quiser.

Verifique se está no mesmo diretório de *helloworld.js* e digite `node`

helloworld.js. Em seguida, abra um navegador, acesse *http://localhost:3000* e *voilà!* Aí está seu primeiro servidor web. Esse em particular não serve HTML; ele exibe somente a mensagem “Hello world!” em texto puro (plain text) no navegador. Se quiser, você pode tentar enviar HTML: apenas altere `text/plain` para `text/html` e `'Hello world!'` para uma string contendo HTML válido. Não demonstrei isso porque tento evitar escrever HTML dentro de JavaScript por razões que serão discutidas com mais detalhes no Capítulo 7.

Programação baseada em eventos

A filosofia básica existente por trás do Node é a da *programação baseada em eventos*. O que isso significa para você, o programador, é que é preciso saber quais eventos estão disponíveis e como responder a eles. Muitas pessoas são introduzidas à programação baseada em eventos pela implementação de uma interface de usuário: o usuário clica em algo e você manipula o *evento de clique*. É uma boa metáfora, porque fica subentendido que o programador não tem controle sobre quando, ou se, o usuário vai clicar em algo, logo, a programação baseada em eventos é mesmo muito intuitiva. Pode ser um pouco mais difícil dar o salto conceitual e responder a eventos no servidor, mas o princípio é o mesmo.

No exemplo de código anterior, o evento é implícito: o evento que está sendo manipulado é uma requisição HTTP. O método `http.createServer` recebe uma função como argumento; essa função será chamada sempre que uma requisição HTTP for feita. Nosso programa simples apenas configura o tipo de conteúdo como texto puro e envia a string “Hello world!”.

Quando você começar a pensar de acordo com a programação baseada em eventos, verá eventos em todos os lugares. Um desses eventos é quando um usuário navega de uma página ou área da aplicação para outra. Como sua aplicação responderá a esse evento de navegação é o que chamamos de *roteamento*.

Roteamento

Roteamento é o mecanismo que serve ao cliente o conteúdo que ele solicitou. Em aplicações cliente/servidor baseadas na web, o cliente define o conteúdo

desejado na URL; especificamente, o path e a querystring (as partes de uma URL serão discutidas com mais detalhes no Capítulo 6).



Normalmente, o roteamento realizado pelo servidor depende do path e da querystring, mas há outras informações disponíveis: cabeçalhos, o domínio, o endereço IP e outras. Isso permite aos servidores levar em consideração, por exemplo, a localização física aproximada do usuário ou seu idioma.

Expandiremos nosso exemplo “Hello world!” para que faça algo mais interessante. Serviremos um site realmente mínimo composto de uma homepage, uma página About e uma página Not Found. Por enquanto, usaremos o exemplo anterior e serviremos apenas texto puro em vez de HTML (*ch02/01-helloworld.js* no repositório fornecido):

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req,res) => {
  // normaliza a url removendo a querystring e a barra final
  // opcional e usando letras minúsculas
  const path = req.url.replace(/^(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('Homepage')
      break
    case '/about':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('About')
      break
    default:
      res.writeHead(404, { 'Content-Type': 'text/plain' })
      res.end('Not Found')
      break
  } })

server.listen(port, () => console.log(`server started on port ${port}; `
  + 'press Ctrl-C to terminate....'))
```

Se você executar esse código, verá que agora pode navegar para a homepage (<http://localhost:3000>) e a página About (<http://localhost:3000/about>). Qualquer querystring será ignorada (logo, <http://localhost:3000/?foo=bar> servirá a homepage) e qualquer outra URL (<http://localhost:3000/foo>) abrirá a página Not Found.

Servindo recursos estáticos

Agora que temos um roteamento simples funcionando, serviremos HTML real e a imagem de um logotipo. Esses recursos são chamados de *estáticos* porque geralmente não mudam (ao contrário de, por exemplo, um cotador de ações: sempre que a página é recarregada, os preços das ações podem ter mudado).



Servir recursos estáticos com o Node é adequado para desenvolvimento e projetos pequenos, mas para projetos maiores é melhor usar um servidor proxy como o NGINX ou uma CDN. Consulte o Capítulo 17 para ver mais informações.

Se você já trabalhou com o Apache ou o IIS, provavelmente está acostumado a apenas criar um arquivo HTML, navegar para ele e distribuí-lo para o navegador automaticamente. O Node não funciona assim: teremos de nos encarregar do trabalho de abrir o arquivo, lê-lo e então enviar seu conteúdo para o navegador. Portanto, criaremos um diretório em nosso projeto chamado *public* (por que não o chamamos de *static* ficará claro no próximo capítulo). Nesse diretório, criaremos *home.html*, *about.html*, *404.html*, um subdiretório chamado *img* e uma imagem chamada *img/logo.png*. Deixarei isso a seu cargo; se está lendo este livro, deve saber como criar um arquivo HTML e encontrar uma imagem. Em seus arquivos HTML, refira-se ao logotipo desta forma: ``.

Agora modifique *helloworld.js* (*ch02/02-helloworld.js* no repositório fornecido):

```
const http = require('http')
const fs = require('fs')
const port = process.env.PORT || 3000
```

```
function serveStaticFile(res, path, contentType, responseCode = 200) {
  fs.readFile(__dirname + path, (err, data) => {
    if(err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' })
      return res.end('500 - Internal Error')
    }
    res.writeHead(responseCode, { 'Content-Type': contentType })
    res.end(data)
  })
}
```

```
const server = http.createServer((req,res) => {
  // normaliza as url removendo a querystring e a barra final
  // opcional e usando letras minúsculas
  const path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      serveStaticFile(res, '/public/home.html', 'text/html')
      break
    case '/about':
      serveStaticFile(res, '/public/about.html', 'text/html')
      break
    case '/img/logo.png':
      serveStaticFile(res, '/public/img/logo.png', 'image/png')
      break
    default:
      serveStaticFile(res, '/public/404.html', 'text/html', 404)
      break
  }
})
```

```
server.listen(port, () => console.log(`server started on port ${port}; `
  + 'press Ctrl-C to terminate....'))
```



Nesse exemplo, estamos sendo pouco criativos em nosso roteamento. Se você navegar para `http://localhost:3000/about`, o arquivo `public/about.html` será servido. Poderíamos alterar a rota para qualquer outra e mudar o arquivo da mesma forma. Por exemplo, se você tivesse uma página About diferente para cada dia da semana, poderia ter os arquivos

`public/about_mon.html`, `public/about_tue.html` e assim por diante e fornecer uma lógica para seu roteamento servir a página apropriada quando o usuário navegar para `http://localhost:3000/about`.

Observe que criamos uma função auxiliar, `serveStaticFile`, que faz grande parte do trabalho. `fs.readFile` é um método assíncrono para a leitura de arquivos. Há uma versão síncrona dessa função, `fs.readFileSync`, porém, quanto mais rápido você começar a pensar assincronamente, melhor. A função `fs.readFile` usa um padrão chamado *callbacks*. Fornecemos uma função chamada *função de callback*, e, quando o trabalho termina, ela é chamada (digamos que ela é “chamada de volta”). Nesse caso, `fs.readFile` lê o conteúdo do arquivo especificado e executa a função de callback quando ele termina de ser lido; se o arquivo não existir ou houver problemas de permissão na sua leitura, a variável `err` será ativada e a função retornará o código de status HTTP 500 indicando erro do servidor. Se o arquivo for lido com sucesso, ele será enviado para o cliente com o código de resposta e o tipo de conteúdo especificado. Os códigos de resposta serão discutidos com mais detalhes no Capítulo 6.



`__dirname` será resolvida para o diretório no qual o script executado reside. Logo, se seu script residir em `/home/sites/app.js`, `__dirname` será resolvida para `/home/sites`. É uma boa ideia usar essa útil variável global sempre que possível. Não o fazer pode causar erros difíceis de diagnosticar se você executar seu aplicativo a partir de um diretório diferente.

A caminho do Express

Até aqui, talvez você não esteja achando o Node tão impressionante. Apenas replicamos o que o Apache ou o IIS fazem automaticamente, mas já sabemos como o Node age e o nível de controle que ele fornece. Não fizemos nada particularmente interessante, porém é claro que podemos usar esse material como ponto de partida para coisas mais sofisticadas. Se continuássemos seguindo esse caminho, criando aplicações Node cada vez mais sofisticadas, acabaríamos com algo parecido com o Express....

Felizmente, não precisamos fazê-lo: o Express já existe e nos ajudará a

implementar uma infraestrutura cuja criação seria muito demorada. Agora que adquirimos um pouco de experiência com o Node, estamos prontos para conhecer o Express.

- 1 Atualmente, vi é sinônimo de vim (vi improved). Na maioria dos sistemas, o vi é chamado de vim, mas geralmente digito vim para me certificar de estar mesmo usando o vim.
- 2 N.T.: Na verdade, npm significa sim Node Package Manager, mas posteriormente ficou decidido que seria melhor dar a aparência de que ele também poderia ou deveria ser usado para programas não relacionados ao Node, portanto, mudaram seu significado.

Economizando tempo com o Express

No Capítulo 2, você aprendeu a criar um servidor web simples usando somente o Node. Neste capítulo, criaremos o servidor usando o Express. Isso fornecerá um ponto de partida para o resto do conteúdo deste livro e o apresentará aos aspectos básicos do Express.

Scaffolding

A ideia de *Scaffolding* não é nova, mas muitas pessoas (inclusive eu) conheceram o conceito no Ruby. A ideia é simples: a maioria dos projetos requer alguma quantidade do chamado código *boilerplate*, e é claro que não queremos recriar esse código sempre que iniciarmos um novo projeto. Uma maneira simples é criar um esboço bruto e, sempre que você precisar de um novo projeto, apenas copiar esse esboço, ou template.

O Ruby on Rails aperfeiçoou esse conceito fornecendo um programa que gera automaticamente o scaffolding. A vantagem dessa abordagem é que ela pode gerar uma estrutura mais sofisticada do que a que obteríamos apenas fazendo a seleção em uma coleção de templates.

O Express seguiu o exemplo do Ruby on Rails e fornece um utilitário de geração de scaffolding que inicia o projeto.

Embora o utilitário de scaffolding seja útil, acho importante aprender como configurar o Express a partir do zero. Além de saber mais, você terá um controle maior sobre o que está sendo instalado e sobre a estrutura de seu projeto. O utilitário de scaffolding do Express também é destinado à geração de HTML no lado do servidor e é menos relevante para APIs e aplicações de página única.

Não usaremos o utilitário de scaffolding, mas recomendo que você o examine

ao terminar o livro: nesse momento você já estará de posse de todas as informações necessárias para avaliar se o scaffolding que ele gera é útil. Para saber mais, veja a *documentação do express-generator* (<http://bit.ly/2CyvvLr>).

Site da Meadowlark Travel

No decorrer deste livro, usaremos um exemplo recorrente: o site fictício da Meadowlark Travel, uma empresa que oferece serviços para as pessoas visitarem o grande Estado do Oregon. Se você estiver mais interessado em criar uma API, não se preocupe: o site da Meadowlark Travel exporá uma API além de servir um site funcional.

Etapas iniciais

Comece criando um novo diretório: ele será o diretório raiz de seu projeto. Neste livro, sempre que falarmos sobre o diretório do projeto, o diretório do aplicativo ou a raiz do projeto, estaremos nos referindo a esse diretório.



Provavelmente você vai querer manter os arquivos de seu aplicativo web separados de todos os outros arquivos que geralmente acompanham um projeto, como notas de reunião, documentação etc. Portanto, recomendo tornar a raiz do projeto um subdiretório do diretório do projeto. Por exemplo, para o site da Meadowlark Travel, eu poderia manter o projeto em `~/projects/meadowlark` e a raiz em `~/projects/meadowlark/site`.

O npm gerencia as dependências do projeto – assim como seus metadados – em um arquivo chamado *package.json*. A maneira mais fácil de criar esse arquivo é executando o comando `npm init`: ele fará várias perguntas e gerará um arquivo *package.json* (para a pergunta do “ponto de entrada”, use *meadowlark.js* como nome de seu projeto).



Quando você executar o npm, pode ver avisos sobre as informações ausentes de um campo de descrição ou repositório. É seguro ignorar esses avisos, mas, se quiser eliminá-los, edite o arquivo *package.json* e forneça valores para os campos sobre os quais o npm estiver reclamando.

Para obter mais informações sobre os campos desse arquivo, consulte a *documentação do arquivo package.json do npm* (<http://bit.ly/2O8HrbW>).

A primeira etapa será instalar o Express. Execute o comando npm a seguir:

```
npm install express
```

A execução de `npm install` instalará os pacotes nomeados no diretório `node_modules` e atualizará o arquivo `package.json`. Já que o diretório `node_modules` pode ser gerado novamente a qualquer momento com o npm, não vamos salvá-lo em nosso repositório. Para termos certeza de que não o adicionaremos acidentalmente ao repositório, criaremos um arquivo chamado *gitignore*:

```
# ignora pacotes instalados pelo npm
node_modules
```

```
# coloque aqui qualquer outro arquivo que não queira
# inserir no repositório, como .DS_Store
# (OSX), *.bak etc.
```

Agora criaremos um arquivo chamado *meadowlark.js*. Ele será o ponto de entrada de nosso projeto. No decorrer do livro, vamos chamá-lo apenas de *arquivo do aplicativo* (*ch03/00-meadowlark.js* no repositório fornecido):

```
const express = require('express')

const app = express()

const port = process.env.PORT || 3000

// página 404 personalizada
app.use((req, res) => {
  res.type('text/plain')
  res.status(404)
  res.send('404 - Not Found')
})

// página 500 personalizada
app.use((err, req, res, next) => {
  console.error(err.message)
  res.type('text/plain')
```



```
res.status(500)
res.send('500 - Server Error')
})
```

```
app.listen(port, () => console.log(
  `Express started on http://localhost:${port}; `
  + `press Ctrl-C to terminate.`))
```



Muitos tutoriais, assim como o gerador de scaffolding do Express, recomendam nomear o arquivo principal como *app.js* (ou às vezes *index.js* ou *server.js*). A menos que você esteja usando um serviço de hospedagem ou um sistema de implantação que exija que o arquivo principal da aplicação tenha um nome específico, não acho que haja uma boa razão para fazer isso, e prefiro nomear o arquivo principal com o nome do projeto. Qualquer pessoa que já tenha olhado para as diversas abas de um editor e percebido que todas são “index.html” verá imediatamente por que é inteligente agir assim. `npm init` usa como padrão *index.js*; se você usar um nome diferente para o arquivo de sua aplicação, certifique-se de atualizar a propriedade `main` em *package.json*.

Agora você tem um servidor Express mínimo. Inicie o servidor (`node meadowlark.js`) e navegue para `http://localhost:3000`. O resultado será frustrante: você não forneceu nenhuma rota para o Express, logo, ele exibirá uma mensagem 404 genérica indicando que a página não existe.



Observe como selecionamos a porta em que queremos que nossa aplicação seja executada: `const port = process.env.PORT || 3000`. Isso nos permite sobrepor a porta definindo uma variável de ambiente antes de iniciar o servidor. Se seu aplicativo não estiver sendo executado na porta 3000 quando você trabalhar com esse exemplo, verifique se a variável de ambiente `PORT` foi definida.

Adicionaremos algumas rotas para a homepage e uma página About. Antes do manipulador da mensagem 404, incluiremos duas rotas novas (*ch03/01-meadowlark.js* no repositório fornecido):

```
app.get('/', (req, res) => {
  res.type('text/plain')
  res.send('Meadowlark Travel');
```

```

}))

app.get('/about', (req, res) => {
  res.type('text/plain')
  res.send('About Meadowlark Travel')
})

// página 404 personalizada
app.use((req, res) => {
  res.type('text/plain')
  res.status(404)
  res.send('404 - Not Found')
})

```

`app.get` é o método com o qual estamos adicionando rotas. Na documentação do Express, você verá `app.METHOD`. Isso não significa que há literalmente um método chamado `METHOD`; é apenas um placeholder para verbos HTTP (em minúsculas; `get` e `post` são os mais comuns). Esse método recebe dois parâmetros: um `path` e uma função.

É o *path* que define a rota. Observe que `app.METHOD` se encarrega do trabalho pesado para nós: por padrão, ele não se importa com letras maiúsculas ou minúsculas ou com a barra final, e não considera a *querystring* quando executa a busca. Logo, a rota da página About funcionaria para `/about`, `/About`, `/about/`, `/about?foo=bar`, `/about/?foo=bar` etc.

A *função* fornecida será chamada quando a rota for encontrada. Os parâmetros passados para essa função são os objetos de requisição e resposta, sobre os quais aprenderemos mais no Capítulo 6. Por enquanto, estamos apenas retornando texto puro com o código de status 200 (o Express usa por padrão o código de status 200 – não é preciso especificá-lo explicitamente).



Recomendo a obtenção de um plugin de navegador que exiba o código de status da requisição HTTP assim como qualquer redirecionamento que ocorra. Ele facilitará a identificação de problemas de redirecionamento em seu código ou códigos de status incorretos, que com frequência são ignorados. Para o Chrome, o Redirect Path da Ayima funciona muito bem. Na maioria dos navegadores, você pode ver o código de status na seção Network das ferramentas de desenvolvedor.

Em vez de usar o método de baixo nível `res.end` do Node, mudamos e estamos usando a extensão do Express, `res.send`. Também estamos substituindo o método `res.writeHead` do Node por `res.set` e `res.status`. O Express também fornece um método conveniente, `res.type`, que define o cabeçalho `Content-Type`. Embora ainda seja possível usar `res.writeHead` e `res.end`, isso não é necessário ou recomendado.

Observe que nossas páginas 404 e 500 personalizadas devem ser manipuladas de maneira um pouco diferente. Em vez de usar `app.get`, estamos usando `app.use`, o método com o qual o Express adiciona *middleware*. Abordaremos o *middleware* com mais detalhes no Capítulo 10, mas por enquanto você pode considerá-lo como um manipulador de qualquer coisa que não coincida com uma rota. Isso levanta uma questão importante: *no Express, a ordem na qual as rotas e o middleware são adicionados é importante*. Se inserirmos o manipulador da mensagem 404 acima das rotas, a homepage e a página About pararão de funcionar; essas URLs resultarão em um erro 404. No momento, nossas rotas são bem simples, mas elas também suportam asteriscos, que podem levar a problemas com a ordem. Por exemplo, e se quiséssemos adicionar subpáginas a About, como `/about/contact` e `/about/directions`? O código a seguir não funcionará como esperado:

```
app.get('/about*', (req,res) => {  
  // envia conteúdo....  
}) app.get('/about/contact', (req,res) => {  
  // envia conteúdo....  
}) app.get('/about/directions', (req,res) => {  
  // envia conteúdo....  
})
```

Nesse exemplo, os manipuladores de `/about/contact` e `/about/directions` nunca serão encontrados porque o primeiro manipulador usa um asterisco em seu `path`: `/about*`.

O Express distingue os manipuladores das mensagens 404 e 500 pelo número de argumentos que suas funções de callback recebem. As rotas inválidas serão abordadas com detalhes no Capítulo 10 e no Capítulo 12.

Agora você pode iniciar o servidor novamente e ver se há uma homepage e uma página About funcionais.

Até aqui, não fizemos nada que não possa ser feito de maneira igualmente fácil sem o Express, mas há funcionalidades que não são imediatamente óbvias. Você deve lembrar-se de que no capítulo anterior vimos que era preciso normalizar `req.url` para determinarmos que recurso estava sendo solicitado. Tivemos de remover manualmente a `querystring` e a barra final e fazer a conversão para minúsculas. Hoje em dia, o roteador do Express está manipulando esses detalhes automaticamente. Embora não pareça grande coisa, é apenas um pouco do que o roteador do Express pode fazer.

Views e Layouts

Se você está familiarizado com o paradigma “model-view-controller”, o conceito de uma *view* não lhe é estranho. Basicamente, uma *view* é o que é distribuído para o usuário. No caso de um site, isso geralmente significa HTML, embora você também possa distribuir um PNG, um PDF ou qualquer coisa que possa ser renderizada pelo cliente. Para nossos fins, consideraremos as *views* como HTML.

Uma *view* difere de um recurso estático (como uma imagem ou um arquivo CSS) porque não precisa necessariamente ser estática: o HTML pode ser construído dinamicamente para fornecer uma página personalizada para cada requisição.

O Express dá suporte a muitos *view engines* diferentes que fornecem níveis distintos de abstração. Ele dá preferência a um *view engine* chamado *Pug* (o que não é surpresa já que o *Pug* também foi criado por TJ Holowaychuk). A abordagem que o *Pug* usa é mínima: o que escrevemos não lembra em nada HTML, o que com certeza significa muito menos digitação (sem colchetes angulares ou tags de fechamento). O *engine Pug* pega isso e converte para HTML.



Originalmente o *Pug* se chamava *Jade*, e o nome mudou com o lançamento da versão 2 devido a um problema de marca registrada.

O *Pug* é interessante, mas esse nível de abstração tem seu custo. Se você é um desenvolvedor front-end, tem de entender HTML, e entender bem, mesmo se estiver criando suas *views* no *Pug*. A maioria dos desenvolvedores

front-end que conheço não se sente à vontade com a ideia de sua principal linguagem de marcação ser abstraída. Portanto, recomendo o uso de outro framework de templating menos abstrato chamado *Handlebars*.

O Handlebars (que é baseado na popular linguagem de templating independente de linguagem Mustache) não tenta abstrair o HTML: escrevemos HTML com tags especiais que permitem que ele injete conteúdo.



Nos anos seguintes ao lançamento original deste livro, o React tomou o mundo de assalto...e abstraiu o HTML dos desenvolvedores front-end! Visto por esse prisma, minha previsão de que os desenvolvedores front-end não queriam que o HTML fosse abstraído não resistiu à passagem do tempo. No entanto, usar o JSX (extensão da linguagem JavaScript que a maioria dos desenvolvedores React usa) é (quase) idêntico a escrever HTML, logo, eu não estava totalmente errado.

Para dar suporte ao Handlebars, usaremos o pacote `express-handlebars` de Eric Ferraiuolo. No diretório de seu projeto, execute o seguinte:

```
npm install express-handlebars
```

Em seguida, em *meadowlark.js*, modifique as primeiras linhas (*ch03/02-meadowlark.js* no repositório fornecido):

```
const express = require('express')
const expressHandlebars = require('express-handlebars')

const app = express()

// configura o view engine Handlebars
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
}))
app.set('view engine', 'handlebars')
```

Esse código cria um view engine e configura o Express para usá-lo por padrão. Agora crie um diretório chamado *views* contendo um subdiretório chamado *layouts*. Se você é um desenvolvedor web experiente, provavelmente já se sente confortável com o conceito de *layouts* (às vezes chamados de *páginas mestre* [master pages]). Quando construímos um site, há certa quantidade de HTML que é igual – ou muito parecida – em cada

página. Além de ser tedioso reescrever todo esse código repetido para cada página, também cria um pesadelo em potencial para a manutenção: se você quiser alterar algo em cada página, terá de alterar *todos* os arquivos. Os layouts nos poupam disso, fornecendo uma estrutura comum para todas as páginas do site.

Criaremos então um template para nosso site. Crie um arquivo chamado *views/layouts/main.handlebars*:

```
<!doctype html>
<html>
  <head>
    <title>Meadowlark Travel</title>
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

A única coisa que talvez você ainda não tenha visto é `{{{body}}}`. Essa expressão será substituída pelo HTML de cada view. Quando criamos a instância de Handlebars, observe que especificamos o layout padrão (`defaultLayout: 'main'`). Isso significa que, a menos que você especifique outra coisa, esse será o layout usado para qualquer view.

Agora criaremos páginas de view (view pages) para nossa home page, *views/home.handlebars*:

```
<h1>Welcome to Meadowlark Travel</h1>
```

Para nossa página About, *views/about.handlebars*:

```
<h1>About Meadowlark Travel</h1>
```

Para nossa página Not Found, *views/404.handlebars*:

```
<h1>404 - Not Found</h1>
```

E para concluir para nossa página Server Error, *views/500.handlebars*:

```
<h1>500 - Server Error</h1>
```



Você pode querer que seu editor associe *.handlebars* e *.hbs* (outra extensão comum para arquivos Handlebars) ao HTML para permitir o realce da sintaxe e o uso de outros recursos do editor. No vim, você pode adicionar

a linha `au BufNewFile,BufRead *.handlebars set filetype=html` ao arquivo `~/vimrc`. Em outros editores, consulte a documentação.

Agora que temos algumas views, precisamos substituir nossas rotas antigas por rotas novas que as utilizem (*ch03/02-meadowlark.js* no repositório fornecido):

```
app.get('/', (req, res) => res.render('home'))

app.get('/about', (req, res) => res.render('about'))

// página 404 personalizada
app.use((req, res) => {
  res.status(404)
  res.render('404')
})

// página 500 personalizada
app.use((err, req, res, next) => {
  console.error(err.message)
  res.status(500)
  res.render('500')
})
```

Observe que não temos mais de especificar o tipo de conteúdo ou o código de status: o view engine retornará o tipo de conteúdo `text/html` e o código de status 200 por padrão. No manipulador genérico, que fornece nossa página 404 personalizada, e no manipulador da página 500, definimos o código de status explicitamente.

Se você iniciar seu servidor e acessar a homepage ou a página About, verá que as views foram renderizadas. Se examinar o código-fonte, notará que o boilerplate HTML¹ de *views/layouts/main.handlebars* está lá.

Sempre que você visitar a homepage, obterá o mesmo HTML, mas essas rotas são consideradas *conteúdo dinâmico*, porque podemos tomar uma decisão diferente a cada vez que a rota for chamada (o que veremos acontecer bastante posteriormente neste livro). No entanto, conteúdo que realmente não muda nunca, em outras palavras, conteúdo estático, é comum e importante, logo, consideraremos o conteúdo estático a seguir.

Arquivos e views estáticos

O Express depende do *middleware* para manipular arquivos e views estáticos. Middleware é um conceito que será abordado com mais detalhes no Capítulo 10. Por enquanto, é suficiente saber que ele fornece modularização, facilitando a manipulação de requisições.

O middleware *static* permite designar um ou mais diretórios como contendo recursos estáticos que devem simplesmente ser distribuídos para o cliente sem nenhuma manipulação especial. É nesse local que você colocará coisas como imagens, arquivos CSS e arquivos JavaScript do lado do cliente.

No diretório de seu projeto, crie um subdiretório chamado *public* (vamos chamá-lo de *public* porque qualquer coisa que estiver nesse diretório será servida para o cliente sem questionamento). Em seguida, antes de declarar qualquer rota, você adicionará o middleware *static* (*ch03/02-meadowlark.js* no repositório fornecido):

```
app.use(express.static(__dirname + '/public'))
```

O middleware *static* produz os mesmos efeitos da criação de uma rota para cada arquivo estático a ser distribuído de modo a renderizá-lo e retorná-lo para o cliente. Logo, criaremos um subdiretório *img* dentro de *public* e inseriremos nosso arquivo *logo.png* aí.

Podemos, então, simplesmente referenciar */img/logo.png* (observe que não especificamos *public*; esse diretório é invisível para o cliente), e o middleware *static* servirá esse arquivo, definindo o tipo de conteúdo apropriadamente. Agora modificaremos nosso layout para que o logotipo apareça em cada página:

```
<body>
  <header>
    
  </header>
  {{{body}}}
</body>
```



Lembre-se de que o middleware é processado em ordem, e o middleware *static* – que geralmente é declarado primeiro ou pelo menos bem no início – sobreporá outras rotas. Por exemplo, se você inserir um

arquivo *index.html* no diretório *public* (tente fazê-lo!), verá que o conteúdo desse arquivo será servido em vez da rota que foi configurada! Logo, se estiver obtendo resultados confusos, verifique seus arquivos estáticos e certifique-se de que não haja nada inesperado destinado a uma rota.

Conteúdo dinâmico em views

As views não são simplesmente uma maneira complicada de distribuir HTML estático (embora certamente também possam fazer isso). Seu poder real é poderem conter informações dinâmicas.

Digamos que quiséssemos distribuir um “biscoito da sorte virtual” na página About. Em nosso arquivo *meadowlark.js*, definiremos um array de biscoitos da sorte:

```
const fortunes = [  
  "Conquer your fears or they will conquer you.",  
  "Rivers need springs.",  
  "Do not fear what you don't know.",  
  "You will have a pleasant surprise.",  
  "Whenever possible, keep it simple.",  
]
```

Modifique a view (*/views/about.handlebars*) para exibir a previsão do biscoito:

```
<h1>About Meadowlark Travel</h1>  
{{#if fortune}}  
  <p>Your fortune for the day:</p>  
  <blockquote>{{fortune}}</blockquote>  
{{/if}}
```

Agora modifique a rota */about* para distribuir o biscoito da sorte aleatório:

```
app.get('/about', (req, res) => {  
  const randomFortune = fortunes[Math.floor(Math.random()*fortunes.length)]  
  res.render('about', { fortune: randomFortune })  
})
```

Se você reiniciar o servidor e carregar a página */about*, verá uma previsão da sorte aleatória e obterá uma nova previsão sempre que recarregar a página. O templating é muito útil e vamos abordá-lo com detalhes no Capítulo 7.

Conclusão

Criamos um site básico com o Express. Ainda que ele seja simples, contém todas as sementes necessárias para termos um site completo. No próximo capítulo, vamos colocar os pingos nos *is* como preparação para adicionar funcionalidades mais avançadas.

¹ N.T.: Boilerplate é um termo muito utilizado para se referir a trechos de código-fonte que podem ser reutilizados várias vezes, sem nenhuma ou com poucas alterações.

CAPÍTULO 4

Arrumando a casa

Nos dois capítulos anteriores, estávamos apenas experimentando: descobrindo o que é possível fazer, digamos assim. Antes de passar para funcionalidades mais complexas, vamos arrumar a casa e construir boas práticas em nosso trabalho.

Neste capítulo, começaremos realmente o projeto da Meadowlark Travel. No entanto, antes de começar a construir o site propriamente dito, nos certificaremos de ter as ferramentas de que precisamos para produzir um produto de alta qualidade.



Você não precisa seguir necessariamente o exemplo recorrente deste livro. Se estiver ansioso para construir seu próprio site, siga a estrutura do exemplo, mas modifique-a conforme apropriado para que, quando terminar o livro, tenha um site concluído!

Estrutura de arquivos e diretórios

A estruturação de aplicações tem gerado um debate acalorado e não há uma maneira correta definitiva de criá-la. No entanto, há algumas convenções comuns que é útil conhecer.

É comum tentarmos restringir o número de arquivos na raiz do projeto. Normalmente, encontramos arquivos de configuração (como *package.json*), um arquivo *README.md* e vários diretórios. Grande parte do código-fonte fica em um diretório com frequência chamado de *src*. Para simplificar, não usaremos essa convenção neste livro (surpreendentemente, nem a aplicação de scaffolding do Express a utiliza). No caso de projetos do mundo real, você pode perceber que a raiz de seu projeto está ficando confusa se estiver colocando código-fonte nela e notará que é melhor coletar esses arquivos em

um diretório como *src*.

Também mencionei que prefiro nomear o arquivo principal de minha aplicação (às vezes chamado de entry point [*ponto de entrada*]) com o nome do projeto (*meadowlark.js*), e não com algo genérico como *index.js*, *app.js* ou *server.js*.

É uma decisão em grande parte nossa a de como estruturaremos a aplicação, e recomendo o fornecimento de um roteiro da estrutura no arquivo *README.md* (ou de um texto readme acessado por link a partir desse arquivo).

No mínimo, recomendo que você sempre tenha os dois arquivos a seguir na raiz de seu projeto: *package.json* e *README.md*. O resto fica por conta de sua imaginação.

Melhores práticas

Escutamos muito a expressão *melhores práticas* atualmente, e ela significa que devemos “fazer as coisas da maneira certa” sem tomar atalhos (falaremos sobre o que isso quer dizer especificamente em breve). Certamente você já ouviu o ditado dos engenheiros de que nossas opções são “rápido”, “barato” e “bom” e podemos escolher duas delas. O que sempre me preocupou nesse modelo é que ele não leva em consideração o *acúmulo de valor* obtido quando fazemos as coisas corretamente. Na primeira vez que você fizer algo da maneira certa, pode demorar cinco vezes mais do que se o fizesse de maneira rápida e desleixada. Na segunda vez, demorará só três vezes mais. Após você ter realizado a tarefa corretamente várias vezes, a executará quase tão velozmente quanto na maneira rápida e desleixada.

Tive um instrutor de esgrima que sempre me lembrava de que a prática não leva à perfeição; a prática leva à *permanência*. Ou seja, se você realizar alguma tarefa repetidamente, ela acabará se tornando automática, será rotina. Essa afirmação é verdadeira, mas não diz nada sobre a qualidade do que estamos praticando. Se você praticar maus hábitos, eles se tornarão rotina. Em vez disso, você deve seguir a regra de que praticar o que é *certo* leva à perfeição. Seguindo essa orientação, recomendo que você acompanhe o resto dos exemplos deste livro como se estivesse criando um site real online, como

se sua reputação e remuneração dependessem da qualidade do resultado. Use o livro não só para aprender novas habilidades, mas também para praticar a construção de bons hábitos.

As práticas que enfocaremos são o controle de versões e a QA (quality assurance, garantia da qualidade). Neste capítulo, discutiremos o controle de versões; a QA será discutida no próximo capítulo.

Controle de versões

Espero não precisar convencê-lo do valor do controle de versões (se o fizesse, precisaria de um livro inteiro só sobre isso). De modo geral, o controle de versões oferece estes benefícios:

Documentação

Poder voltar no tempo percorrendo o histórico de um projeto para ver as decisões que foram tomadas e a ordem em que os componentes foram desenvolvidos pode ser uma documentação valiosa. Pode ser muito útil o projeto ter um histórico técnico.

Atribuição

Se você estiver trabalhando em equipe, a atribuição pode ser muito importante. Sempre que achar algo no código que seja obscuro ou questionável, saber quem fez essa alteração pode economizar muitas horas. Talvez os comentários associados à alteração sejam suficientes para esclarecer suas dúvidas, mas, se não forem, você saberá com quem falar.

Experimentação

Um bom sistema de controle de versões permite a experimentação. Podemos nos desviar da trajetória atual e testar algo novo, sem medo de afetar a estabilidade do projeto. Se o experimento for bem-sucedido, você poderá acrescentá-lo ao projeto, e, se não for, basta abandoná-lo.

Anos atrás, migrei para os sistemas de controle de versões distribuídos (DVCSs, distributed version control systems). Estreitei minhas opções ao Git e ao Mercurial e optei pelo Git, devido à sua onipresença e flexibilidade. Os

dois são sistemas de controle de versões excelentes e gratuitos, e recomendo que você use um deles. Neste livro, usaremos o Git, mas fique à vontade para substituí-lo pelo Mercurial (ou outro sistema de controle de versões).

Se você não conhece o Git, recomendo o ótimo livro de Jon Loeliger *Version Control with Git* (O'Reilly). Além disso, o GitHub tem uma boa listagem de recursos de aprendizagem de uso do Git (<https://try.github.io>).

Como usar o Git com este livro

Primeiro, verifique se você tem o Git. Digite `git --version`. Se o comando não responder com um número de versão, será preciso instalar o Git. Consulte a *documentação do Git* (<https://git-scm.com>) para ver as instruções de instalação.

Há duas maneiras de acompanhar os exemplos deste livro. Uma é você mesmo os digitar e acompanhar com os comandos do Git. A outra é clonar o repositório de acompanhamento que estou usando para todos os exemplos e extrair os arquivos associados a cada um deles. Algumas pessoas aprendem melhor digitando os exemplos, enquanto outras preferem apenas ver e executar as alterações sem precisar digitar tudo.

Caso esteja acompanhando por conta própria

Já temos uma estrutura preliminar para nosso projeto: algumas views, um layout, um logotipo, o arquivo principal da aplicação e um arquivo *package.json*. Seguiremos em frente criando um repositório Git e adicionando todos esses arquivos.

Primeiro, acessaremos o diretório do projeto e inicializaremos um repositório Git nele:

```
git init
```

Antes de adicionar todos os arquivos, criaremos um arquivo *gitignore* para nos ajudar e evitar a inclusão acidental de coisas que não queremos adicionar. Crie um arquivo de texto chamado *.gitignore* no diretório de seu projeto no qual possa adicionar qualquer arquivo ou diretório que queira que o Git ignore por padrão (um por linha). Esse arquivo também dará suporte a

caracteres curingas. Por exemplo, se seu editor criar arquivos de backup com um til no final (como em *meadowlark.js~*), você pode inserir **~* no arquivo *.gitignore*. Se estiver no Mac, é melhor também inserir *.DS_Store*. Além disso, é apropriado inserir *node_modules* (por razões que serão discutidas em breve). Por enquanto, o arquivo ficará assim:

```
node_modules
*~
.DS_Store
```



As entradas do arquivo *.gitignore* também são aplicáveis aos subdiretórios. Logo, se você inserir **~* no arquivo *.gitignore* na raiz do projeto, todos os arquivos de backup serão ignorados mesmo se estiverem em subdiretórios.

Agora podemos adicionar todos os arquivos existentes. Há muitas maneiras de fazer isso no Git. Geralmente prefiro usar `git add -A`, que entre todas as variantes é a mais genérica. Se você é iniciante no Git, recomendo adicionar os arquivos um a um (por exemplo, `git add meadowlark.js`) se quiser executar o commit somente de um ou dois arquivos, ou adicionar todas as suas alterações (inclusive arquivos que possa ter excluído) usando `git add -A`. Já que queremos adicionar todo o trabalho que fizemos, usaremos o seguinte:

```
git add -A
```



Iniciantes no uso do Git normalmente ficam confusos com o comando `git add`; ele adiciona *alterações* e não arquivos. Logo, se você modificar *meadowlark.js* e depois digitar `git add meadowlark.js`, o que fará na verdade é adicionar as alterações feitas.

O Git tem uma “área temporária” (staging area) para onde as alterações vão quando o comando `git add` é executado. Portanto, as alterações que adicionamos na verdade ainda não foram confirmadas, mas estão prontas para ser. Para confirmar (commit) as alterações, use `git commit`:

```
git commit -m "Initial commit."
```

A parte `-m "Initial commit."` permite escrever uma mensagem associada a esse commit. Na verdade, o Git não permite executar um commit sem uma mensagem, e com razão. Tente sempre criar mensagens significativas para o

commit; elas devem descrever de maneira breve, porém exata, o trabalho feito.

Caso esteja acompanhando com o repositório oficial

Para obter o repositório oficial deste livro, execute git clone:

```
git clone https://github.com/EthanRBrown/web-development-with-node-and-express-2e
```

Esse repositório tem um diretório para cada capítulo que contém exemplos de código. Por exemplo, o código-fonte deste capítulo pode ser encontrado no diretório ch04. Os exemplos de código de cada capítulo foram numerados para facilitar a consulta. Adicionei em todo o repositório arquivos *README.md* contendo notas adicionais sobre os exemplos.



Na primeira versão deste livro, adotei uma abordagem diferente em relação ao repositório com um histórico linear como se estivéssemos desenvolvendo um projeto cada vez mais sofisticado. Embora essa abordagem reflita a maneira como um projeto pode se desenvolver no mundo real, ela causou muita dor de cabeça, tanto para mim quanto para meus leitores. À medida que os pacotes npm mudavam, os exemplos de código também mudavam e, exceto recriando o histórico inteiro, não havia uma boa maneira de atualizar o repositório ou comentar as alterações no texto. A abordagem de um capítulo por diretório é mais artificial, porém permite que o texto fique mais sincronizado com o repositório e também facilita que ocorram contribuições da comunidade.

Conforme este livro for atualizado e melhorado, o repositório também será atualizado, e, quando o for, adicionarei uma tag de versão para permitir a extração de uma versão do repositório que corresponda à versão do livro que você estiver lendo. A versão atual do repositório é a 2.0.0. Estou tentando seguir os princípios do *versionamento semântico* aqui (falarei mais sobre isso posteriormente neste capítulo); o incremento PATCH (o último número) representa alterações menores que não devem afetar sua habilidade de fazer o acompanhamento com o livro. Ou seja, se o repositório estiver na versão 2.0.15, ela ainda deve corresponder a esta versão do livro. No entanto, se o incremento MINOR (o segundo número) for diferente (2.1.0), isso indicará que o conteúdo do repositório de acompanhamento pode divergir do que você

está lendo e pode ser melhor extrair uma tag que comece com 2.0.

O repositório de acompanhamento faz uso generoso de arquivos *README.md* para adicionar explicações aos exemplos de código.



Se em algum momento você quiser fazer um teste, lembre-se de que a tag que extrair do repositório o colocará no que o Git chama de estado “detached HEAD”. Embora você possa editar qualquer arquivo, é inseguro executar o commit de algo que tenha feito sem criar um branch antes. Logo, se quiser estabelecer um branch experimental fora de uma tag, apenas crie um novo branch e extraia-o, o que pode ser feito com o comando `git checkout -b experiment` (onde `experiment` é o nome do branch; podemos usar o que quisermos). Assim poderá editar esse branch seguramente e o quanto quiser e executar o commit.

Pacotes npm

Os pacotes npm dos quais seu projeto depende residem em um diretório chamado *node_modules*. (Foi uma escolha infeliz chamá-lo de *node_modules*, e não de *npm_packages*, já que os módulos Node são um conceito relacionado, porém diferente). Você pode explorar esse diretório e satisfazer sua curiosidade ou depurar seu programa, mas nunca deve modificar os códigos. Além de ser prática não recomendada, todas as suas alterações poderiam ser facilmente desfeitas pelo npm.

Se você precisar fazer uma modificação em um pacote do qual seu projeto depende, o curso de ação correto é criar seu próprio fork do pacote. Se tomar essa rota e sentir que suas melhorias serão úteis para outras pessoas, parabéns: agora você está envolvido em um projeto open source! Você pode enviar suas alterações, e, se elas atenderem aos padrões do projeto, serão incluídas no pacote oficial. Dar contribuições para pacotes existentes e criar builds personalizados não faz parte do escopo deste livro, mas há uma comunidade de desenvolvedores vibrante para ajudá-lo se você quiser contribuir para o desenvolvimento de pacotes existentes.

Dois objetivos importantes do arquivo *package.json* são descrever o projeto e listar suas dependências. Vá em frente e examine seu arquivo *package.json*

agora. Você deve ver algo assim (provavelmente os números de versão serão diferentes, já que esses pacotes são atualizados com frequência):

```
{
  "dependencies": {
    "express": "^4.16.4",
    "express-handlebars": "^3.0.0"
  }
}
```

Nesse momento, nosso arquivo *package.json* contém somente informações sobre dependências. O acento circunflexo (^) na frente das versões dos pacotes indica que qualquer versão que comece com o número especificado – até o próximo número principal de versão – servirá. Por exemplo, esse arquivo *package.json* indica que qualquer versão do Express que comece com 4.0.0 servirá, logo, tanto 4.0.1 quanto 4.9.9 serviriam, mas 3.4.7 não, nem 5.0.0. Essa é a especificidade de versão padrão quando usamos `npm install`, e geralmente é uma indicação muito segura. A consequência dessa abordagem é que, se você quiser passar para uma versão mais recente, terá de editar o arquivo para especificar a nova versão. Normalmente, isso é bom porque impede que alterações nas dependências invalidem o projeto sem sabermos. Os números de versão do npm são analisados por um componente chamado *semver* (abreviação de “semantic versioning”). Se quiser obter mais informações sobre o versionamento no npm, consulte a *Especificação de Versionamento Semântico* (<http://try.github.io/>) e este artigo de Tamas Piros (<http://bit.ly/34Vr3lX>).



A Especificação de Versionamento Semântico explica que um software que estiver usando o versionamento semântico deve declarar uma “API pública”. Sempre achei esse linguajar confuso; o que eles querem dizer na verdade é “alguém pode querer interagir com seu software”. Se considerarmos isso no sentido mais amplo, pode ser interpretado como qualquer coisa. Logo, não se preocupe com essa parte da especificação; os detalhes importantes são os do formato.

Já que o arquivo *package.json* lista todas as dependências, o diretório *node_modules* é na verdade um artefato derivado. Ou seja, se você o excluir, tudo que terá de fazer para que o projeto funcione novamente será executar

npm install, que recriará o diretório e inserirá nele todas as dependências necessárias. É por isso que recomendo inserir `node_modules` no arquivo `.gitignore` e não o incluir no controle de código-fonte. No entanto, algumas pessoas acham que o repositório deve conter tudo que é necessário para a execução do projeto e preferem manter `node_modules` no controle de código-fonte. Acho que ele acaba sendo um “ruído” no repositório e prefiro omiti-lo.



A partir da versão 5 do npm, um arquivo adicional, `package-lock.json`, será criado. Enquanto `package.json` é “vago” na especificação de versões de dependências (com os modificadores de versão `^` e `~`), `package-lock.json` registra as versões *exatas* que foram instaladas, o que pode ser útil se você precisar recriar as versões de dependências em seu projeto. Recomendo que você insira esse arquivo no controle de versões e não o modifique manualmente. Consulte a *documentação do package-lock.json* (<http://bit.ly/2O8IjNK>) para obter mais informações.

Metadados do projeto

A outra finalidade do arquivo `package.json` é armazenar metadados do projeto, como o nome do projeto, os autores, informações de licença e assim por diante. Se você usar o comando `npm init` para a criação inicial de seu arquivo `package.json`, ele preencherá o arquivo com os campos necessários, que poderão ser atualizados a qualquer momento. Se pretende tornar seu projeto disponível no npm ou GitHub, os metadados serão críticos. Caso queira obter mais informações sobre os campos desse arquivo, consulte a *documentação do package.json* (<http://bit.ly/2X7GVPS>). O outro componente importante dos metadados é o arquivo `README.md`. Esse arquivo pode ser um local útil para a descrição da arquitetura geral do site, assim como para qualquer informação essencial da qual alguém que não conheça o projeto possa precisar. Ele usa um formato wiki baseado em texto chamado Markdown. Verifique a *documentação do Markdown* (<http://bit.ly/2q7BQur>) para obter mais informações.

Módulos Node

Como mencionado anteriormente, os módulos Node e os pacotes npm são conceitos relacionados, porém diferentes. Os *módulos Node*, como o nome sugere, oferecem um mecanismo para a modularização e o encapsulamento. Os *pacotes npm* fornecem um esquema padronizado para armazenamento, versionamento e referenciamento de projetos (que não estão restritos aos módulos). Por exemplo, importamos o Express como um módulo no arquivo principal da aplicação:

```
const express = require('express')
```

`require` é uma função do Node para a importação de um módulo. Por padrão, o Node procura módulos no diretório *node_modules* (e não é surpresa haver um diretório *express* dentro de *node_modules*). No entanto, o Node também fornece um mecanismo para a criação de nossos próprios módulos (você nunca deve criar seus próprios módulos no diretório *node_modules*). Além dos módulos instalados em *node_modules* por um gerenciador de pacotes, há mais de 30 “módulos básicos” fornecidos pelo Node, como `fs`, `http`, `os` e `path`. Para ver a lista inteira, veja essa pergunta esclarecedora no Stack Overflow (<http://bit.ly/2NDIkKH>) e consulte a documentação oficial do Node (<https://nodejs.org/en/docs/>).

Vejamos como poderíamos modularizar a funcionalidade do biscoito da sorte que implementamos no capítulo anterior.

Primeiro criaremos um diretório para armazenar nossos módulos. Você pode chamá-lo como quiser, mas *lib* (abreviação de “library”) é uma opção comum. Nessa pasta, crie um arquivo chamado *fortune.js* (*ch04/lib/fortune.js* no repositório fornecido):

```
const fortuneCookies = [
  "Conquer your fears or they will conquer you.",
  "Rivers need springs.",
  "Do not fear what you don't know.",
  "You will have a pleasant surprise.",
  "Whenever possible, keep it simple.",
]

exports.getFortune = () => {
  const idx = Math.floor(Math.random()*fortuneCookies.length)
```

```
    return fortuneCookies[idx]
  }
```

O importante a observar aqui é o uso da variável global `exports`. Se você quiser que algo seja visto fora do módulo, terá de adicioná-lo a `exports`. Nesse exemplo, a função `getFortune` estará disponível fora do módulo, mas nosso array `fortuneCookies` ficará *totalmente oculto*. Isso é bom: o encapsulamento permite a criação de código menos propenso a erro e menos frágil.



Há várias maneiras de exportar as funcionalidades de um módulo. Abordaremos diferentes métodos no decorrer do livro e os resumiremos no Capítulo 22.

Agora, em *meadowlark.js*, podemos remover o array `fortuneCookies` (embora não haja problema em deixá-lo; ele não criará nenhum conflito com o array de mesmo nome definido em *lib/fortune.js*). É comum (mas não obrigatório) especificar as importações no início do arquivo, logo, no início do arquivo *meadowlark.js*, adicionaremos a linha a seguir (*ch04/meadowlark.js* no repositório fornecido):

```
const fortune = require('./lib/fortune')
```

Observe que prefixamos o nome do módulo com `./`. Isso indica para o Node que ele não deve procurar o módulo no diretório *node_modules*; se omitíssemos esse prefixo, essa linha falharia.

Na rota da página About, podemos utilizar o método `getFortune` a partir de nosso módulo:

```
app.get('/about', (req, res) => {
  res.render('about', { fortune: fortune.getFortune() } )
})
```

Se você está acompanhando, faremos o commit dessas alterações:

```
git add -A
git commit -m "Moved 'fortune cookie' into module."
```

Você verá que os módulos são uma maneira poderosa e fácil de encapsular funcionalidades, o que melhorará o design geral e a capacidade de manutenção de seu projeto, assim como facilitará a execução de testes. Consulte a *documentação oficial dos módulos Node* (<https://nodejs.org/api/modules.html>) para obter mais informações.



Os módulos Node também são chamados de *módulos CommonJS (CJS)*, em referência a uma especificação mais antiga na qual o Node se baseia. A linguagem JavaScript está adotando um mecanismo de pacotes oficial, chamado ECMAScript Modules (ESM). Se você tem escrito JavaScript no React ou outra linguagem de front-end progressivo, já deve conhecer o ESM, que usa `import` e `export` (em vez de `exports`, `module.exports` e `require`). Para obter mais informações, consulte a postagem de blog do Dr. Axel Rauschmayer “*ECMAScript 6 modules: the final syntax*” (<http://bit.ly/2X8ZSkM>).

Conclusão

Agora que você tem mais informações sobre o Git, o npm e os módulos, estamos prontos para discutir como gerar um produto melhor empregando boas práticas de garantia da qualidade (QA) em nossa codificação.

Recomendo que você memorize as seguintes lições deste capítulo:

- O controle de versões torna o processo de desenvolvimento de software mais seguro e previsível e aconselho você a usá-lo mesmo em pequenos projetos; assim construirá bons hábitos!
- A modularização é uma técnica importante para o gerenciamento da complexidade do software. Além de fornecer um rico ecossistema de módulos que outras pessoas desenvolveram por intermédio do npm, você pode empacotar seu próprio código em módulos para organizar melhor seu projeto.
- Os módulos Node (também chamados de CJS) usam uma sintaxe diferente dos módulos ECMAScript (ESM) e você pode ter de se alternar entre as duas sintaxes quando estiver entre código de front-end e código de back-end. É uma boa ideia se familiarizar com ambas.

Garantia da qualidade

Garantia da qualidade é uma expressão que costuma causar calafrios nos desenvolvedores – infelizmente. Afinal, você não quer criar software de qualidade? Claro que sim. Logo, não é o objetivo final que importa, e sim a condução do problema. Percebi que duas situações são comuns no desenvolvimento web:

Empresas grandes ou com bom orçamento

Costuma haver um departamento de QA e, infelizmente, um relacionamento de rivalidade surge entre a QA e o desenvolvimento. Isso é o pior que pode ocorrer. Os dois departamentos estão jogando no mesmo time, para o mesmo objetivo, mas com frequência a QA define sucesso como encontrar mais bugs, enquanto o desenvolvimento o define como gerar menos bugs, e isso serve como base para o conflito e a competição.

Empresas pequenas ou com baixo orçamento

Geralmente, não há departamento de QA; espera-se que a equipe de desenvolvimento desempenhe o papel duplo de estabelecer a garantia da qualidade e desenvolver software. Não se trata de um ridículo devaneio fantasioso ou de um conflito de interesse. No entanto, a QA é uma disciplina muito diferente do desenvolvimento e atrai personalidades e talentos distintos. Não é uma situação impossível, e certamente há desenvolvedores que têm inclinação para a QA, mas, quando o prazo final está chegando, normalmente é a QA que tem pouco tempo, para prejuízo do projeto.

Na maioria dos casos do mundo real, várias habilidades são requeridas, e está ficando cada vez mais difícil ser um especialista em todas essas habilidades.

Porém, alguma competência nas áreas pelas quais você não é diretamente responsável o tornará mais valioso para a equipe e a fará funcionar mais eficientemente. Um desenvolver que adquire habilidades de QA é um ótimo exemplo: essas duas disciplinas estão tão intimamente relacionadas que o conhecimento de sua interdisciplinaridade é muito valioso.

Também é comum a passagem de atividades normalmente executadas pela QA para o desenvolvimento, tornando os desenvolvedores responsáveis pela QA. Nesse paradigma, os engenheiros de software que se especializaram em QA agem quase como consultores dos desenvolvedores, ajudando-os a construir a QA em seu fluxo de trabalho de desenvolvimento. Sejam as funções de QA divididas ou integradas, fica claro que conhecer a QA é benéfico para os desenvolvedores.

Este livro não é para profissionais de QA; ele se destina a desenvolvedores. Logo, meu objetivo não é torná-lo um especialista em QA, e sim fornecer alguma experiência nessa área. Se sua empresa tem uma equipe de QA dedicada, minha abordagem tornará mais fácil para você se comunicar e colaborar com eles. Se não houver essa equipe, será um ponto de partida para o estabelecimento de um plano de QA abrangente para seu projeto.

Neste capítulo, você aprenderá o seguinte:

- Fundamentos da qualidade e hábitos eficazes
- Os tipos de testes (unitário e de integração)
- Como criar testes unitários com o Jest
- Como criar testes de integração com o Puppeteer
- Como configurar o ESLint para ajudá-lo a evitar erros comuns
- O que é integração contínua e onde começar a conhecê-la

Plano de QA

Em geral, o desenvolvimento é um processo criativo que propõe uma ideia e depois a transforma em realidade. A QA, por outro lado, pertence mais ao universo da validação e da ordem. Como tal, grande parte dela consiste simplesmente em *saber o que precisa ser feito* e *garantir que seja feito*. É uma disciplina apropriada para checklists, procedimentos e documentação.

Chegaria até a dizer que a principal atividade da QA não é o teste do software, e sim a *criação de um plano de garantia da qualidade abrangente e repetível*.

Recomendo a criação de um plano de QA para cada projeto, seja grande ou pequeno (sim, até mesmo para seu projeto “divertido” de fim de semana!). O plano de QA não precisa ser grande ou elaborado; você pode inseri-lo em um arquivo de texto, em um documento no processador de palavras ou em um wiki. O objetivo do plano é registrar todas as etapas executadas para assegurar que o produto funcione como esperado.

Independentemente da forma adotada, o plano de QA é um documento ativo. Você o atualizará em resposta ao seguinte:

- Novos recursos
- Alterações em recursos existentes
- Recursos removidos
- Alterações em tecnologias ou técnicas de teste
- Defeitos que não foram detectados pelo plano de QA

Esse último ponto merece menção especial. Mesmo se sua QA for robusta, defeitos ocorrerão. E, quando ocorrerem, você deve indagar “Como poderíamos ter impedido isso?”. Quando responder a essa pergunta, você poderá modificar seu plano de QA conforme apropriado para impedir futuras ocorrências desse tipo de defeito.

Você já deve ter percebido o esforço significativo que a QA envolve, e pode estar ponderando com toda razão o nível de esforço que deseja dedicar a ela.

QA: Vale a pena?

A QA pode ser cara – às vezes *muito* cara. Então, vale a pena? É uma fórmula complicada com entradas complexas. A maioria das empresas usa algum tipo de modelo de “retorno sobre o investimento”. Quando gastamos dinheiro, esperamos receber pelo menos a mesma quantia de volta (de preferência mais). Com a QA, no entanto, o relacionamento pode ser confuso. Um produto estabelecido e com boa aceitação, por exemplo, pode ser capaz de se sair bem com problemas de qualidade por mais tempo que um projeto novo e

desconhecido. É claro que ninguém *quer* criar um produto de baixa qualidade, mas as pressões sobre a tecnologia são altas. A hora de entrar no mercado (time-to-market) pode ser crítica, e às vezes entrar com algo que esteja longe do ideal é melhor do que disponibilizar o produto perfeito meses depois.

No desenvolvimento web, a qualidade pode ser dividida em quatro dimensões:

Alcance

Alcance é a penetração do produto no mercado: o número de pessoas que estão visualizando seu site ou usando seu serviço. Há uma correlação direta entre o alcance e a lucratividade: quanto maior o número de pessoas que visitarem o site, mais pessoas comprarão o produto ou serviço. Do ponto de vista do desenvolvimento, a otimização do mecanismo de busca (SEO) tem maior impacto sobre o alcance, e é por isso que incluiremos o SEO em nosso plano de QA.

Funcionalidade

Uma vez que as pessoas estiverem visitando seu site ou usando seu serviço, a qualidade da funcionalidade do site terá um grande impacto sobre a retenção do usuário; um site que funciona como propagandeado tem mais probabilidade de trazer visitantes de volta do que um em que isso não ocorra. A funcionalidade oferece a melhor oportunidade de automação de teste.

Usabilidade

Enquanto a funcionalidade está relacionada com o funcionamento correto, a usabilidade avalia a interação homem-computador (HCI, human-computer interaction). A pergunta básica é: “a funcionalidade está sendo distribuída de um modo que seja útil para o público-alvo?”. Com frequência, isso pode ser interpretado como “É fácil de usar?”, no entanto a busca da facilidade pode se opor à flexibilidade ou ao poder; o que parece fácil para um programador pode ser diferente do que parece fácil para um consumidor não técnico. Em outras palavras, você deve considerar seu público-alvo ao

avaliar a usabilidade. Já que um componente fundamental para uma avaliação de usabilidade é o usuário, geralmente a usabilidade não é algo que possa ser automatizado. Porém, o teste de usuário deve ser incluído em seu plano de QA.

Estética

A estética é a mais subjetiva das quatro dimensões e, portanto, é a menos relevante para o desenvolvimento. Embora haja poucas preocupações de desenvolvimento quando se trata da estética do site, revisões de rotina da estética devem fazer parte do plano de QA. Mostre seu site para uma amostragem representativa do público-alvo e descubra se ele parece ultrapassado ou não produz a resposta desejada. Lembre-se de que a estética depende da época (os padrões estéticos mudam com o tempo) e do público específico (o que agrada a um tipo de público pode ser totalmente desinteressante para outro).

Embora todas as quatro dimensões tenham de ser abordadas em seu plano de QA, o teste da funcionalidade e o SEO podem ser verificados automaticamente durante o desenvolvimento, logo, esse será o foco deste capítulo.

Lógica versus apresentação

Em um sentido amplo, há dois “universos estruturais” em um site: a *lógica* (com frequência chamada de *lógica do negócio*, um termo que evito por ter um viés de empreendimento comercial) e a *apresentação*. Podemos considerar a lógica do site como existindo em um domínio puramente intelectual. Por exemplo, no cenário da Meadowlark Travel, poderia haver uma regra de que o cliente deve possuir carteira de habilitação válida antes de alugar uma scooter. Essa é uma regra simples baseada em dados: para cada reserva de scooter, o usuário precisa de uma carteira de habilitação válida. A *apresentação* do cenário independe disso. Pode ser apenas uma caixa de seleção no formulário final da página do pedido ou talvez o cliente tenha de fornecer um número de carteira de habilitação válido, que será verificado pela Meadowlark Travel. É uma diferença importante, porque as coisas devem ser

muito claras e simples no domínio da lógica, enquanto a apresentação pode ser tão complicada ou tão simples quanto precisar ser. A apresentação também está sujeita a preocupações de usabilidade e estética, enquanto o domínio do negócio não.

Sempre que possível, você deve procurar traçar uma linha clara entre a lógica e a apresentação. Há muitas maneiras de fazer isso, e neste livro enfocaremos o encapsulamento da lógica em módulos JavaScript. A apresentação, por outro lado, será uma combinação de HTML, CSS, multimídia e frameworks front-end como o React, Vue ou Angular.

Tipos de testes

O tipo de teste que consideraremos neste livro se encaixa em duas categorias abrangentes: o teste unitário e o de integração (estou considerando o teste do sistema como um tipo de teste de integração). O teste unitário é muito granular, examinando componentes individuais para verificar se funcionam apropriadamente, enquanto o teste de integração verifica a interação entre vários componentes ou até mesmo no sistema inteiro.

Em geral, o teste unitário é mais útil e apropriado para o teste da lógica. O teste de integração é útil nos dois universos estruturais.

Visão geral das técnicas de QA

Neste livro, usaremos as técnicas e os programas a seguir para ter um ambiente de teste completo:

Testes unitários

Os testes unitários abordam as menores unidades de funcionalidade da aplicação, geralmente uma função individual. Quase sempre eles são criados pelos desenvolvedores, e não pela equipe de QA (mas a equipe de QA tem de poder avaliar a qualidade e a abrangência dos testes unitários). Neste livro, usaremos o Jest para os testes unitários.

Testes de integração

Os testes de integração abordam unidades de funcionalidade maiores, geralmente envolvendo várias partes da aplicação (funções, módulos, subsistemas etc.). Já que estamos construindo aplicações web, o teste de integração “definitivo” é renderizar a aplicação em um navegador, manipular esse navegador e verificar se a aplicação se comporta como esperado. Normalmente esses testes são mais difíceis de configurar e editar, e, como o foco deste livro não é a QA, teremos um único exemplo simples, usando o Puppeteer e o Jest.

*Linting*¹

Linting não é o processo de encontrar erros, e sim de encontrar *possíveis* erros. O conceito geral de linting é a identificação de áreas que possam representar possíveis erros ou de estruturas frágeis que possam levar a erros no futuro. Usaremos o ESLint para executar o linting.

Começaremos com o Jest, nosso framework de teste (que executará testes tanto unitários quanto de integração).

Instalando e configurando o Jest

Demorei um pouco para decidir que framework de teste usaria neste livro. O Jest começou sua trajetória como um framework para testar aplicações React (e continua sendo a opção óbvia para isso), mas não é específico do React e é um ótimo framework de teste de uso geral. Claro que não é o único: o *Mocha* (<https://mochajs.org>), o *Jasmine* (<https://jasmine.github.io>), o *Ava* (<https://github.com/avajs/ava>) e o *Tape* (<https://github.com/substack/tape>) também são ótimas opções.

Acabei escolhendo o Jest porque acho que ele oferece a melhor experiência geral (uma opinião baseada na sua excelente pontuação na pesquisa *State of JavaScript 2018* (<http://bit.ly/33ErHUE>)). Dito isso, há várias semelhanças entre os framework de teste mencionados aqui, logo, você deve poder pegar o que aprendeu e aplicar ao seu framework de teste preferido.

Para instalar o Jest, execute o comando a seguir a partir da raiz do projeto:

```
npm install --save-dev jest
```

(Observe que usamos `--save-dev` aqui; estamos informando ao npm que essa é uma dependência de desenvolvimento e não é essencial para a aplicação funcionar; ela será listada na seção `devDependencies` do arquivo *package.json* em vez de na seção `dependencies`).

Antes de prosseguir, precisamos de uma maneira de executar o Jest (que executará os testes de nosso projeto). A maneira convencional de fazer isso é adicionando um script a *package.json*. Edite *package.json* (*ch05/package.json* no repositório fornecido) e modifique a propriedade `scripts` (ou adicione-a se não existir):

```
"scripts": {  
  "test": "jest"  
},
```

Agora você pode executar todos os testes em seu projeto simplesmente digitando:

```
npm test
```

Se tentar fazer isso nesse momento, provavelmente verá a mensagem de erro de que não há testes configurados... porque ainda não adicionamos nenhum. Logo, criaremos alguns testes unitários!



Normalmente, quando adicionamos um script ao arquivo *package.json*, o executamos com `npm run`. Por exemplo, se você adicionasse um script `foo`, digitaria `npm run foo` para executá-lo. No entanto, o script `test` é tão comum que o npm saberá como executá-lo mesmo se digitarmos apenas `npm test`.

Teste unitário

Agora voltaremos nossa atenção para o teste unitário. Já que o teste unitário dá ênfase ao isolamento de uma única função ou componente, primeiro temos de conhecer o `mocking`, uma técnica importante para chegarmos a esse isolamento.

Mocking²

Um dos desafios com o qual você deparará com frequência é como escrever

código que seja “testável.” Em geral, códigos que tentam fazer muitas coisas ou pressupõem várias dependências são mais difíceis de testar do que códigos dedicados que pressupõem poucas dependências ou a ausência delas.

Sempre que há uma dependência, temos algo que precisa ser *representado*³ (simulado) para o teste ser eficaz. Por exemplo, nossa principal dependência é o Express, que já foi plenamente testado, logo, não precisamos ou queremos testar o Express propriamente dito, apenas *como o estamos usando*. A única maneira pela qual podemos determinar se estamos usando o Express corretamente é simulando o Express.

As rotas que temos atualmente (a home page, a página About, a página 404 e a página 500) são muito difíceis de testar porque pressupõem três dependências relacionadas ao Express: elas pressupõem que temos o aplicativo Express (para podermos ter `app.get`), e também que temos objetos de requisição e resposta. Felizmente, é muito fácil eliminar a dependência relacionada ao aplicativo Express (é mais difícil fazê-lo com os objetos de requisição e resposta...falaremos sobre isso posteriormente). Ainda bem que não estamos usando muitas funcionalidades do objeto de resposta (estamos usando apenas o método `render`), logo, será fácil simulá-lo, o que veremos em breve.

Refatorando a aplicação para melhorar a testabilidade

Na verdade ainda não temos muito código em nossa aplicação para testar. Até agora, adicionamos apenas alguns manipuladores de rota e a função `getFortune`.

Para tornar nosso aplicativo mais testável, vamos *extrair* os manipuladores de rota e enviá-los para sua própria biblioteca. Crie um arquivo *lib/handlers.js* (*ch05/lib/handlers.js* no repositório fornecido):

```
const fortune = require('./fortune')

exports.home = (req, res) => res.render('home')

exports.about = (req, res) =>
  res.render('about', { fortune: fortune.getFortune() })
```

```
exports.notFound = (req, res) => res.render('404')
```

```
exports.serverError = (err, req, res, next) => res.render('500')
```

Agora podemos recriar o arquivo *meadowloark.js* da aplicação para usar esses manipuladores (*ch05/meadowlark.js* no repositório fornecido):

```
// normalmente no início do arquivo
```

```
const handlers = require('./lib/handlers')
```

```
app.get('/', handlers.home)
```

```
app.get('/about', handlers.about)
```

```
// página 404 personalizada
```

```
app.use(handlers.notFound)
```

```
// página 500 personalizada
```

```
app.use(handlers.serverError)
```

Ficou mais fácil testar esses manipuladores: eles são apenas funções que recebem objetos de requisição e resposta, e precisamos verificar se estamos usando esses objetos corretamente.

Criando nosso primeiro teste

Há várias maneiras de identificar testes para o Jest. As duas mais comuns são inserir os testes em subdiretórios chamados `__test__` (dois underscores antes e depois de *test*) e nomear os arquivos com a extensão *.test.js*. Gosto de combinar as duas técnicas porque considero que cada uma tem um objetivo. Inserir os testes em diretórios `__test__` evita que eles desorganizem os diretórios de código-fonte (caso contrário, tudo parecerá duplicado no diretório de código-fonte... você terá um *foo.test.js* para cada arquivo *foo.js*), e usar a extensão *.test.js* significa que, se eu estiver olhando para as diversas abas de meu editor, poderei ver imediatamente o que é um teste e o que é código-fonte.

Criaremos então um arquivo chamado *lib/__tests__/handlers.test.js* (*ch05/lib/__tests__/handlers.test.js* no repositório fornecido):

```
const handlers = require('../handlers')
```



```
test('home page renders', () => {  
  const req = {}  
  const res = { render: jest.fn() }  
  handlers.home(req, res)  
  expect(res.render.mock.calls[0][0]).toBe('home')  
})
```

Se você é iniciante na execução de testes, isso deve estar parecendo bem estranho, logo, vamos analisar.

Primeiro, importamos o código que estamos tentando testar (nesse caso, os manipuladores de rota). Em seguida, cada teste tem uma descrição; estamos tentando descrever o que está sendo testado. Nesse caso, queremos nos certificar se a home page está sendo renderizada.

Para chamar a renderização, precisamos de objetos de requisição e resposta. Teríamos de escrever código a semana inteira se quiséssemos simular os objetos de requisição e resposta em sua totalidade, mas felizmente não precisamos de muita coisa deles. Sabemos que não precisamos de nada do objeto de requisição nesse caso (então usaremos apenas um objeto vazio) e a única coisa que precisamos do objeto de resposta é um método de renderização. Observe como a função de renderização é construída: simplesmente chamamos um método do Jest denominado *jest.fn()*. Isso cria uma função de mocking genérica que registra como ela está sendo chamada.

Para concluir, chegamos à parte importante do teste: as asserções. Resolvemos tudo que era preciso para chamar o código que estamos testando, mas como verificar se ele fez o que deveria? No caso em questão, o que o código deve fazer é chamar o método *render* do objeto de resposta com a string *home*. A função de mocking do Jest registra todas as vezes em que foi chamada, logo, só precisamos verificar se ela foi chamada exatamente uma vez (provavelmente seria um problema se fosse chamada duas vezes), que é o que o primeiro método *expect* faz, e se foi chamada com *home* como seu primeiro argumento (o primeiro índice do array especifica a chamada e o segundo especifica o argumento).



Pode ser tedioso reexecutar continuamente os testes sempre que você

fizer uma alteração no código. Felizmente, a maioria dos frameworks de teste tem um “modo watch (observação)” que monitora constantemente o código, procura alterações e reexecuta os testes automaticamente. Para executar seus testes no modo watch, digite `npm test -- --watch` (o traço duplo adicional é necessário para que o npm saiba que deve passar o argumento `--watch` para o Jest).

Altere o manipulador `home` para que renderize algo diferente da `home view`; você notará que agora seu teste falhará e um bug será detectado!

Agora podemos adicionar testes para nossas outras rotas:

```
test('about page renders with fortune', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.about(req, res)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('about')
  expect(res.render.mock.calls[0][1])
    .toEqual(expect.objectContaining({
      fortune: expect.stringMatching(/^W/),
    }))
})
```

```
test('404 handler renders', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.notFound(req, res)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('404')
})
```

```
test('500 handler renders', () => {
  const err = new Error('some error')
  const req = {}
  const res = { render: jest.fn() }
  const next = jest.fn()
  handlers.serverError(err, req, res, next)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('500')
```

})

Observe que há alguma funcionalidade adicional nos testes da página “about” e de erro do servidor. A função de renderização de “about” é chamada com a sorte tirada no biscoito, logo, incluímos a expectativa de que seja uma previsão de sorte na forma de uma string contendo pelo menos um caractere. Não faz parte do escopo deste livro descrever todas as funcionalidades que estão disponíveis com o Jest e seu método `expect`, mas você pode encontrar uma documentação abrangente na *home page do Jest* (<https://jestjs.io>). Repare que o manipulador de erro do servidor recebe quatro argumentos, e não dois, portanto, tivemos de fornecer mais mocks.

Manutenção do teste

Você deve ter percebido que nos testes não temos uma simples questão de “configurar e esquecer”. Por exemplo, se renomeássemos nossa “home” view por razões legítimas, o teste falharia, e teríamos de corrigir não só o teste como também o código.

As equipes se esforçam muito para definir expectativas realistas sobre como devem ser os testes e o nível de especificidade que eles devem ter. Por exemplo, não tivemos de verificar se o manipulador de “about” estava sendo chamado com a sorte tirada... o que nos poupou de precisar corrigir o teste caso abandonássemos esse recurso.

Não posso dar muitos conselhos sobre o que seria um teste completo do código. O esperado é que você tenha padrões para testar um código da área de aviãoica ou de equipamentos médicos muito diferentes dos que teria para testar o código existente por trás de um site de marketing.

O que posso oferecer é uma maneira de responder à pergunta “Até onde meu código deve ser testado?”. A resposta para ela chama-se *cobertura de código*, que é o que discutiremos a seguir.

Cobertura de código

A cobertura de código oferece uma resposta quantitativa para até onde o código deve ser testado, mas, como na maioria dos tópicos em programação, não há respostas simples.

O Jest fornece alguma análise de cobertura de código automatizada. Para ver até onde seu código deve ser testado, execute o seguinte:

```
npm test -- --coverage
```

Se você está acompanhando, deve ver vários números de cobertura “100%” tranquilizadamente verdes para os arquivos de *lib*. O Jest relata o percentual de cobertura de instruções (Stmts, statements), branches, funções (Funcs) e linhas.

As instruções se referem aos comandos JavaScript, como cada expressão, instrução de fluxo de controle etc. Observe que você poderia ter 100% de cobertura de linhas, mas não chegar a 100% de cobertura de instruções porque é possível inserir várias instruções em uma única linha em JavaScript. A cobertura de branches se refere às instruções de fluxo de controle, como if-else. Se você tiver uma instrução if-else e seu teste procurar só a parte if, haverá 50% de cobertura de branches para essa instrução.

Você deve ter notado que *meadowlark.js* não tem 100% de cobertura. Isso não é necessariamente um problema; se você examinar nosso arquivo *meadowlark.js* refatorado, verá que grande parte do que existe nele agora é simplesmente configuração... estamos apenas unindo as diferentes partes. Estamos configurando o Express com o middleware relevante e iniciando o servidor. Além de ser difícil testar esse código de maneira significativa, é um argumento válido o de que não devemos fazê-lo já que ele está simplesmente montando código já testado.

Você poderia até mesmo argumentar que os testes que criamos até agora não são particularmente úteis; eles também estão apenas verificando se estamos configurando o Express corretamente.

Novamente, não tenho respostas fáceis. No fim das contas, o tipo de aplicação que você estiver construindo, seu nível de experiência e o tamanho e a configuração de sua equipe terão um grande impacto sobre até onde se aprofundará no alcance de seu teste. Recomendo que, se errar, seja mais para o lado do teste *excessivo* do que para o lado do teste *insuficiente*, porém, ao adquirir mais experiência, encontrará o equilíbrio “correto”.

Testando a funcionalidade entrópica

Testar a funcionalidade *entrópica* (funcionalidade que é aleatória) apresenta seus próprios desafios. Outro teste que poderíamos adicionar ao nosso gerador de biscoitos da sorte seria um que verificasse se ele retorna um biscoito da sorte *aleatório*. No entanto, como saber se algo é aleatório? Uma abordagem é obter um grande número de previsões da sorte – mil, por exemplo – e medir a distribuição das respostas. Se a função for apropriadamente aleatória, nenhuma resposta terá destaque. A desvantagem dessa abordagem é que ela é não determinística: é possível (mas improvável) obter a mesma previsão da sorte com uma frequência 10 vezes maior do que qualquer outra previsão. Se tal fato ocorresse, o teste poderia falhar (dependendo do rigor com que você definir o limite do que é “aleatório”), mas talvez isso não indicasse que o sistema que está sendo testado está falhando; é apenas uma consequência do teste em sistemas entrópicos. No caso de nosso gerador de previsões da sorte, seria razoável gerar 50 previsões e esperar retirar pelo menos três diferentes. Por outro lado, se estivéssemos desenvolvendo uma fonte aleatória para uma simulação científica ou componente de segurança, provavelmente seria melhor termos testes muito mais detalhados. O que importa é que testar a funcionalidade entrópica é difícil e requer mais reflexão.

Teste de integração

Atualmente não há nada de interessante para testarmos em nossa aplicação; temos apenas algumas páginas e não há interação. Logo, antes de criar um teste de integração, adicionaremos alguma funcionalidade que possamos testar. Na tentativa de manter a simplicidade, essa funcionalidade será um link que nos permita ir da home page para a página About. Não poderia ser mais simples! E, mesmo parecendo tão simples para o usuário, trata-se de um verdadeiro teste de integração porque está verificando não só dois manipuladores de rota do Express, mas também o HTML e a interação com o DOM (o usuário clicar no link e a resultante navegação nas páginas). Adicionaremos um link a *views/home.handlebars*:

```
<p>Questions? Checkout out our  
<a href="/about" data-test-id="about">About Us</a> page!</p>
```

Você deve estar querendo saber por que usei o atributo `data-test-id`. Para executar o teste, precisamos de alguma maneira identificar o link para poder clicar (virtualmente) nele. Poderíamos ter usado uma classe CSS, mas prefiro reservar as classes para a estilização e usar atributos de dados para a automação. Também poderíamos ter procurado o texto *About Us*, mas essa seria uma busca frágil e dispendiosa com o DOM. Ou poderíamos ter consultado o parâmetro `href`, o que faria sentido (mas seria mais difícil fazer esse teste falhar, e precisamos fazer isso para fins de aprendizagem).

Podemos prosseguir; executaremos nossa aplicação e verificaremos com nossas desajeitadas mãos humanas se a funcionalidade está operando como esperado antes de passar para algo mais automatizado.

Antes de instalar o Puppeteer e criar um teste de integração, temos de modificar a aplicação para que possa ser requerida como um módulo (ela foi projetada para ser executada diretamente). A maneira de fazer isso no Node é um pouco hermética: no fim de *meadowlark.js*, substitua a chamada a `app.listen` pelo seguinte:

```
if(require.main === module) {
  app.listen(port, () => {
    console.log( `Express started on http://localhost:${port}` +
      '; press Ctrl-C to terminate.' )
  })
} else {
  module.exports = app
}
```

Pularei a explicação técnica dessa operação porque ela é tediosa, mas, se você estiver interessado, a leitura cuidadosa da *documentação dos módulos do Node* (<http://bit.ly/32BDO3H>) a esclarecerá. O que é importante saber é que, se você executar um arquivo JavaScript diretamente com o node, `require.main` será igual ao objeto global `module`; caso contrário, o arquivo será importado de outro módulo.

Agora que resolvemos isso, podemos instalar o Puppeteer. Ele é basicamente uma versão headless e controlável do Chrome (*Headless* significa simplesmente que o navegador pode ser executado sem renderizar uma UI na tela). Para instalar o Puppeteer, use o comando a seguir:

```
npm install --save-dev puppeteer
```

Também instalaremos um utilitário pequeno para encontrar uma porta aberta e não vermos várias mensagens de erro de teste porque o aplicativo não pôde ser iniciado na porta que solicitamos:

```
npm install --save-dev portfinder
```

Já podemos criar um teste de integração que faça o seguinte:

1. Execute o servidor da aplicação em uma porta desocupada
2. Inicie um navegador Chrome headless e abra uma página
3. Navegue para a home page da aplicação
4. Encontre um link com `data-test-id="about"` e clique nele
5. Espere a navegação ocorrer
6. Verifique se estamos na página `/about`

Crie um diretório chamado *integration-tests* (fique à vontade para dar outro nome se quiser) e um arquivo nesse diretório chamado *basic-navigation.test.js* (*ch05/integration-tests/basic-navigation.test.js* no repositório fornecido):

```
const portfinder = require('portfinder')
const puppeteer = require('puppeteer')

const app = require('../meadowlark.js')

let server = null
let port = null

beforeEach(async () => {
  port = await portfinder.getPortPromise()
  server = app.listen(port)
})

afterEach(() => {
  server.close()
})

test('home page links to about page', async () => {
  const browser = await puppeteer.launch()
```

```
const page = await browser.newPage()
await page.goto(`http://localhost:${port}`)
await Promise.all([
  page.waitForNavigation(),
  page.click('[data-test-id="about"]'),
])
expect(page.url()).toBe(`http://localhost:${port}/about`)
await browser.close()
})
```

Estamos usando os hooks `beforeEach` e `afterEach` do Jest para iniciar nosso servidor antes de cada teste e interrompê-lo após o teste (atualmente temos apenas um teste, logo, isso será importante quando adicionarmos outros). Também poderíamos usar `beforeAll` e `afterAll` para não iniciar e interromper o servidor a cada teste, o que talvez os acelerasse, mas não teríamos um ambiente “limpo” para cada teste. Ou seja, se um dos testes fizer alterações que afetem o resultado de testes futuros, você estará introduzindo dependências de difícil manutenção.

Nosso teste usa a API do Puppeteer, que fornece várias funcionalidades de consulta do DOM. Observe que quase tudo aqui é assíncrono, e estamos usando `await` liberalmente para tornar o teste mais fácil de ler e gravar (quase todas as chamadas de API do Puppeteer retornam uma promessa).⁴ Encapsulamos a navegação e o clique juntos em uma chamada a `Promise.all` para evitar condições `race`⁵ de acordo com a documentação do Puppeteer.

Há mais funcionalidades na API do Puppeteer do que seria possível abordar neste livro. Felizmente, ele tem uma *ótima documentação* (<http://bit.ly/2KctokI>).

A execução de testes é uma precaução vital para assegurarmos a qualidade do produto, mas não é a única ferramenta à disposição. O linting ajuda principalmente a evitar erros comuns.

Linting

Um bom linter é como se tivéssemos um segundo par de olhos: ele identificará coisas que nossos cérebros humanos deixarão passar. O linter JavaScript original é o JSLint de Douglas Crockford. Em 2011, Anton

Kovalyov criou um fork do JSLint e assim nasceu o JSHint. Kovalyov achava que o JSLint estava se tornando muito inflexível e queria criar um linter JavaScript mais personalizável desenvolvido pela comunidade. Após o JSHint veio o *ESLint* (<https://eslint.org>) de Nicholas Zakas, que se tornou a opção mais popular (ele venceu com a maioria dos votos na *pesquisa State of JavaScript 2017* (<http://bit.ly/2Q7w32O>)). Além de sua onipresença, o ESLint parece ser o linter que tem manutenção mais ativa e prefiro sua configuração flexível à do JSHint, logo, é o que recomendo.

O ESLint pode ser instalado por projeto ou globalmente. Para não danificar algo inadvertidamente, tento evitar instalações globais (por exemplo, se eu instalar o ESLint globalmente e atualizá-lo com frequência, projetos antigos podem não conseguir mais executar o linting com sucesso devido a alterações incompatíveis, e tenho de executar a tarefa adicional de atualizar meu projeto).

Para instalar o ESLint em seu projeto, execute:

```
npm install --save-dev eslint
```

O ESLint requer um arquivo de configuração que lhe diga que regras aplicar. Criar esse arquivo a partir do zero seria uma tarefa demorada, mas felizmente o ESLint fornece um utilitário para a criação automática de um. Na raiz do projeto, execute o seguinte:

```
./node_modules/.bin/eslint --init
```



Se instalássemos o ESLint globalmente, poderíamos usar apenas `eslint --init`. O estranho path `./node_modules/.bin` é requerido para a execução local direta de utilitários instalados. Veremos em breve que não é preciso fazer isso se adicionarmos os utilitários à seção `scripts` do arquivo `package.json`, o que é recomendado para tarefas executadas com frequência. No entanto, criar uma configuração para o ESLint é algo que só temos de fazer uma vez a cada projeto.

O ESLint fará algumas perguntas. Para a maioria delas, é seguro selecionar os padrões, mas há as que precisam de explicação:

Que tipo de módulos seu projeto usa?

Já que estamos usando o Node (e não código que será executado no

navegador), é melhor selecionar “CommonJS (require/exports)”. Você também pode ter JavaScript no lado do cliente em seu projeto, caso em que talvez prefira uma configuração de lint separada. A maneira mais fácil de resolver isso é criar dois projetos distintos, mas podemos ter várias configurações para o ESLint no mesmo projeto. Consulte a *documentação do ESLint* (<https://eslint.org/>) para obter mais informações.

Que framework seu projeto usa?

A menos que você veja o Express entre as opções (não vi quando este texto foi escrito), selecione “None of these”.

Onde seu código é executado?

Selecione Node.

Agora que o ESLint está configurado, precisamos de uma maneira conveniente de executá-lo. Adicione o seguinte à seção scripts de *package.json*:

```
"lint": "eslint meadowlark.js lib"
```

Observe que temos de informar explicitamente ao ESLint em que arquivos e diretórios queremos executar o linting. Esse é um bom argumento para reunirmos todo o código-fonte sob o mesmo diretório (geralmente *src*).

Seja corajoso e execute o código a seguir:

```
npm run lint
```

Provavelmente você verá muitos erros de aparência desagradável – geralmente é isso que acontece na primeira vez que executamos o ESLint. No entanto, se está acompanhando o teste com o Jest, haverá alguns erros espúrios relacionados a esse programa, como:

```
3:1  error 'test' is not defined  no-undef
5:25 error 'jest' is not defined  no-undef
7:3  error 'expect' is not defined no-undef
8:3  error 'expect' is not defined no-undef
11:1 error 'test' is not defined  no-undef
13:25 error 'jest' is not defined  no-undef
15:3  error 'expect' is not defined no-undef
```

O ESLint não gosta (com toda a razão) de variáveis globais não reconhecidas. O Jest injeta variáveis globais (notadamente `test`, `describe`, `jest` e `expect`). Felizmente, esse é um problema fácil de corrigir. Na raiz do projeto, abra o arquivo `.eslintrc.js` (ele contém a configuração do ESLint). Na seção `env`, adicione o seguinte:

```
"jest": true,
```

Se você executar `npm run lint` novamente, deve ver menos erros.

Mas o que fazer com os outros erros? Nesse caso posso oferecer apenas conselhos em vez de uma orientação específica. Em geral, um erro de linting tem uma entre três causas:

- É um problema legítimo e você deve corrigi-lo. Nem sempre ele é óbvio, caso em que pode ser preciso consultar a documentação do ESLint à procura do erro específico.
- É uma regra com a qual não concordamos e podemos simplesmente a desativar. Muitas das regras do ESLint são apenas uma questão de opinião. Demonstrarei a desativação de uma regra em breve.
- Concordamos com a regra, mas há casos em que é inviável usá-la ou ela é difícil de corrigir em alguma situação específica. Nessas situações, podemos desativar as regras apenas para determinadas linhas de um arquivo, caso para o qual também veremos um exemplo.

Se você está acompanhando, no momento deve ver os seguintes erros:

```
/Users/ethan/wdne2e-companion/ch05/meadowlark.js
27:5  error  Unexpected console statement  no-console
```

```
/Users/ethan/wdne2e-companion/ch05/lib/handlers.js
10:39 error  'next' is defined but never used  no-unused-vars
```

O ESLint está reclamando do logging do console porque essa não é necessariamente uma boa maneira de fornecer saída para a aplicação; pode ter ruído, ser inconsistente e, dependendo de como você executar, a saída pode passar despercebida. No entanto, para nosso uso, suponhamos que isso não importasse e quiséssemos desativar essa regra. Abra seu arquivo `.eslintrc`, encontre a seção `rules` (se não houver uma seção `rules`, crie uma no início do objeto exportado) e adicione a regra a seguir:

```
"rules": {  
  "no-console": "off",  
},
```

Se você executar `npm run lint` novamente, não verá mais esse erro! O próximo erro é um pouco mais complicado...

Abra `lib/handlers.js` e considere a linha em questão:

```
exports.serverError = (err, req, res, next) => res.render('500')
```

O ESLint está correto; fornecemos `next` como argumento, mas não estamos fazendo nada com ele (também não estamos fazendo nada com `err` e `req`, mas, devido à maneira como o JavaScript trata argumentos de função, tivemos de inserir *algo* aí para chegar a `res`, que *estamos* usando).

Você pode ficar tentado a apenas remover o argumento `next`. “O que há de errado nisso?” poderia pensar. Realmente não haveria erros de runtime e o linter ficaria satisfeito... mas um erro difícil de ver ocorreria: seu manipulador de erros personalizado pararia de funcionar! (Se quiser ver por si próprio, lance uma exceção a partir de uma de suas rotas, tente visitá-la e, em seguida, remova o argumento `next` do manipulador de `serverError`).

O Express está fazendo algo sutil aqui: está usando o número de argumentos passados para reconhecer que esse deve ser um manipulador de erro. Sem o argumento `next` – sendo ele usado ou não –, o Express não saberá mais que se trata de um manipulador de erro.



O que a equipe do Express fez com o manipulador de erro é inegavelmente “inteligente”, mas códigos inteligentes com frequência podem ser confusos, falhar facilmente ou ser inescrutáveis. Mesmo adorando o Express, acho que essa foi uma escolha errada da equipe: eles deveriam ter encontrado uma maneira menos idiossincrática e mais explícita de especificar um manipulador de erro.

Não podemos alterar o código de nosso manipulador, e precisamos do manipulador, mas gostamos dessa regra e não queremos desativá-la. Poderíamos apenas conviver com o erro, mas os erros acumularão sendo uma irritação constante e acabarão tornando inútil termos um linter. Felizmente, podemos corrigir isso desativando essa regra somente para essa linha. Edite `lib/handlers.js` e adicione o seguinte circundando seu manipulador de erro:

```
// O Express reconhece o manipulador de erro pelos seus
// quatro argumentos, logo, temos de desativar a regra
// de variáveis não usadas do ESLint
/* eslint-disable no-unused-vars */
exports.serverError = (err, req, res, next) => res.render('500')
/* eslint-enable no-unused-vars */
```

O linting pode ser um pouco frustrante no início – pode parecer que ele está sempre tentando enganá-lo. E é claro que você deve ficar à vontade para desativar regras que não lhe atendam. Porém, vai achá-lo cada vez menos frustrante quando aprender a evitar os erros comuns que ele foi projetado para detectar.

Sem dúvida os testes e o linting são úteis, mas nenhuma ferramenta ajudará se você não a usar! Pode parecer despropositado você passar pelo processo demorado e problemático de criar testes unitários e acabar executando o linting, mas já vi isso acontecer, principalmente quando há pressão para resolver algo. Felizmente, há uma maneira de assegurar que essas úteis ferramentas não sejam esquecidas: a integração contínua.

Integração contínua

Mostrarei outro conceito de QA extremamente útil: a integração contínua (CI, continuous integration). A CI será particularmente importante se você estiver trabalhando em equipe, mas, mesmo se estiver trabalhando sozinho, ela fornecerá alguma orientação útil.

Basicamente, a CI executa alguns de nossos testes sempre que contribuimos com código para um repositório de código-fonte (podemos controlar a que branches isso será aplicado). Quando todos os testes têm um bom resultado, geralmente nada acontece (podemos receber um email dizendo “bom trabalho”, dependendo de como a CI for configurada).

Quando, por outro lado, há falhas, geralmente as consequências são mais... públicas. Novamente, depende de como você configurar sua CI, mas a equipe inteira costuma receber um email dizendo que o “build está danificado”. Se o responsável pela integração tiver inclinações sádicas, seu chefe também entrará na lista de emails! Soube de equipes que ligam luzes e sirenes quando

alguém danifica o build, e em um escritório particularmente criativo, um minúsculo e robótico lançador de mísseis de espuma dispara projéteis macios na direção do desenvolvedor ofensor! É um incentivo poderoso para executarmos nosso conjunto de ferramentas de QA antes de executar um commit.

Não faz parte do escopo deste livro abordar a instalação e a configuração de um servidor de CI, mas um capítulo sobre QA não estaria completo sem mencioná-las.

Atualmente, o servidor de CI mais popular para projetos Node é o *Travis CI* (<https://travisci.org/>). Trata-se de uma solução hospedada que pode ser interessante (ela nos poupa de configurar nosso próprio servidor de CI). Se você está usando o GitHub, ele oferece um ótimo suporte à integração. O *CircleCI* (<https://circleci.com>) é outra opção.

Se está trabalhando em um projeto sozinho, talvez não se beneficie tanto de um servidor de CI, mas, se está em uma equipe ou em um projeto open source, recomendo usar a CI.

Conclusão

Este capítulo abordou muita coisa, mas considero essas habilidades do mundo real essenciais em qualquer framework de desenvolvimento. O ecossistema JavaScript é enorme, e, se você é iniciante, pode ser difícil saber onde começar. Espero que o capítulo tenha lhe indicado a direção certa.

Agora que temos alguma experiência com essas ferramentas, voltaremos nossa atenção para certos aspectos básicos dos objetos do Node e Express que envolvem tudo que acontece em uma aplicação Express: os objetos de requisição e resposta.

¹ N.T.: Linting é o processo de análise de código para localizar erros potenciais no código-fonte.

² N.T.: Mocking é o processo para criar objetos que simulam o comportamento de objetos reais complexos, difíceis ou impossíveis de serem incorporados no teste unitário.

³ N.T.: Em inglês, mocked.

⁴ Se você não conhece await, recomendo *esse artigo* (<http://bit.ly/2rEXU0d>) de Tamas

Piros.

- 5 Uma condição race (race condition) ocorre quando duas ou mais threads acessam dados compartilhados e tentam efetuar alterações ao mesmo tempo.


Objetos de requisição e resposta

Neste capítulo, aprenderemos os importantes detalhes dos objetos de requisição e resposta – que são o início e o fim de tudo que acontece em uma aplicação Express. Quando você estiver construindo um servidor web com o Express, grande parte do que fará começará com um objeto de requisição e terminará com um objeto de resposta.

Esses dois objetos se originam no Node e são estendidos pelo Express. Antes de ver o que eles oferecem, forneceremos uma breve explicação de como um cliente (geralmente um navegador) requisita uma página a um servidor e como essa página é retornada.

As partes de uma URL

Vemos URLs o tempo todo, mas geralmente não paramos para pensar sobre as partes que as compõem. Consideraremos três URLs e examinaremos seus componentes:



https://	google.com			#q=express	
http://	www.bing.com		/search	?q=grunt&first=9	
http://	localhost	:3000	/about	?test=1	#history
https://	google.com		/		#q=express
protocolo	nome do host	porta	path	querystring	fragmento

Protocolo

O protocolo determina como a requisição será transmitida. Lidaremos exclusivamente com *http* e *https*. Outros protocolos comuns são *file* e *ftp*.

Host

O host identifica o servidor. Os servidores de um computador (localhost) ou de uma rede local podem ser identificados por uma palavra ou por um endereço IP numérico. Na internet, o host termina com um domínio de nível superior (TLD, top-level domain) como *.com* ou *.net*. Além disso, pode haver *subdomínios*, incluídos como prefixo do nome do host. *www* é um subdomínio comum, mas poderia ser qualquer outro. Os subdomínios são opcionais.

Porta

Cada servidor tem uma coleção de portas numeradas. Alguns números de porta são especiais, como 80 e 443. Se você omitir a porta, a porta 80 será assumida para o HTTP e a 443 para o HTTPS. Em geral, quando não usamos a porta 80 ou 443, devemos usar um número de porta maior que 1023.¹ É comum o uso de números de porta fáceis de lembrar como 3000, 8080 e 8088. Só um servidor pode estar associado a uma porta específica, e, ainda que haja muitos números para escolher, você pode ter de alterar o número da porta se estiver empregando um número normalmente usado.

Path

Geralmente o path é a primeira parte da URL importante para o aplicativo (é possível tomar decisões baseadas no protocolo, no host e na porta, mas não é boa prática). O path deve ser usado para identificar as páginas ou outros recursos do aplicativo de maneira exclusiva.

Querystring

A querystring é uma coleção opcional de pares nome/valor. Ela começa com um ponto de interrogação (?), e os pares nome/valor são separados por ampersands (&). Tanto os nomes quanto os valores devem ser *codificados em URL*. O JavaScript fornece uma função interna que faz isso: `encodeURIComponent`. Por exemplo, os espaços são substituídos por sinais de adição (+). Outros caracteres especiais são substituídos por referências de caracteres numéricos. A querystring também é chamada de *string de pesquisa* ou simplesmente *pesquisa*.

Fragmento

O fragmento (ou *hash*) não é passado para o servidor; ele é de uso estrito do navegador. Algumas aplicações de página única utilizam o fragmento para controlar a navegação na aplicação. Originalmente, a única finalidade do fragmento era fazer o navegador exibir uma parte específica do documento, marcada por uma tag de âncora (por exemplo: ``).

Métodos da requisição HTTP

O protocolo HTTP define uma coleção de *métodos de requisição* (com frequência chamados de *verbos HTTP*) que um cliente usa para se comunicar com um servidor. Os métodos mais comuns são GET e POST.

Quando digitamos uma URL em um navegador (ou clicamos em um link), ele emite uma requisição HTTP GET para o servidor. As informações importantes passadas para o servidor são o path e a querystring da URL. A combinação de método, path e querystring é o que o aplicativo usa para determinar como responder.

No caso de um site, a maioria das páginas responderá a requisições GET. Geralmente as requisições POST são reservadas para enviar informações de volta para o servidor (o processamento de um formulário, por exemplo). É muito comum as requisições POST responderem com o mesmo HTML da requisição GET correspondente após o servidor ter processado qualquer informação incluída na requisição (como um formulário). Os navegadores usarão principalmente os métodos GET e POST quando se comunicarem com o servidor. No entanto, as requisições Ajax criadas pela aplicação podem usar qualquer verbo HTTP. Por exemplo, há um método HTTP chamado DELETE que é útil para uma chamada de API que exclua algo.

Com o Node e o Express, você será responsável pelos métodos aos quais responderá. No Express, geralmente criamos manipuladores para métodos específicos.

Cabeçalhos de requisição

A URL não é a única coisa que é passada para o servidor quando navegamos

para uma página. O navegador envia várias informações “invisíveis” sempre que visitamos um site. Não estou falando de informações pessoais (mas, se o navegador estiver infectado por um malware, isso pode ocorrer). O navegador informará ao servidor em que idioma ele prefere receber a página (por exemplo, se você baixar o Chrome em espanhol, ele solicitará a versão espanhola das páginas visitadas, se existir). Também enviará informações sobre o *agente do usuário* (o navegador, o sistema operacional e o hardware) e outros dados. Todas essas informações serão enviadas como cabeçalho da requisição, que é disponibilizado pela propriedade `headers` do objeto de requisição. Se estiver curioso para ver as informações que seu navegador está enviando, você pode criar um rota simples do Express para exibí-las (*ch06/00-echo-headers.js* no repositório fornecido):

```
app.get('/headers', (req, res) => {  
  res.type('text/plain')  
  const headers = Object.entries(req.headers)  
    .map(([key, value]) => `${key}: ${value}`)  
  res.send(headers.join("\n"))  
})
```

Cabeçalhos de resposta

Da mesma forma que seu navegador envia informações ocultas para o servidor na forma de cabeçalhos de requisição, quando o servidor responder, ele também retornará informações que não serão necessariamente renderizadas ou exibidas pelo navegador. As informações normalmente incluídas nos cabeçalhos de resposta são metadados e informações do servidor. Já vimos o cabeçalho `Content-Type`, que informa ao servidor que tipo de conteúdo está sendo transmitido (HTML, uma imagem, CSS, JavaScript etc.). É bom ressaltar que o navegador respeitará o cabeçalho `Content-Type` independentemente de qual for o path da URL. Logo, você poderia servir HTML a partir de um path */image.jpg* ou uma imagem a partir de um path */text.html*. (Não há razão para fazer isso; só é importante saber que os paths são abstratos e o navegador usa `Content-Type` para determinar como renderizar conteúdo). Além de `Content-Type`, os cabeçalhos podem indicar se a resposta está compactada e que tipo de codificação ela está usando. Os cabeçalhos de

resposta também podem conter dicas para o navegador sobre por quanto tempo ele pode armazenar o recurso em cache. Essa é uma consideração importante para a otimização do site e a discutiremos com detalhes no Capítulo 17.

Também é comum os cabeçalhos de resposta conterem algumas informações sobre o servidor, indicando qual o seu tipo e às vezes fornecendo detalhes até mesmo sobre o sistema operacional. A desvantagem de retornar informações do servidor é que isso dá aos hackers um ponto de partida para o comprometimento do site. Servidores extremamente protegidos contra problemas de segurança com frequência omitem essas informações ou fornecem informações falsas. É fácil desativar o cabeçalho X-Powered-By padrão do Express (*ch06/01-disable-x-powered-by.js* no repositório fornecido):

```
app.disable('x-powered-by')
```

Se você quiser ver os cabeçalhos de resposta, eles podem ser encontrados nas ferramentas de desenvolvedor do navegador. Por exemplo, para ver os cabeçalhos de resposta no Chrome:

1. Abra o console JavaScript.
2. Clique na aba Network.
3. Recarregue a página.
4. Selecione o HTML na lista de requisições (ele será o primeiro).
5. Clique na aba Headers; você verá todos os cabeçalhos de resposta.

Tipos de mídia de internet

O cabeçalho Content-Type é muito importante; sem ele, o cliente teria de adivinhar como renderizar o conteúdo. O formato do cabeçalho Content-Type é um *tipo de mídia de internet*, que é composto de um tipo, um subtipo e parâmetros opcionais. Por exemplo, `text/html; charset=UTF-8` especifica o tipo “texto”, o subtipo “html” e a codificação de caracteres UTF-8. A Internet Assigned Numbers Authority (Autoridade para Atribuição de Números de Internet) mantém uma *lista oficial de tipos de mídia de internet* (<https://www.iana.org/assignments/media-types/media-types.xhtml>). Com

frequência, você ouvirá os termos “tipo de conteúdo”, “tipo de mídia de internet” e “tipo MIME” sendo usados alternadamente. O MIME (Multipurpose Internet Mail Extensions) foi um precursor dos tipos de mídia de internet e, em grande parte, é equivalente.

Corpo da requisição

Além de cabeçalhos, uma requisição pode ter um *corpo* (da mesma forma que o corpo de uma resposta é o conteúdo real que está sendo retornado). Requisições GET comuns não têm corpos, mas geralmente as requisições POST têm. O tipo de mídia mais comum para o corpo de POST é `application/x-www-form-urlencoded`, que se trata simplesmente de pares nome/valor codificados e separados por ampersands (basicamente o mesmo formato de uma querystring). Se POST precisar dar suporte a uploads de arquivo, o tipo de mídia será `multipart/form-data`, que é um formato mais complicado. Por fim, as requisições Ajax podem usar `application/json` para o corpo. Aprenderemos mais sobre os corpos das requisições no Capítulo 8.

O objeto de requisição

Inicialmente, o *objeto de requisição* (que é passado como primeiro parâmetro de um manipulador de requisição, o que significa que você pode chamá-lo como quiser; é comum chamá-lo de `req` ou `request`) é uma instância de `http.IncomingMessage`, um objeto básico do Node. O Express adiciona outras funcionalidades. Examinaremos as propriedades e os métodos mais úteis do objeto de requisição (todos esses métodos são adicionados pelo Express, exceto `req.headers` e `req.url`, que são originários do Node):

`req.params`

Array contendo os *parâmetros de rota nomeados*. Aprenderemos mais sobre isso no Capítulo 14.

`req.query`

Objeto contendo parâmetros de querystring (também chamados de parâmetros de GET) como pares nome/valor.

`req.body`

Objeto contendo parâmetros de POST. Tem esse nome porque os parâmetros de POST são passados no corpo da requisição, e não na URL, como o são os parâmetros de querystring. Para tornar `req.body` disponível, você precisará de um middleware que consiga fazer o parsing do tipo de conteúdo do corpo, o que aprenderemos no Capítulo 10.

`req.route`

Informações sobre a rota procurada atualmente. É útil principalmente para a depuração de rotas.

`req.cookies/req.signedCookies`

Objetos contendo valores de cookies passados pelo cliente. Consulte o Capítulo 9.

`req.headers`

Os cabeçalhos de requisição recebidos do cliente. É um objeto cujas chaves e valores são os nomes e valores dos cabeçalhos, respectivamente. É bom ressaltar que ele vem do objeto `http.IncomingMessage` subjacente, logo, você não o encontrará listado na documentação do Express.

`req.accepts(types)`

Método de conveniência que determina se o cliente aceitará um tipo (ou tipos) específico (tipos opcionais poderiam ser um tipo MIME individual, como `application/json`, uma lista delimitada por vírgulas ou um array). Esse método é interessante principalmente para quem estiver criando APIs públicas; pressupõe-se que por padrão os navegadores sempre aceitarão HTML.

`req.ip`

Endereço IP do cliente.

`req.path`

Path da requisição (sem protocolo, host, porta ou querystring).

`req.hostname`

Método de conveniência que retorna o nome de host relatado pelo cliente. Essa informação pode ser falsa e não deve ser usada para fins de segurança.

`req.xhr`

Propriedade de conveniência que retorna `true` se a requisição tiver sido originada por uma chamada Ajax.

`req.protocol`

Protocolo usado na criação dessa requisição (no nosso caso, será `http` ou `https`).

`req.secure`

Propriedade de conveniência que retorna `true` se a conexão for segura. É equivalente a `req.protocol === 'https'`.

`req.url/req.originalUrl`

Com nomes inapropriados, essas propriedades retornam o `path` e a `querystring` (elas não incluem o protocolo, o `host` ou a porta). `req.url` pode ser reescrita para fins de roteamento interno, mas `req.originalUrl` foi projetada para permanecer com a requisição e a `querystring` originais.

Objeto de resposta

Inicialmente, o *objeto de resposta* (que é passado como segundo parâmetro de um manipulador de requisição, o que significa que você pode chamá-lo como quiser; é comum chamá-lo de `res`, `resp` ou `response`) é uma instância de `http.ServerResponse`, um objeto básico do Node. O Expressa adiciona outras funcionalidades. Examinaremos as propriedades e os métodos mais úteis do objeto de resposta (eles são todos adicionados pelo Express):

`res.status(code)`

Define o código de status HTTP. O Express usa como padrão 200 (OK), logo, você terá de empregar esse método para retornar um status 404 (Not Found), 500 (Server Error) ou qualquer outro código de status que queira

usar. Para redirecionamentos (códigos de status 301, 302, 303 e 307), há um método `redirect` que é preferível. É bom ressaltar que `res.status` retorna o objeto de resposta, o que significa que você pode encadear chamadas: `res.status(404).send('Not found')`.

`res.set(name, value)`

Define um cabeçalho de resposta. Isso não é algo que costume ser feito manualmente. Você também pode definir vários cabeçalhos ao mesmo tempo passando um único objeto como argumento cujas chaves e valores sejam os nomes e valores dos cabeçalhos, respectivamente.

`res.cookie(name, value, [options])`, `res.clearCookie(name, [options])`

Define ou remove os cookies que serão armazenados no cliente. Isso requer algum suporte de middleware; consulte o Capítulo 9.

`res.redirect([status], url)`

Redireciona o navegador. O código de redirecionamento padrão é 302 (Found). Você deve diminuir o redirecionamento a menos que esteja permanentemente movendo uma página, caso em que deve usar o código 301 (Moved Permanently).

`res.send(body)`

Envia uma resposta para o cliente. O Express usa como padrão o tipo de conteúdo `text/html`, logo, se você quiser alterá-lo para `text/plain` (por exemplo), terá de chamar `res.type('text/plain')` antes de chamar `res.send`. Se `body` for um objeto ou um array, a resposta será enviada como JSON (com o tipo de conteúdo sendo definido apropriadamente), no entanto, se você quiser enviar JSON, recomendo fazê-lo explicitamente chamando `res.json`.

`res.json(json)`

Envia JSON para o cliente.

`res.jsonp(json)`

Envia JSONP para o cliente.

`res.end()`

Encerra a conexão sem enviar uma resposta. Para aprender mais sobre as diferenças entre `res.send`, `res.json` e `res.end`, consulte *esse artigo* (<https://blog.fullstacktraining.com/res-json-vs-res-send-vs-res-end-in-express/>) de Tamas Piros.

`res.type(type)`

Método de conveniência que define o cabeçalho `Content-Type`. É equivalente a `res.set('Content-Type ', type)`, exceto por também tentar mapear extensões de arquivo para um tipo de mídia de internet se você fornecer uma string sem barra. Por exemplo, `res.type('txt')` resultará em um `Content-Type` igual a `text/plain`. Há áreas em que essa funcionalidade pode ser útil (como para servir automaticamente diferentes arquivos multimídia), mas, em geral, devemos evitá-la e preferir definir o tipo de mídia de internet correto explicitamente.

`res.format(object)`

Esse método permite enviar diferentes conteúdos dependendo do cabeçalho de requisição `Accept`. Ele é usado principalmente em APIs e o discutiremos com mais detalhes no Capítulo 15. Veja um exemplo simples: `res.format({'text/plain': 'hi there', 'text/html': 'hi there'})`.

`res.attachment([filename]), res.download(path, [filename], [callback])`

Esses dois métodos configuram um cabeçalho de resposta chamado `Content-Disposition` com `attachment`; isso solicitará ao navegador que baixe o conteúdo em vez de exibi-lo. Você pode especificar `filename` como uma dica para o navegador. Com `res.download`, podemos especificar o arquivo a ser baixado, enquanto `res.attachment` apenas define o cabeçalho; você ainda terá de enviar o conteúdo para o cliente.

`res.sendFile(path, [options], [callback])`

Esse método lê um arquivo especificado por `path` e envia seu conteúdo para o cliente. Ele é de pouca utilidade; é mais fácil usar o middleware `static` e inserir os arquivos que você deseja disponibilizar para o cliente no diretório *public*. No entanto, se você quiser que um recurso diferente seja servido a

partir da mesma URL dependendo de alguma condição, ele pode ser útil.

`res.links(links)`

Define o cabeçalho de resposta Links. Trata-se de um cabeçalho especializado que tem pouco uso na maioria das aplicações.

`res.locals, res.render(view, [locals], callback)`

`res.locals` é um objeto que armazena o contexto *padrão* para a renderização de views. `res.render` renderiza uma view usando o engine de templating configurado (o parâmetro `locals` de `res.render` não deve ser confundido com `res.locals`: ele sobrepõe o contexto de `res.locals`, mas um contexto que não for sobreposto continuará disponível). É bom ressaltar que `res.render` usa como padrão o código de resposta 200; utilize `res.status` para especificar um código de resposta diferente. A renderização de views será abordada com detalhes no Capítulo 7.

Obtendo mais informações

Devido à herança baseada em protótipo do JavaScript, saber com que exatamente estamos lidando pode ser um desafio. O Node fornece objetos que o Express estende, e os pacotes que você adicionar também poderão estendê-los. Descobrir exatamente o que está disponível pode ser desafiador. Em geral, recomendo procurar o que já existe: se você estiver procurando alguma funcionalidade, primeiro verifique a *documentação da API Express* (<http://expressjs.com/api.html>). A API Express é bastante completa e há chances de que você encontre nela o que está procurando.

Se você precisar de informações que não estejam documentadas, talvez tenha de examinar o código-fonte do Express (<https://github.com/expressjs/express>). Recomendo que o faça! Verá que é menos complicado do que parece. Aqui está um rápido roteiro de onde você pode encontrar o que deseja no código-fonte do Express:

lib/application.js

Interface principal do Express. Se você deseja entender como o middleware

é conectado ou como as views são renderizadas, esse é o local que deve acessar.

lib/express.js

Arquivo relativamente curto que fornece principalmente a função `createApplication` (a exportação padrão desse arquivo), que cria a instância de uma aplicação Express.

lib/request.js

Estende o objeto `http.IncomingMessage` do Node para fornecer um objeto de requisição robusto. Se quiser obter informações sobre todas as propriedades e métodos do objeto de requisição, esse é o local certo.

lib/response.js

Estende o objeto `http.ServerResponse` do Node para fornecer o objeto de resposta. Se quiser obter informações sobre as propriedades e métodos do objeto de resposta, verifique aqui.

lib/router/route.js

Dá suporte ao roteamento básico. Embora o roteamento seja essencial para o aplicativo, esse arquivo tem menos de 230 linhas; você verá que ele é bem simples e elegante.

Quando estiver examinando o código-fonte do Express, se quiser consulte a *documentação do Node* (<https://nodejs.org/en/docs/>), principalmente a seção sobre o módulo HTTP.

Fazendo um resumo

Este capítulo forneceu uma visão geral dos objetos de requisição e resposta, que são as partes mais importantes de uma aplicação Express. No entanto, quase sempre você só vai usar um subconjunto dessas funcionalidades. Logo, dividiremos esse tópico pelas funcionalidades que você usará com mais frequência.

Renderizando conteúdo

Quando renderizamos conteúdo, geralmente usamos `res.render`, que renderiza views dentro de layouts, agregando o máximo de valor. Ocasionalmente, você pode querer criar uma página de teste rápida; se for esse o caso, basta usar `res.send`. Podemos usar `req.query` para obter valores de `querystrings`, `req.session` para obter valores de sessões ou `req.cookie/req.signedCookies` para obter cookies. Os Exemplos 6.1 a 6.8 demonstram tarefas comuns de renderização de conteúdo.

Exemplo 6.1 – Uso básico (ch06/02-basic-rendering.js)

```
// uso básico
app.get('/about', (req, res) => {
  res.render('about')
})
```

Exemplo 6.2 – Códigos de resposta diferentes de 200 (ch06/03-different-response-codes.js)

```
app.get('/error', (req, res) => {
  res.status(500)
  res.render('error')
})
```

// ou em uma única linha...

```
app.get('/error', (req, res) => res.status(500).render('error'))
```

Exemplo 6.3 – Passando um contexto para uma view, incluindo valores de `querystring`, `cookie` e sessão (ch06/04-view-with-content.js)

```
app.get('/greeting', (req, res) => {
  res.render('greeting', {
    message: 'Hello esteemed programmer!',
    style: req.query.style,
    userid: req.cookies.userid,
    username: req.session.username
  })
})
```

Exemplo 6.4 – Renderizando uma view sem um layout (ch06/05-view-

without-layout.js)

```
// o layout a seguir não tem um arquivo, logo,  
// views/no-layout.handlebars deve incluir todo o HTML necessário  
app.get('/no-layout', (req, res) =>  
  res.render('no-layout', { layout: null })  
)
```

Exemplo 6.5 – Renderizando uma view com um layout personalizado (ch06/06-custom-layout.js)

```
// o arquivo de layout views/layouts/custom.handlebars será usado  
app.get('/custom-layout', (req, res) =>  
  res.render('custom-layout', { layout: 'custom' })  
)
```

Exemplo 6.6 – Renderizando saída em texto puro (ch06/07-plaintext-output.js)

```
app.get('/text', (req, res) => {  
  res.type('text/plain')  
  res.send('this is a test')  
})
```

Exemplo 6.7 – Adicionando um manipulador de erro (ch06/08-error-handler.js)

```
// esse código deve aparecer APÓS todas as suas rotas mesmo  
// se você não precisar de "next", ele deve ser incluído para  
// o Express reconhecer que esse é um manipulador de erro  
app.use((err, req, res, next) => {  
  console.error('** SERVER ERROR: ' + err.message)  
  res.status(500).render('08-error',  
    { message: "you shouldn't have clicked that!" })  
})
```

Exemplo 6.8 – Adicionando um manipulador de erro 404 (ch06/09-custom-404.js)

```
// esse código deve aparecer APÓS todas as suas rotas  
app.use((req, res) =>  
  res.status(404).render('404')  
)
```

Processando formulários

Quando processamos formulários, em geral as informações ficam em `req.body` (ou ocasionalmente em `req.query`). Você pode usar `req.xhr` para determinar se se trata de uma requisição Ajax ou se ela foi originada no navegador (isso será abordado com detalhes no Capítulo 8). Consulte o Exemplo 6.9 e o Exemplo 6.10. Nos exemplos a seguir, é preciso conectar o middleware body parser:

```
const bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: false })))
```

Aprenderemos mais sobre o middleware body parser no Capítulo 8.

Exemplo 6.9 – Processamento de formulário básico (ch06/10-basic-form-processing.js)

```
app.post('/process-contact', (req, res) => {
  console.log(`received contact from ${req.body.name} <${req.body.email}>`)
  res.redirect(303, '10-thank-you')
})
```

Exemplo 6.10 – Processamento de formulário mais robusto (ch06/11-more-robust-form-processing.js)

```
app.post('/process-contact', (req, res) => {
  try {
    // é aqui que tentaríamos salvar o contato no banco de dados ou em outro
    // mecanismo de persistência...por enquanto, apenas simularemos um erro
    if(req.body.simulateError) throw new Error("error saving contact!")
    console.log(`contact from ${req.body.name} <${req.body.email}>`)
    res.format({
      'text/html': () => res.redirect(303, '/thank-you'),
      'application/json': () => res.json({ success: true }),
    })
  } catch(err) {
    // aqui manipularíamos qualquer falha na persistência
    console.error(`error processing contact from ${req.body.name} ` +
      `<${req.body.email}>`)
    res.format({
      'text/html': () => res.redirect(303, '/contact-error'),
      'application/json': () => res.status(500).json({
        error: 'error saving contact information' }),
    })
  }
})
```

```
    })  
  }  
})
```

Fornecendo uma API

Quando fornecemos uma API, de maneira semelhante ao que ocorre no processamento de formulários, geralmente os parâmetros ficam em `req.query`, no entanto, você também pode usar `req.body`. A diferença com relação às APIs é que retornamos JSON, XML ou até mesmo texto puro, em vez de HTML, e com frequência usamos métodos HTTP menos comuns como PUT, POST e DELETE. O fornecimento de uma API será abordado no Capítulo 15. O Exemplo 6.11 e o Exemplo 6.12 usam o array de “produtos” a seguir (que normalmente seria recuperado em um banco de dados):

```
const tours = [  
  { id: 0, name: 'Hood River', price: 99.99 },  
  { id: 1, name: 'Oregon Coast', price: 149.95 },  
]
```



O termo *endpoint* costuma ser usado para descrever uma função individual de uma API.

Exemplo 6.11 – Endpoint GET simples retornando apenas JSON
(ch06/12-api.get.js)

```
app.get('/api/tours', (req, res) => res.json(tours))
```

O Exemplo 6.12 usa o método `res.format` no Express para responder de acordo com as preferências do cliente.

Exemplo 6.12 – Endpoint GET que retorna JSON, XML ou texto
(ch06/13-api-json-xml-text.js)

```
app.get('/api/tours', (req, res) => {  
  const toursXml = '<?xml version="1.0"?><tours>' +  
    tours.map(p =>  
      `<tour price="${p.price}" id="${p.id}">${p.name}</tour>`  
    ).join("") + '</tours>'  
  const toursText = tours.map(p =>  
    `${p.id}: ${p.name} (${p.price})`  
  )  
  res.format({  
    json: toursText,  
    xml: toursXml,  
    text: toursText  
  })  
})
```

```

    ).join('\n')
  res.format({
    'application/json': () => res.json(tours),
    'application/xml': () => res.type('application/xml').send(toursXml),
    'text/xml': () => res.type('text/xml').send(toursXml),
    'text/plain': () => res.type('text/plain').send(toursXml),
  })
})

```

No Exemplo 6.13, o endpoint PUT atualiza um produto e retorna JSON. Os parâmetros são passados no corpo da requisição (o `:id` na string da rota solicita ao Express que adicione uma propriedade `id` a `req.params`).

Exemplo 6.13 – Endpoint PUT para atualização (ch06/14-api-put.js)

```

app.put('/api/tour/:id', (req, res) => {
  const p = tours.find(p => p.id === parseInt(req.params.id))
  if(!p) return res.status(404).json({ error: 'No such tour exists' })
  if(req.body.name) p.name = req.body.name
  if(req.body.price) p.price = req.body.price
  res.json({ success: true })
})

```

Para concluir, o Exemplo 6.14 mostra um endpoint DELETE.

Exemplo 6.14 – Endpoint DELETE para exclusão (ch06/15-api-del.js)

```

app.delete('/api/tour/:id', (req, res) => {
  const idx = tours.findIndex(tour => tour.id === parseInt(req.params.id))
  if(idx < 0) return res.json({ error: 'No such tour exists.' })
  tours.splice(idx, 1)
  res.json({ success: true })
})

```

Conclusão

Espero que os pequenos exemplos deste capítulo tenham lhe dado uma ideia do tipo de funcionalidade que é comum em uma aplicação Express. Esses exemplos foram pensados para ser uma referência rápida que você possa revisar no futuro.

No próximo capítulo, nos aprofundaremos no templating, visto

superficialmente nos exemplos de renderização deste capítulo.

¹ As portas 0–1023 são “portas conhecidas” reservadas para *serviços comuns* (<http://bit.ly/33InJu7>).

Templating com o Handlebars

Neste capítulo, abordaremos o *templating*, que é uma técnica para a construção e a formatação do conteúdo a fim de exibi-lo para o usuário. Você pode considerar o templating como uma evolução da carta modelo: “Caro [Nome]: sentimos informar que ninguém mais usa [Tecnologia Ultrapassada], mas o templating está tendo plena adoção!”. Para enviar essa carta para várias pessoas, basta substituir [Nome] e [Tecnologia Ultrapassada].



Esse processo de substituir campos às vezes é chamado de *interpolação*, que é apenas uma palavra extravagante para dizermos “fornecer informações ausentes” nesse contexto.

Embora o templating no lado do servidor esteja sendo rapidamente substituído pelos frameworks front-end como o React, Angular e Vue, ele ainda tem suas aplicações, como a criação de email em HTML. Além disso, tanto o Angular quanto o Vue usam uma abordagem semelhante à dos templates para criar HTML, logo, o que você aprender sobre o templating no lado do servidor poderá ser aplicado a esses frameworks front-end.

Se você vem da experiência com PHP, deve estar se perguntando qual é o motivo de tanto barulho: o PHP é uma das primeiras linguagens que pode realmente ser chamada de linguagem de templating. Quase todas as principais linguagens que foram adaptadas para a web incluíram algum tipo de suporte ao templating. O que é diferente hoje é que o *engine de templating* está sendo desacoplado da linguagem.

Então, qual é a aparência do templating? Começaremos com o que o templating está substituindo considerando a maneira mais óbvia e simples de gerar uma linguagem a partir de outra (especificamente, geraremos HTML com JavaScript):

```
document.write('<h1>Please Don\'t Do This</h1>')
document.write('<p><span class="code">document.write</span> is naughty,\n')
document.write('and should be avoided at all costs.</p>')
document.write('<p>Today\'s date is ' + new Date() + '.</p>')
```

Talvez a única razão para isso parecer “óbvio” seja porque é dessa maneira que a programação sempre foi ensinada:

```
10 PRINT "Hello world!"
```

Em linguagens imperativas, costumamos dizer: “Faça isso, faça aquilo e depois faça mais aquilo”. Para algumas situações, essa abordagem funciona bem. Se você tiver 500 linhas de JavaScript para executar um cálculo complicado que resulte em um único número, e cada etapa dependa da etapa anterior, não haverá problema. No entanto, e se for o contrário? Você tem 500 linhas de HTML e 3 linhas de JavaScript. Faz sentido escrever `document.write` 500 vezes? Definitivamente não.

Na verdade, a questão é simplesmente a seguinte: mudar de contexto é problemático. Se você estiver escrevendo muitas linhas em JavaScript, será inconveniente e confuso incluir HTML. O oposto não é tão ruim. Estamos bastante acostumados a escrever JavaScript em blocos `<script>`, mas espero que você perceba a diferença: ainda há mudança de contexto, e ou você está escrevendo HTML ou está em um bloco `<script>` escrevendo JavaScript. Fazer JavaScript emitir HTML é problemático:

- Temos de nos preocupar constantemente com que caracteres precisam ser escapados e como fazer isso.
- Usar JavaScript para gerar HTML que também inclua JavaScript leva rapidamente qualquer pessoa à loucura.
- Geralmente perdemos o interessante realce da sintaxe e outros recursos úteis específicos da linguagem que o editor tenha.
- Pode ser muito mais complicado detectar HTML malformatado.
- É difícil analisar o código visualmente.
- Torna mais difícil para outras pessoas entenderem o código.

O templating resolve o problema nos permitindo escrever na linguagem de destino, fornecendo ao mesmo tempo a possibilidade de inserção de dados dinâmicos. Considere o exemplo anterior reescrito como um template do

Mustache:

```
<h1>Much Better</h1>
<p>No <span class="code">document.write</span> here!</p>
<p>Today's date is {{today}}.</p>
```

Agora tudo que temos de fazer é fornecer um valor para `{{today}}`, e essa é a essência das linguagens de templating.

Não há regras absolutas exceto essa

Não estou sugerindo que você *nunca* deve escrever HTML em JavaScript, apenas que deve evitar fazer isso sempre que possível. Especificamente, seria um pouco mais aceitável fazê-lo em código de front-end, principalmente se você estiver usando um framework front-end robusto. Por exemplo, eu não reclamaria muito deste código:

```
document.querySelector('#error').innerHTML =
  'Something <b>very bad</b> happened!'
```

Porém, digamos que o alterássemos para ficar assim:

```
document.querySelector('#error').innerHTML =
  '<div class="error"><h3>Error</h3>' +
  '<p>Something <b><a href="/error-detail/' + errorNumber +
  '">very bad</a></b>' +
  'happened. <a href="/try-again">Try again<a>, or ' +
  '<a href="/contact">contact support</a>.</p></div>'
```

Então, eu sugeriria que é hora de empregar um template. O que quero dizer é que sugiro que você pense bem ao definir a fronteira entre escrever HTML em strings e usar templates. Eu penderia para o lado dos templates, no entanto, e evitaria gerar HTML com JavaScript exceto para os casos mais simples.

Selecionando um engine de template

No universo do Node, temos muitos engines de templating para escolher, mas qual é o melhor? É uma pergunta complicada, e depende muito das suas necessidades. Aqui estão alguns critérios a considerar:

Desempenho

É claro que você deseja que seu engine de templating seja o mais rápido possível. Não vai querer que ele torne seu site lento.

Cliente, servidor ou ambos?

A maioria dos engines de templating, mas não todos, está disponível tanto no lado do servidor quanto no do cliente. Se você precisar usar templates nos dois lados (e precisará), recomendo selecionar algo que seja igualmente eficaz em ambos os casos.

Abstração

Você deseja algo familiar (como HTML comum com colchetes angulares, por exemplo) ou secretamente não gosta de HTML e adoraria usar algo que o poupasse de todos esses colchetes angulares? O templating (principalmente no lado do servidor) oferece mais opções nesse caso.

Esses são apenas alguns dos critérios mais relevantes na seleção de uma linguagem de templating. As opções de templating evoluíram muito, logo, você não deve errar independentemente do que selecionar.

O Express nos permite usar o engine de templating que quisermos, portanto, se você não gosta do Handlebars, verá que é fácil mudar para outro. Se quiser explorar suas opções, use o divertido e útil *Template-Engine-Chooser* (<http://bit.ly/2CExtK0>) (ele ainda é útil mesmo que não esteja mais sendo atualizado).

Examinaremos um engine de templating particularmente abstrato antes de passarmos para a nossa discussão do Handlebars.

Pug: uma abordagem diferente

Enquanto a maioria dos engines de templating usa uma abordagem baseada em HTML, o Pug se destaca ao nos distanciar dos detalhes dessa linguagem. Também vale a pena mencionar que o Pug foi criado por TJ Holowaychuk, a mesma pessoa que nos deu o Express. Logo, não é surpresa que a integração do Pug com o Express seja muito boa. A abordagem que o Pug usa é nobre:

sua essência está na admissão de que escrever na linguagem HTML é confuso e tedioso. Veremos como é a aparência de um template Pug, junto com o HTML que ele exibirá (extraído originalmente da *home page do Pug* (<https://pugjs.org>) porém com algumas alterações para ficar de acordo com o formato do livro:

doctype html	<!DOCTYPE html>
html(lang="en")	<html lang="en">
head	<head>
title= pageTitle	<title>Pug Demo</title>
script.	<script>
if (foo) {	if (foo) {
bar(1 + 5)	bar(1 + 5)
}	}
body	</script>
	<body>
h1 Pug	<h1>Pug</h1>
#container	<div id="container">
if youAreUsingPug	
p You are amazing	<p>You are amazing</p>
else	
p Get on it!	
p.	<p>
Pug is a terse and	Pug is a terse and
simple templating	simple templating
language with a	language with a
strong focus on	strong focus on
performance and	performance and
powerful features.	powerful features.
	</p>
	</body>
	</html>

O Pug com certeza representa muito menos digitação (esqueça os colchetes angulares ou as tags de fechamento). Em vez disso, ele usa indentação e algumas regras sensatas, tornando mais fácil dizer o que queremos. O Pug tem uma vantagem adicional: teoricamente, quando o HTML mudar, basta fazer o Pug usar sua versão mais recente, o que permite tornar o conteúdo “à prova de mudanças futuras”.

Mesmo admirando a filosofia do Pug e a elegância de sua execução, descobri que não quero me distanciar dos detalhes do HTML. Já que sou desenvolvedor web, o HTML está na essência de tudo que faço, e, se o preço for gastar as teclas de colchetes angulares de meu teclado, que assim seja. Muitos desenvolvedores front-end com quem converso também concordam, logo, talvez o mundo simplesmente ainda esteja preparado para o Pug.

Daqui em diante deixaremos o Pug de lado; você não o verá neste livro. No entanto, se a abstração lhe interessa, você certamente não terá problemas para usar o Pug com o Express, e há vários recursos para ajudá-lo a fazer isso.

Aspectos básicos do Handlebars

O *Handlebars* é uma extensão do Mustache, outro engine de templating popular. Recomendo o Handlebars por sua fácil integração com o JavaScript (tanto no front-end quanto no back-end) e a sintaxe familiar. Para mim, ele chega a um ponto de equilíbrio em todas as questões e é o que discutiremos neste livro. No entanto, os conceitos que estamos examinando são aplicáveis a outros engines de templating, logo, você estará preparado para testar outros engines se o Handlebars não lhe agradar.

A chave para entender o templating é compreender o conceito de *contexto*. Quando renderizamos um template, passamos para o engine um objeto chamado *objeto de contexto*, e é isso que permite que as substituições funcionem.

Por exemplo, se meu objeto de contexto for

```
{ name: 'Buttercup' }
```

e meu template for

```
<p>Hello, {{name}}!</p>
```

então, `{{name}}` será substituído por Buttercup. E se você quiser passar HTML para o template? Por exemplo, se nosso contexto fosse

```
{ name: '<b>Buttercup</b>' }
```

usar o template anterior resultaria em `<p>Hello, Buttercup</p>`, que provavelmente não é o que você está procurando. Para resolver esse problema, simplesmente use três chaves em vez de duas: `{{{name}}}`.



Embora já tenhamos estabelecido que devemos evitar escrever HTML em JavaScript, a possibilidade de desativar o escape HTML com três chaves tem alguns usos importantes. Por exemplo, se você estivesse construindo um sistema de gerenciamento de conteúdo (CMS, content management system) com editores *what you see is what you get*¹ (WYSIWYG), provavelmente iria querer passar HTML para suas views. Além disso, a possibilidade de renderizar propriedades do contexto sem o escape HTML é importante para *layouts* e *seções*, o que veremos em breve.

Na Figura 7.1, podemos ver como o engine Handlebars usa o contexto (representado por uma forma oval) combinado com o template para renderizar HTML.

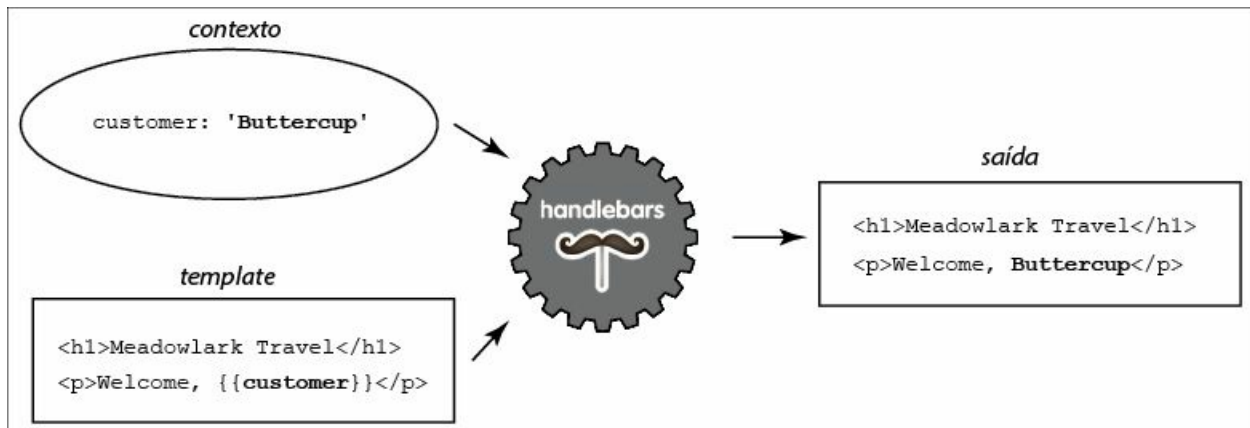


Figura 7.1 – Renderizando HTML com o Handlebars.

Comentários

Os *comentários* no Handlebars têm a aparência `{{! o comentário entra aqui }}`. É importante entender a diferença entre os comentários do Handlebars e os comentários HTML. Considere o template a seguir:

```
{{! comentário supersecreto}}  
<!-- comentário não tão secreto -->
```

Supondo que esse seja um template do lado do servidor, o comentário *supersecreto* nunca será enviado para o navegador, enquanto o comentário *não tão secreto* poderá ser visto se o usuário inspecionar o código-fonte HTML. Você deve dar preferência para os comentários do Handlebars para

algo que exponha detalhes da implementação, ou qualquer outra coisa que não queira que seja exposta.

Blocos

As coisas começam a ficar mais complicadas quando consideramos os *blocos*. Eles fornecem controle de fluxo, execução condicional e extensibilidade. Considere o objeto de contexto a seguir:

```
{
  currency: {
    name: 'United States dollars',
    abbrev: 'USD',
  },
  tours: [
    { name: 'Hood River', price: '$99.95' },
    { name: 'Oregon Coast', price: '$159.95' },
  ],
  specialsUrl: '/january-specials',
  currencies: [ 'USD', 'GBP', 'BTC' ],
}
```

Agora examinaremos um template para o qual podemos passar esse contexto:

```
<ul>
  {{#each tours}}
    {{! Estou em um novo bloco...e o contexto mudou }}
    <li>
      {{name}} - {{price}}
      {{#if ../currencies}}
        ({{../currency.abbrev}})
      {{/if}}
    </li>
  {{/each}}
</ul>
{{#unless currencies}}
  <p>All prices in {{currency.name}}.</p>
{{/unless}}
{{#if specialsUrl}}
  {{! Estou em um novo bloco...mas o contexto não mudou (mais ou menos) }}
  <p>Check out our <a href="{{specialsUrl}}">specials!</p>
```

```

{{else}}
  <p>Please check back often for specials.</p>
{{/if}}
<p>
  {{#each currencies}}
    <a href="#" class="currency">{{.}}</a>
  {{else}}
    Unfortunately, we currently only accept {{currency.name}}.
  {{/each}}
</p>

```

Há muita coisa ocorrendo nesse template, logo, vamos por partes. Ele começa com o auxiliar `each`, que permite iterar por um array. O que é preciso entender é que, entre `{{#each tours}}` e `{{/each tours}}`, o contexto muda. Na primeira passagem, ele muda para `{ name: 'Hood River', price: '$99.95' }`, e na segunda, o contexto é `{ name: 'Oregon Coast', price: '$159.95' }`. Logo, dentro desse bloco, podemos referenciar `{{name}}` e `{{price}}`. No entanto, se quisermos acessar o objeto `currency`, temos de usar `../` para acessar o contexto *pai*.

Se uma propriedade do contexto for ela própria um objeto, podemos acessar suas propriedades normalmente com um ponto, como em `{{currency.name}}`.

Tanto `if` quanto `each` têm um bloco `else` opcional (no caso de `each`, se não houver elementos no array, o bloco `else` será executado). Também usamos o auxiliar `unless`, que é basicamente o oposto do auxiliar `if`: ele só será executado se o argumento for falso.

A última coisa que devemos observar nesse template é o uso de `{{.}}` no bloco `{{#each currencies}}`. `{{.}}` referencia o contexto atual; nesse caso, o contexto atual é simplesmente uma string de um array que queremos exibir.



Acessar o contexto atual apenas com um ponto tem outra utilidade: pode diferenciar os auxiliares (que estudaremos em breve) das propriedades do contexto. Por exemplo, se você tiver um auxiliar `foo` e uma propriedade no contexto atual chamada `foo`, `{{foo}}` será o auxiliar e `{{./foo}}` a propriedade.

Templates do lado do servidor

Os *templates do lado do servidor* permitem renderizar o HTML *antes* que ele

seja enviado para o cliente. Ao contrário do templating no lado do cliente, em que os templates ficam disponíveis para o usuário curioso que souber como visualizar o código-fonte HTML, seus usuários nunca verão seu template do lado do servidor ou os objetos de contexto usados para gerar o HTML final.

Além de ocultar os detalhes da implementação, os templates do lado do servidor dão suporte ao *cache* de templates, que é importante para o desempenho. O engine de templating armazenará em cache os templates compilados (fazendo a recompilação e o rearmazenamento em cache somente quando o template mudar), o que melhorará o desempenho das views. Por padrão, o cache de views é desativado no modo de desenvolvimento e ativado no modo de produção. Se você quiser ativar explicitamente o cache de views, pode fazê-lo da seguinte forma:

```
app.set('view cache', true)
```

O Express dá suporte imediato ao Pug, EJS e JSHTML. Já discutimos o Pug e não vejo razão para recomendar o EJS ou o JSHTML (na minha opinião, nenhum dos dois vai muito longe sintaticamente). Logo, teremos de adicionar um pacote Node que dê suporte ao Handlebars no Express:

```
npm install express-handlebars
```

Em seguida, o conectaremos ao Express (*ch07/00/meadowlark.js* no repositório fornecido):

```
const expressHandlebars = require('express-handlebars')
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
}))
app.set('view engine', 'handlebars')
```



express-handlebars espera que os templates Handlebars tenham a extensão *.handlebars*. Acostumei-me a isso, mas, se você achar muito verboso, pode alterar a extensão para a também comum *.hbs* quando criar a instância de express-handlebars: `app.engine('handlebars', expressHandlebars({ extname: '.hbs' })))`.

Views e Layouts

Geralmente uma *view* representa uma página individual do site (embora possa

representar a parte de uma página carregada com Ajax, um email ou qualquer outra coisa). Por padrão, o Express procura views no subdiretório *views*. Um *layout* é um tipo especial de view – basicamente, é um template para templates. Os layouts são essenciais porque a maioria das páginas (quando não todas) de um site tem layout quase idêntico. Por exemplo, devem ter um elemento `<html>` e um elemento `<title>`, geralmente todos eles carregam os mesmos arquivos CSS e assim por diante. Você não vai querer duplicar esse código para cada página, e é aí que os layouts entram em cena. Examinaremos um arquivo de layout básico:

```
<!doctype html>
<html>
  <head>
    <title>Meadowlark Travel</title>
    <link rel="stylesheet" href="/css/main.css">
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

Observe o texto dentro da tag `<body>`: `{{{body}}}`. Isso é para o view engine saber onde deve renderizar o conteúdo da view. É importante usar três chaves em vez de duas: provavelmente a view conterá HTML e não queremos que o Handlebars tente escapá-lo. Repare que não há restrição para onde o campo `{{{body}}}` será inserido. Por exemplo, se você estivesse construindo um layout responsivo no Bootstrap, iria querer inserir sua view dentro de um contêiner `<div>`. Além disso, elementos de página comuns como cabeçalhos e rodapés geralmente residem no layout, e não na view. Aqui está um exemplo:

```
<!-- ... -->
<body>
  <div class="container">
    <header>
      <div class="container">
        <h1>Meadowlark Travel</h1>
        
      </div>
    </header>
```

```
<div class="container">
  {{{body}}}
</div>
<footer>&copy; 2019 Meadowlark Travel</footer>
</div>
</body>
```

Na Figura 7.2, vemos como o engine de template combina a view, o layout e o contexto. O elemento importante que esse diagrama deixa claro é a ordem das operações. A *view* é *renderizada primeiro*, antes do layout. Inicialmente, isso pode parecer contraintuitivo: a view está sendo renderizada *dentro* do layout, logo, o layout não deveria ser renderizado primeiro? Embora tecnicamente possa ser feito dessa forma, há vantagens em fazê-lo de maneira oposta. Particularmente, permite que a view personalize ainda mais o layout, o que será útil quando discutirmos as *seções* posteriormente neste capítulo.



Devido à ordem das operações, você pode passar uma propriedade chamada `body` para a view e ela será renderizada corretamente. No entanto, quando o layout for renderizado, o valor de `body` será sobreposto pela view renderizada.

Usando (ou não) layouts no Express

Provavelmente a maioria das páginas (se não todas) usará o mesmo layout, logo, não faz sentido continuar especificando o layout sempre que renderizarmos uma view. Observe que, quando criamos o view engine, especificamos o nome do layout padrão:

```
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
}))
```

Por padrão, o Express procura views no subdiretório `views` e os layouts em `views/layouts`. Portanto, se você tiver uma view `views/foo.handlebars`, pode renderizá-la desta forma:

```
app.get('/foo', (req, res) => res.render('foo'))
```

Ela usará `views/layouts/main.handlebars` como layout. Se você não quiser usar um layout (o que significa que precisará ter todo o boilerplate na view),

pode especificar layout: null no objeto de contexto:

```
app.get('/foo', (req, res) => res.render('foo', { layout: null })))
```

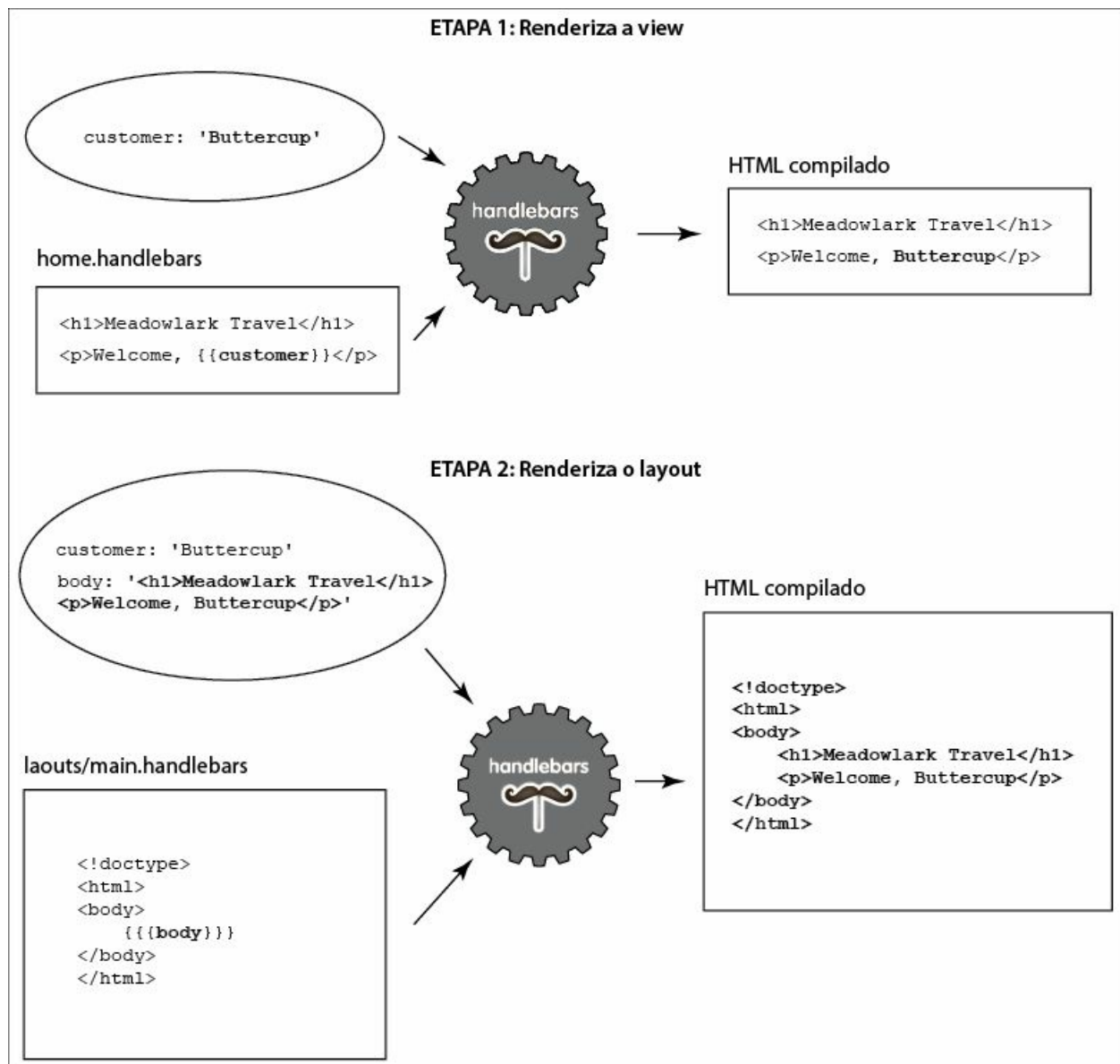


Figura 7.2 – Renderizando uma view com um layout.

Ou, se quisermos usar um template diferente, podemos especificar seu nome:

```
app.get('/foo', (req, res) => res.render('foo', { layout: 'microsite' })))
```

Esse código renderizará a view com o layout `views/layouts/microsite.handlebars`.

Lembre-se de que, quanto maior for o número de templates que você tiver, mais básico será o layout HTML que precisará de manutenção. Por outro

lado, se você tiver páginas que sejam significativamente diferentes no layout, a abordagem anterior pode valer a pena; é preciso encontrar um equilíbrio que funcione para seus projetos.

Seções

Uma técnica que estou pegando emprestada do excelente engine de template *Razor* da Microsoft é a ideia de *seções*. O layout funciona bem quando a view inteira cabe claramente dentro de um único elemento, mas o que acontece quando a view precisa se injetar em diferentes partes do layout? Um exemplo comum seria uma view que precisasse adicionar algo ao elemento `<head>` ou inserir um `<script>`, que costuma ser o último elemento do layout, por razões de desempenho.

Tanto o Handlebars quanto `express-handlebars` não têm uma maneira interna de fazer isso. Felizmente, os helpers do Handlebars facilitam muito essa tarefa. Quando instanciarmos o objeto Handlebars, adicionaremos um helper chamado `section` (`ch07/01/meadowlark.js` no repositório fornecido):

```
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
  helpers: {
    section: function(name, options) {
      if(!this._sections) this._sections = {}
      this._sections[name] = options.fn(this)
      return null
    },
  },
}))
```

Agora podemos usar o auxiliar `section` em uma view. Adicionaremos uma view (`views/section-test.handlebars`) para incluir algo em `<head>` além de inserir um script:

```
{{#section 'head'}}
  <!-- queremos que o Google ignore essa página -->
  <meta name="robots" content="noindex">
{{/section}}

<h1>Test Page</h1>
```

```
<p>We're testing some script stuff.</p>
```

```
{{#section 'scripts'}}  
<script>  
  document.querySelector('body')  
    .insertAdjacentHTML('beforeEnd', '<small>(scripting works!)</small>')  
</script>  
{{/section}}
```

Podemos então inserir as seções em nosso layout como inserimos {{{body}}}:

```
{{#section 'head'}}  
<!-- queremos que o Google ignore essa página -->  
<meta name="robots" content="noindex">  
{{/section}}
```

```
<h1>Test Page</h1>  
<p>We're testing some script stuff.</p>
```

```
{{#section 'scripts'}}  
<script>  
  const div = document.createElement('div')  
  div.appendChild(document.createTextNode('(scripting works!)'))  
  document.querySelector('body').appendChild(div)  
</script>  
{{/section}}
```

Partials

Com frequência você terá componentes que vai querer reutilizar em diferentes páginas (às vezes chamados de *widgets* no universo front-end). Uma maneira de fazer isso com templates é usar *partials* (que têm esse nome porque não renderizam uma view ou uma página inteira). Suponhamos que quiséssemos um componente de previsão do tempo que exibisse as condições meteorológicas atuais em Portland, Bend e Manzanita. Queremos que esse componente seja reutilizável para podermos inseri-lo facilmente em qualquer página, logo, usaremos um partial. Primeiro, crie um arquivo de partial, `views/partials/weather.handlebars`:

```
<div class="weatherWidget">
```




```

{{#each partials.weatherContext}}
  <div class="location">
    <h3>{{location.name}}</h3>
    <a href="{{location.forecastUrl}}">
      
      {{weather}}, {{temp}}
    </a>
  </div>
{{/each}}
<small>Source: <a href="https://www.weather.gov/documentation/services-web-api">
  National Weather Service</a></small>
</div>

```

Observe que definimos o namespace de nosso contexto começando com `partials.weatherContext`. Já que queremos usar o partial em qualquer página, não é prático passar o contexto para cada view, portanto, em vez disso usaremos `res.locals` (que fica disponível para cada view). No entanto, como não queremos afetar o contexto especificado por views individuais, inserimos todo o contexto do partial no objeto `partials`.

 `express-handlebars` nos permite passar templates de partial como parte do contexto. Por exemplo, se você adicionar `partials.foo = "Template!"` ao seu contexto, poderá renderizar esse partial com `{{> foo}}`. Isso sobreporá qualquer arquivo de view `.handlebars`, e é por isso que usamos `partials.weatherContext` anteriormente, em vez de `partials.weather`, que sobreporia `views/partials/weather.handlebars`.

No Capítulo 19, veremos como obter informações meteorológicas atualizadas a partir da API gratuita do National Weather Service. Por enquanto, usaremos apenas os dados fictícios retornados por uma função que chamaremos de `getWeatherData`.

Nesse exemplo, queremos que os dados meteorológicos estejam disponíveis para qualquer view, e o melhor mecanismo para isso é o middleware (que conheceremos melhor no Capítulo 10). Nosso middleware injetará os dados meteorológicos no objeto `res.locals.partials`, que os tornará disponíveis como o contexto de nosso partial.

Para tornar o middleware mais testável, vamos inseri-lo em seu próprio

arquivo, *lib/middleware/weather.js* (ch07/01/lib/middleware/weather.js no repositório fornecido):

```
const getWeatherData = () => Promise.resolve([
  {
    location: {
      name: 'Portland',
      coordinates: { lat: 45.5154586, lng: -122.6793461 },
    },
    forecastUrl: 'https://api.weather.gov/gridpoints/PQR/112,103/forecast',
    iconUrl: 'https://api.weather.gov/icons/land/day/tsra,40?size=medium',
    weather: 'Chance Showers And Thunderstorms',
    temp: '59 F',
  },
  {
    location: {
      name: 'Bend',
      coordinates: { lat: 44.0581728, lng: -121.3153096 },
    },
    forecastUrl: 'https://api.weather.gov/gridpoints/PDT/34,40/forecast',
    iconUrl: 'https://api.weather.gov/icons/land/day/tsra_sct,50?size=medium',
    weather: 'Scattered Showers And Thunderstorms',
    temp: '51 F',
  },
  {
    location: {
      name: 'Manzanita',
      coordinates: { lat: 45.7184398, lng: -123.9351354 },
    },
    forecastUrl: 'https://api.weather.gov/gridpoints/PQR/73,120/forecast',
    iconUrl: 'https://api.weather.gov/icons/land/day/tsra,90?size=medium',
    weather: 'Showers And Thunderstorms',
    temp: '55 F',
  },
])
```

```
const weatherMiddleware = async (req, res, next) => {
  if(!res.locals.partials) res.locals.partials = {}
  res.locals.partials.weatherContext = await getWeatherData()
```

```
next()
}
```

```
module.exports = weatherMiddleware
```

Com tudo definido, é só usar o `partial` em uma `view`. Por exemplo, para inserir nosso widget na home page, edite `views/home.handlebars`:

```
<h2>Home</h2>
{{> weather}}
```

A sintaxe `{{> partial_name}}` permite incluir um `partial` em uma `view`: `express-handlebars` saberá como procurar em `views/partials` uma `view` chamada `partial_name.handlebars` (ou `weather.handlebars`, em nosso exemplo).



`express-handlebars` dá suporte a subdiretórios, logo, se você tiver muitos `partials`, poderá organizá-los. Por exemplo, se você tiver alguns `partials` de mídia social, poderia inseri-los no diretório `views/partials/social` e incluí-los usando `{{> social/facebook}}`, `{{> social/twitter}}` etc.

Aperfeiçoando seus templates

Os templates são a parte principal do site. Uma boa estrutura de template economiza tempo de desenvolvimento, promove a consistência dentro do site e reduz o número de locais em que esquisitices dos layouts possam se ocultar. No entanto, para usufruir esses benefícios, você deve reservar algum tempo e elaborar seus templates cuidadosamente. Decidir quantos templates teremos é uma arte; geralmente, menos é melhor, mas há um limite em que os benefícios diminuem, dependendo da uniformidade das páginas. Seus templates também são sua primeira linha de defesa contra problemas de compatibilidade entre os diferentes navegadores e HTML válido. Eles devem ser criados com carinho e editados por alguém que seja versado em desenvolvimento front-end. Um ótimo ponto de partida – principalmente se você for iniciante – é o *HTML5 Boilerplate* (<http://html5boilerplate.com>). Nos exemplos anteriores, usamos um template HTML5 mínimo para respeitar o formato do livro, mas, para nosso projeto real, usaremos HTML5 Boilerplate.

Outro recurso popular que você pode usar para começar a manipular

templates são os temas de terceiros. Sites como *Themeforest* (<http://bit.ly/34Tdkfj>) e *WrapBootstrap* (<https://wrapbootstrap.com>) têm centenas de temas HTML5 prontos para uso que você pode empregar como ponto de partida para seu template. Para usar um tema de terceiros primeiro é preciso pegar o arquivo principal (geralmente *index.html*), renomeá-lo para *main.handlebars* (ou qualquer que seja o nome que você queira dar para seu arquivo de layout) e inserir os outros recursos (CSS, JavaScript, imagens) no diretório *public* usado para arquivos estáticos. Em seguida, você terá de editar o arquivo de template e descobrir onde deseja inserir a expressão `{{body}}`.

Dependendo dos elementos de seu template, pode ser melhor mover alguns deles para *partials*. Um ótimo exemplo é o do *herói* (um banner alto projetado para chamar a atenção do usuário). Se o herói aparecer em cada página (provavelmente uma escolha ruim), você o deixará no arquivo de template. Se aparecer apenas em uma página (geralmente a home page), ele só entrará nessa view. Se aparecer em várias páginas – mas não em todas –, considere sua inserção em um *partial*. A decisão é sua, e é a chance de demonstrar talento para a criação de um site exclusivo e cativante.

Conclusão

Vimos como o templating pode tornar seu código mais fácil de escrever, ler e editar. Graças aos templates, não precisamos ficar tediosamente compondo o HTML a partir de strings JavaScript; podemos escrever o HTML em nosso editor favorito e usar uma linguagem de templating compacta e fácil de ler para torná-lo dinâmico.

Agora que sabemos como formatar conteúdo para exibição, voltaremos nossa atenção para como trazer dados *para* nosso sistema com formulários HTML.

¹ N.T: A tradução é “o que você vê é o que você obtém”.

Manipulação de formulários

A maneira usual de coletar informações dos usuários é usar *formulários* HTML. Independentemente de se você permitir que o navegador envie o formulário normalmente, usar Ajax ou empregar vistosos controles de front-end, em geral o mecanismo subjacente continuará sendo um formulário HTML. Neste capítulo, discutiremos os diferentes métodos para manipulação e validação de formulários, e uploads de arquivos.

Enviando dados do cliente para o servidor

De um modo geral, as duas opções para envio de dados do cliente para o servidor são a querystring e o corpo da requisição. Normalmente, quando usamos a querystring, estamos criando uma requisição GET, e, quando usamos o corpo da requisição, estamos empregando uma requisição POST. (O protocolo HTTP não nos impede de fazer o contrário, mas não há motivo para isso: é melhor adotar a prática padrão.)

As pessoas costumam achar erroneamente que POST é segura e GET não: na verdade, as duas são seguras quando usamos o HTTPS, e nenhuma delas o é quando não o usamos. Se você não estiver usando o HTTPS, um invasor poderá examinar com a mesma facilidade os dados do corpo de POST e a querystring de GET. No entanto, se você estiver usando requisições GET, seus usuários verão todas as suas entradas (inclusive campos ocultos) na querystring, o que é deselegante e confuso. Além disso, com frequência os navegadores impõem limites para o tamanho da querystring (não há essa restrição para o tamanho do corpo). É por essas razões que geralmente recomendo usar POST para o envio de formulários.

Formulários HTML

Este livro está dando ênfase ao lado do servidor, mas é importante conhecer alguns aspectos básicos da construção de formulários HTML. Aqui está um exemplo simples:

```
<form action="/process" method="POST">
  <input type="hidden" name="hush" val="hidden, but not secret!">
  <div>
    <label for="fieldColor">Your favorite color: </label>
    <input type="text" id="fieldColor" name="color">
  </div>
  <div>
    <button type="submit">Submit</button>
  </div>
</form>
```

Observe que o método é especificado explicitamente como POST na tag <form>; se você não fizer isso, o padrão usado será GET. O atributo action especifica a URL que receberá o formulário quando ele for enviado. Se você omitir esse campo, o formulário será enviado para a mesma URL a partir da qual ele foi carregado. Recomendo que você sempre forneça um atributo action válido, mesmo se estiver usando Ajax (deve fazê-lo para evitar perder dados; consulte o Capítulo 22 para ver mais informações).

Da perspectiva do servidor, os atributos importantes dos campos <input> são os atributos name: é com eles que o servidor identifica o campo. É importante entender que o atributo name é diferente do atributo id, que só deve ser usado para a estilização e para funcionalidades de front-end (ele não é passado para o servidor).

Observe o campo oculto: ele não será renderizado no navegador do usuário. No entanto, você não deve usá-lo para informações secretas ou sigilosas; se o usuário examinar o código-fonte da página, o campo oculto será exposto.

O HTML não nos impede de incluir vários formulários na mesma página (essa era uma restrição infeliz de alguns frameworks de servidor antigos; estou me referindo ao ASP). Recomendo que você mantenha seus formulários logicamente consistentes; um formulário deve conter todos os campos que quisermos que sejam enviados ao mesmo tempo (campos

opcionais/vazios são permitidos) e nenhum outro. Se você tiver duas ações diferentes em uma página, use dois formulários diferentes. Um exemplo disso seria termos um formulário para uma busca no site e outro para a inscrição para o envio de newsletters por email. É possível usar um formulário grande e descobrir que ação deve ser executada de acordo com o botão no qual a pessoa clicou, mas é complicado e geralmente pouco amigável para pessoas com deficiência (devido à maneira como os navegadores com recursos de acessibilidade renderizam formulários).

Quando o usuário enviar o formulário desse exemplo, a URL `/process` será chamada e os valores dos campos serão transmitidos para o servidor no corpo da requisição.

Codificando

Quando o formulário for enviado (pelo navegador ou com Ajax), deve ser codificado de alguma forma. Se você não especificar explicitamente uma codificação, o padrão usado será `application/x-www-form-urlencoded` (trata-se apenas de um tipo de mídia extenso que significa “codificado em URL”). Essa é uma codificação básica e fácil de usar que tem suporte imediato no Express.

Se você precisar fazer o upload de arquivos, as coisas ficarão mais complicadas. Não há maneira fácil de enviar arquivos usando a codificação em URL, logo, você será forçado a usar o tipo de codificação `multipart/form-data`, que não é manipulada diretamente pelo Express.

Diferentes abordagens para a manipulação de formulários

Se você não está usando Ajax, sua única opção é enviar o formulário pelo navegador, o que recarregará a página. No entanto, fica a seu critério *como* a página será recarregada. Há duas coisas que devem ser consideradas no processamento de formulários: que path os manipulará (a ação) e que resposta será enviada para o navegador.

Quando o formulário usa `method="POST"` (o que é recomendado), é comum a

utilização do mesmo path para sua exibição e processamento: é possível distinguir essas duas ações porque a primeira é uma requisição GET e a outra é uma requisição POST. Se você adotar essa abordagem, poderá omitir o atributo `action` no formulário.

A outra opção é usar um path separado para processar o formulário. Por exemplo, se sua página de contatos usar o path `/contact`, você pode utilizar o path `/process-contact` para processar o formulário (pela especificação de `action="/process-contact"`). Se usar essa abordagem, terá a opção de enviar o formulário com GET (o que não recomendo; expõe desnecessariamente os campos do formulário na URL). Pode ser preferível empregar um endpoint separado para o envio do formulário se você tiver várias URLs que usem o mesmo mecanismo de envio (por exemplo, poderia haver uma caixa de assinatura para o recebimento de emails em várias páginas do site).

Seja qual for o path que você usar para processar o formulário, terá de decidir que resposta retornará para o navegador. Estas são suas opções:

Resposta HTML direta

Após processar o formulário, você pode retornar o HTML diretamente para o navegador (uma view, por exemplo). Essa abordagem exibirá um aviso se o usuário tentar recarregar a página e pode afetar o bookmarking e o botão Back, e, portanto, não é recomendada.

Redirecionamento 302

Embora essa seja uma abordagem comum, é um uso incorreto do significado original do código de resposta 302 (Found). O HTTP 1.1 adicionou o código de resposta 303 (See Other), que é preferível. A menos que você tenha alguma razão para abranger navegadores criados antes de 1996, deve usar 303.

Redirecionamento 303

O código de resposta 303 (See Other) foi adicionado no HTTP 1.1 para resolver a má utilização do redirecionamento 302. A especificação HTTP indica que o navegador deve usar uma requisição GET ao seguir um redirecionamento 303, independentemente do método original. Esse é o

método recomendado para a resposta a uma requisição de envio de formulário.

Já que a recomendação é que você responda a um envio de formulário com um redirecionamento 303, a próxima pergunta é: “Para onde o redirecionamento apontará?”. A resposta é você quem decide. Aqui estão as abordagens mais comuns:

Redirecionamento para páginas de sucesso/falha dedicadas

Esse método requer que dediquemos URLs às mensagens de sucesso ou falha apropriadas. Por exemplo, se o usuário se inscrevesse para receber emails promocionais, mas houvesse um erro de banco de dados, o redirecionaríamos para `/error/database`. Se o endereço de email de um usuário for inválido, podemos redirecioná-lo para `/error/invalid-email`, e, se tudo correr bem, o redirecionaríamos para `/promo-email/thank-you`. Uma das vantagens desse método é que ele permite análises: o número de visitas à página `/promo-email/thank-you` deve ser correspondente ao número de pessoas que estão se inscrevendo para receber emails promocionais. Ele também é fácil de implementar. No entanto, apresenta algumas desvantagens. Seu uso demanda alocar URLs para cada possibilidade, o que significa páginas a serem projetadas, preenchidas com conteúdo e editadas. Outra desvantagem é que a experiência do usuário talvez não seja ótima: os usuários gostam de receber agradecimentos, mas depois têm de navegar de volta para onde estavam ou para onde querem ir. Essa é a abordagem que usaremos por enquanto: mudaremos para o uso de mensagens flash (que não devem ser confundidas com o Adobe Flash) no Capítulo 9.

Redirecionamento para o local original com uma mensagem flash

Para formulários pequenos que estejam espalhados pelo site (como uma inscrição para o recebimento de emails, por exemplo), a melhor experiência de usuário é não interromper o fluxo de navegação. Isto é, forneça uma maneira de enviar um endereço de email sem abandonar a página. É claro que uma forma de fazer isso é com Ajax, mas, se você não quiser usá-lo (ou quiser que seu mecanismo de fallback forneça uma boa experiência de usuário), pode fazer o redirecionamento de volta para a página em que o

usuário estava. O modo mais fácil é usar um campo oculto no formulário que seja preenchido com a URL atual. Já que você quer que haja algum feedback de que o envio do usuário foi recebido, pode usar mensagens flash.

Redirecionamento para um novo local com uma mensagem flash

Geralmente formulários grandes têm sua própria página e não faz sentido permanecer nela uma vez que você os tiver enviado. Nessa situação, é preciso tentar adivinhar com cuidado para onde o usuário pode querer ir em seguida e fazer o redirecionamento apropriadamente. Por exemplo, se você estiver construindo uma interface administrativa, e tiver um formulário para a criação de um novo pacote de férias, é de se esperar que seu usuário queira ir para a página administrativa que lista todos os pacotes de férias após enviar o formulário. No entanto, você ainda deve empregar uma mensagem flash para dar feedback ao usuário sobre o resultado do envio.

Se você estiver usando Ajax, recomendo uma URL dedicada. É tentador começar os manipuladores de Ajax com um prefixo (por exemplo, */ajax/enter*), mas não acho essa uma boa abordagem: ela anexa detalhes da implementação à URL. Além disso, como veremos em breve, seu manipulador de Ajax deve tratar envios regulares do navegador como uma atividade livre de falhas (fail-safe).

Manipulação de formulários com o Express

Caso você esteja usando GET para a manipulação de formulários, os campos estarão disponíveis no objeto `req.query`. Por exemplo, se houver um campo de entrada HTML com o atributo de nome `email`, seu valor será passado para o manipulador como `req.query.email`. Não há muita coisa a ser dita sobre essa abordagem; ela é bastante simples.

Se estiver usando POST (o que recomendo), terá de conectar um middleware para fazer o parsing do corpo codificado em URL. Primeiro, instale o middleware `body-parser` (`npm install body-parser`); em seguida, conecte-o (*ch08/meadowlark.js* no repositório fornecido):

```
const bodyParser = require('body-parser')
```

```
app.use(bodyParser.urlencoded({ extended: true })))
```

Uma vez que você tiver conectado o body-parser, verá que agora o objeto req.body está disponível e é nele que todos os campos de seu formulário serão disponibilizados. É bom ressaltar que req.body não nos impede de usar a querystring. Daremos prosseguimento e adicionaremos um formulário ao site da Meadowlark Travel que permita que o usuário se inscreva em uma mailing list. A título de demonstração, usaremos a querystring, um campo oculto e campos visíveis em `/views/newsletter-signup.handlebars`:

```
<h2>Sign up for our newsletter to receive news and specials!</h2>
<form class="form-horizontal" role="form"
  action="/newsletter-signup/process?form=newsletter" method="POST">
  <input type="hidden" name="_csrf" value="{{csrf}}">
  <div class="form-group">
    <label for="fieldName" class="col-sm-2 control-label">Name</label>
    <div class="col-sm-4">
      <input type="text" class="form-control"
        id="fieldName" name="name">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
    <div class="col-sm-4">
      <input type="email" class="form-control" required
        id="fieldEmail" name="email">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-4">
      <button type="submit" class="btn btn-primary">Register</button>
    </div>
  </div>
</form>
```

Observe que estamos usando estilos do Bootstrap, como faremos no resto do livro. Se você não conhece o Bootstrap, consulte *sua documentação* (<http://getbootstrap.com>).

Já conectamos nosso body parser, logo, precisamos adicionar os

manipuladores da página de inscrição nas newsletters, a função de processamento e a página de agradecimento (*ch08/lib/handlers.js* no repositório fornecido):

```
exports.newsletterSignup = (req, res) => {
  // aprenderemos o que é CSRF posteriormente...por enquanto,
  // forneceremos apenas um valor fictício
  res.render('newsletter-signup', { csrf: 'CSRF token goes here' })
}
exports.newsletterSignupProcess = (req, res) => {
  console.log('Form (from querystring): ' + req.query.form)
  console.log('CSRF token (from hidden form field): ' + req.body._csrf)
  console.log('Name (from visible form field): ' + req.body.name)
  console.log('Email (from visible form field): ' + req.body.email)
  res.redirect(303, '/newsletter-signup/thank-you')
}
exports.newsletterSignupThankYou = (req, res) =>
  res.render('newsletter-signup-thank-you')
```

(Se ainda não o fez, crie um arquivo *views/newsletter-signup-thank-you.handlebars*).

Para concluir, conectaremos nossos manipuladores à aplicação (*ch08/meadowlark.js* no repositório fornecido):

```
app.get('/newsletter-signup', handlers.newsletterSignup)
app.post('/newsletter-signup/process', handlers.newsletterSignupProcess)
app.get('/newsletter-signup/thank-you', handlers.newsletterSignupThankYou)
```

Isso é tudo. Observe que, em nosso manipulador, estamos fazendo o redirecionamento para uma view “thank you”. Poderíamos renderizar uma view aqui, mas, se o fizéssemos, o campo de URL no navegador do visitante permaneceria sendo */process*, o que poderia ser confuso. Emitir um redirecionamento resolve esse problema.



É importante que você use um redirecionamento 303 (ou 302) em vez de 301 nesse caso. Os redirecionamentos 301 são “permanentes”, o que significa que seu navegador poderá armazenar em cache o destino do redirecionamento. Se você usar um redirecionamento 301 e tentar enviar o formulário novamente, seu navegador pode ignorar o manipulador de */process* e ir diretamente para */thank-you*, já que intuirá corretamente que o

redirecionamento é permanente. O redirecionamento 303, por outro lado, diz ao navegador “Sim, sua solicitação é válida e você pode encontrar a resposta aqui” e não armazena em cache o destino do redirecionamento.

Na maioria dos frameworks de front-end, é mais comum vermos dados de formulários sendo enviados em JSON com a API `fetch`, que examinaremos a seguir. Mesmo assim, é bom entender como os navegadores manipulam o envio de formulários por padrão, já que você ainda encontrará formulários implementados dessa forma.

Examinaremos, então, o envio de formulários com `fetch`.

Usando `fetch` para enviar dados de formulário

Usar a API `fetch` para enviar dados de formulário codificados em JSON é uma abordagem muito mais moderna que fornece um controle maior sobre a comunicação cliente/servidor e permite que haja menos atualizações de páginas.

Já que não estamos enviando requisições round-trip para o servidor, não temos mais de nos preocupar com redirecionamentos e múltiplas URLs de usuário (ainda teremos uma URL separada para o processamento do formulário), e, portanto, consolidaremos a “experiência inteira de inscrição na newsletter” sob uma única URL chamada `/newsletter`.

Começaremos com o código de front-end. O conteúdo do formulário HTML não precisa ser alterado (os campos e o layout são os mesmos), mas não é necessário especificar uma ação ou um método e encapsularemos nosso formulário em um contêiner `<div>` que tornará mais fácil a exibição da mensagem “thank you”:

```
<div id="newsletterSignupFormContainer">
  <form class="form-horizontal" role="form" id="newsletterSignupForm">
    <!-- o resto do conteúdo do formulário é o mesmo... -->
  </form>
</div>
```

Em seguida, teremos um script que interceptará o envio do formulário e o cancelará (usando `Event#preventDefault`), logo, nós mesmos poderemos manipular o processamento do formulário (`ch08/views/newsletter.handlebars`

no repositório fornecido):

```
<script>
document.getElementById('newsletterSignupForm')
  .addEventListener('submit', evt => {
    evt.preventDefault()
    const form = evt.target
    const body = JSON.stringify({
      _csrf: form.elements._csrf.value,
      name: form.elements.name.value,
      email: form.elements.email.value,
    })
    const headers = { 'Content-Type': 'application/json' }
    const container =
      document.getElementById('newsletterSignupFormContainer')
    fetch('/api/newsletter-signup', { method: 'post', body, headers })
      .then(resp => {
        if(resp.status < 200 || resp.status >= 300)
          throw new Error(`Request failed with status ${resp.status}`)
        return resp.json()
      })
      .then(json => {
        container.innerHTML = '<b>Thank you for signing up!</b>'
      })
      .catch(err => {
        container.innerHTML = `<b>We're sorry, we had a problem ` +
          `signing you up. Please <a href="/newsletter">try again</a>`
      })
  })
</script>
```

Agora, no arquivo do servidor (*meadowlark.js*), certifique-se de conectar um middleware que possa fazer o parsing de corpos JSON, antes de especificar nossos dois endpoints:

```
app.use(bodyParser.json())

//...

app.get('/newsletter', handlers.newsletter)
app.post('/api/newsletter-signup', handlers.api.newsletterSignup)
```

Observe que estamos inserindo nosso endpoint de processamento do formulário em uma URL que começa com `api`; essa é uma técnica comum que diferencia os endpoints do usuário (navegador) dos endpoints da API acessados com `fetch`.

Agora adicionaremos os endpoints ao arquivo `lib/handlers.js`:

```
exports.newsletter = (req, res) => {  
  // aprenderemos o que é CSRF posteriormente...por enquanto,  
  // forneceremos apenas um valor fictício  
  res.render('newsletter', { csrf: 'CSRF token goes here' })  
}  
exports.api = {  
  newsletterSignup: (req, res) => {  
    console.log('CSRF token (from hidden form field): ' + req.body._csrf)  
    console.log('Name (from visible form field): ' + req.body.name)  
    console.log('Email (from visible form field): ' + req.body.email)  
    res.send({ result: 'success' })  
  },  
}
```

Podemos executar qualquer processamento que precisarmos no manipulador de processamento do formulário; normalmente salvaríamos os dados no banco de dados. Se houver problemas, retornaremos um objeto JSON com a propriedade `err` (em vez de `result: success`).



Nesse exemplo, estamos assumindo que todas as requisições Ajax estão procurando JSON, mas não há o requisito de que o Ajax deva usar JSON para a comunicação (na verdade, Ajax costumava ser um acrônimo em que o “X” representava XML). Essa abordagem é muito apropriada para JavaScript, que é uma linguagem habilitada para manipular JSON. Se você estiver tornando seus endpoints Ajax disponíveis ou se souber que suas requisições Ajax podem usar algo diferente de JSON, deve retornar uma resposta apropriada baseada *exclusivamente* no cabeçalho `Accepts`, que podemos acessar com o método auxiliar `req.accepts`. Se estiver respondendo com base somente no cabeçalho `Accepts`, pode ser interessante examinar também `res.format`, que é um método de conveniência útil que facilita responder apropriadamente dependendo do que o cliente espera. Se o fizer, você terá de se certificar de definir a

propriedade `dataType` ou `accepts` ao enviar requisições Ajax com JavaScript.

Uploads de arquivos

Já mencionamos que os uploads de arquivos causam muitas complicações. Felizmente, há ótimos projetos que ajudam a facilitar a manipulação de arquivos.

Há quatro opções populares e robustas para o processamento de formulário multiparte: `busboy`, `multipart`, `formidable` e `multer`. Usei todas as quatro, e elas são boas, mas acho que o `multipart` tem melhor manutenção, então, vamos usá-lo aqui.

Criaremos um formulário de upload de arquivo para um concurso de fotos de férias da Meadowlark Travel (`views/contest/vacation-photo.handlebars`):

```
<h2>Vacation Photo Contest</h2>
```

```
<form class="form-horizontal" role="form"
  enctype="multipart/form-data" method="POST"
  action="/contest/vacation-photo/{{year}}/{{month}}">
  <input type="hidden" name="_csrf" value="{{csrf}}">
  <div class="form-group">
    <label for="fieldName" class="col-sm-2 control-label">Name</label>
    <div class="col-sm-4">
      <input type="text" class="form-control"
        id="fieldName" name="name">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
    <div class="col-sm-4">
      <input type="email" class="form-control" required
        id="fieldEmail" name="email">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldPhoto" class="col-sm-2 control-label">Vacation photo</label>
    <div class="col-sm-4">
```



```

    <input type="file" class="form-control" required accept="image/*"
      id="fieldPhoto" name="photo">
  </div>
</div>
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-4">
    <button type="submit" class="btn btn-primary">Register</button>
  </div>
</div>
</form>

```

Observe que é preciso especificar `enctype="multipart/form-data"` para permitir o upload de arquivos. Também estamos restringindo o tipo dos arquivos envolvidos no upload usando o atributo `accept` (que é opcional).

Agora precisamos criar manipuladores de rota, mas há um dilema. Queremos manter nossa habilidade de testar facilmente os manipuladores de rota, o que será complicado com o processamento de formulário multiparte (como quando usamos o middleware para processar outros tipos de codificação de corpo antes de chegar aos manipuladores). Já que não queremos testar a decodificação de formulários multiparte por conta própria (podemos assumir que ela seja feita totalmente pelo multiparty), manteremos nossos manipuladores “puros” passando para eles as informações já processadas. Como ainda não sabemos com que aparência isso ficará, começaremos com o fluxo (plumbing) do Express em *meadowlark.js*:

```

const multiparty = require('multiparty')

app.post('/contest/vacation-photo/:year/:month', (req, res) => {
  const form = new multiparty.Form()
  form.parse(req, (err, fields, files) => {
    if(err) return res.status(500).send({ error: err.message })
    handlers.vacationPhotoContestProcess(req, res, fields, files)
  })
})

```

Estamos usando o método `parse` do `multiparty` para fazer o parsing dos dados requisitados nos campos de dados e nos arquivos. Esse método armazenará os arquivos em um diretório temporário no servidor e as informações serão retornadas no array `files` passado de volta.

Agora temos informações adicionais para passar para nosso manipulador de rota (testável): os campos (que não estarão em req.body como nos exemplos anteriores já que estamos usando um body parser diferente) e informações sobre os arquivos que foram coletados. Como já sabemos que aparência isso terá, podemos criar o manipulador de rota:

```
exports.vacationPhotoContestProcess = (req, res, fields, files) => {  
  console.log('field data: ', fields)  
  console.log('files: ', files)  
  res.redirect(303, '/contest/vacation-photo-thank-you')  
}
```

(O ano e o mês estão sendo especificados como *parâmetros de rota*, algo que conheceremos no Capítulo 14). Vá em frente, execute o código e examine o log do console. Você verá os campos de seu formulário surgirem como esperado: como um objeto com propriedades correspondentes aos nomes dos campos. O objeto files contém mais dados, mas é relativamente simples. Para cada arquivo carregado, você verá que há propriedades para o tamanho, o path de destino do upload (geralmente um nome aleatório em um diretório temporário) e o nome original do arquivo (apenas o nome do arquivo, e não o path inteiro, por razões de segurança e privacidade).

O que você fará com esse arquivo é uma decisão sua: pode armazená-lo em um banco de dados, copiá-lo em um local mais permanente ou fazer seu upload em um sistema de armazenamento de arquivos baseado em nuvem. Lembre-se de que, se você estiver usando o armazenamento local para salvar arquivos, sua aplicação não será tão escalável, o que torna essa opção insatisfatória para a hospedagem baseada em nuvem. Revisitaremos esse exemplo no Capítulo 13.

Uploads de arquivo com fetch

Felizmente, usar fetch para uploads de arquivos é quase idêntico a permitir que o navegador os manipule. A parte difícil dos uploads na verdade está na codificação, que está sendo manipulada para nós com o middleware.

Considere este código JavaScript para o envio do conteúdo de nosso formulário com o uso de fetch:

```

<script>
document.getElementById('vacationPhotoContestForm')
  .addEventListener('submit', evt => {
    evt.preventDefault()
    const body = new FormData(evt.target)
    const container =
      document.getElementById('vacationPhotoContestFormContainer')
    const url = '/api/vacation-photo-contest/{ {year} }/{ {month} }'
    fetch(url, { method: 'post', body })
      .then(resp => {
        if(resp.status < 200 || resp.status >= 300)
          throw new Error(`Request failed with status ${resp.status}`)
        return resp.json()
      })
      .then(json => {
        container.innerHTML = '<b>Thank you for submitting your photo!</b>'
      })
      .catch(err => {
        container.innerHTML = `<b>We're sorry, we had a problem processing ` +
          `your submission. Please <a href="/newsletter">try again</a>`
      })
  })
</script>

```

O detalhe importante a ser observado aqui é que convertemos o elemento de formulário em um objeto *FormData* (<https://mzl.la/2CErVzb>), que *fetch* pode aceitar diretamente como corpo da requisição. Isso é tudo que é preciso saber! Já que a codificação é exatamente a mesma de quando deixamos que o navegador fizesse a manipulação, nosso manipulador é quase igual. Queremos apenas retornar uma resposta JSON em vez de um redirecionamento:

```

exports.api.vacationPhotoContest = (req, res, fields, files) => {
  console.log('field data: ', fields)
  console.log('files: ', files)
  res.send({ result: 'success' })
}

```

Melhorando a UI de upload de arquivo

O controle interno `<input>` do navegador para uploads de arquivo é, digamos, um pouco deficiente se visto de uma perspectiva das UIs. Provavelmente você já deve ter visto interfaces de arrastar e soltar e botões de upload de arquivos estilizados de maneira mais interessante.

A boa nova é que as técnicas que você aprendeu aqui são aplicáveis a quase todos os populares e “vistosos” componentes de upload de arquivos. No fim das contas, a maioria deles dá uma aparência bonita ao mesmo mecanismo de upload por formulário.

Alguns dos front-ends de upload de arquivos mais populares são:

- *jQuery File Upload* (<http://bit.ly/2Qbcd6I>)
- *Uppy* (<http://bit.ly/2rEFWeb>) (tem a vantagem de dar suporte a muitos destinos de upload populares)
- *file-upload-with-preview* (<http://bit.ly/2X5fS7F>) (esse nos dá controle total; temos acesso a um array de objetos de arquivo que podemos empregar para construir um objeto `FormData` para usar com `fetch`)

Conclusão

Neste capítulo, você aprendeu diversas técnicas para usar no processamento de formulários. Examinamos a maneira tradicional na qual os formulários são manipulados por navegadores (permitindo que o navegador emita uma requisição `POST` para o servidor com o conteúdo do formulário e renderizando a resposta a partir do servidor, geralmente um redirecionamento) assim como a cada vez mais onipresente abordagem de impedir que o navegador envie o formulário e manipularmos isso nós mesmos com `fetch`.

Conhecemos as maneiras comuns de os formulários serem codificados:

`application/x-www-form-urlencoded`

Codificação padrão e fácil de usar normalmente associada ao processamento de formulários tradicional

`application/json`

Comum para dados (não pertencentes a arquivos) enviados com `fetch`

multipart/form-data

Codificação para ser usada quando é preciso transferir arquivos

Agora que abordamos como levar dados do usuário para nosso servidor, examinaremos os *cookies* e as *sessões*, que também ajudam a sincronizar o servidor e o cliente.

Cookies e sessões

Neste capítulo, você aprenderá como usar cookies e sessões para fornecer uma melhor experiência para seus usuários lembrando-se de suas preferências de página a página e até mesmo entre sessões de navegador.

O HTTP é um protocolo *stateless*. Isso significa que, quando você carrega uma página em seu navegador e depois navega para outra página no mesmo site, o servidor e o navegador não têm uma maneira intrínseca de saber que o mesmo navegador está visitando o mesmo site. Outra forma de dizer isso é que a maneira de a web funcionar é com *cada requisição HTTP contendo todas as informações necessárias apenas para o servidor atendê-la*.

No entanto, isso é um problema: se a história terminasse aqui, não conseguiríamos nos conectar a nada. O streaming de mídia não funcionaria. Os sites não conseguiriam lembrar-se de nossas preferências de uma página para a outra. Logo, é preciso haver uma maneira de construir um estado que complemente o HTTP, e é aí que os cookies e as sessões entram em cena.

Infelizmente, os cookies têm má reputação por causa das coisas execráveis que as pessoas têm feito com eles. Isso é uma pena porque na verdade eles são muito importantes para o funcionamento da “web moderna” (embora o HTML5 tenha introduzido alguns recursos novos, como o armazenamento local, que poderiam ser usados para a mesma finalidade).

O conceito de cookie é simples: o servidor envia uma informação e o navegador a armazena por algum período de tempo configurável. O servidor é que define qual será a informação. Com frequência, trata-se apenas de um número de ID exclusivo que identifica um navegador específico para que a impressão de que existe um estado seja mantida.

Há algumas coisas importantes que você precisa saber sobre os cookies:

Os cookies não são ocultados do usuário

Todos os cookies que o servidor envia para o cliente ficam disponíveis para este acessar. Não há razão para não podermos enviar algo criptografado para proteger seu conteúdo, mas raramente isso é necessário (pelo menos se você não estiver fazendo nada reprovável!). Os cookies *assinados*, que discutiremos em breve, podem obscurecer seu conteúdo, mas não são de forma alguma uma maneira criptograficamente segura de nos proteger contra olhos curiosos.

O usuário pode excluir os cookies ou desautorizar seu uso

Os usuários têm controle total sobre os cookies, e os navegadores permitem sua exclusão em massa ou individualmente. A não ser que você esteja se comportando de forma inapropriada, não há razão para fazer isso, mas é algo útil durante a execução de testes. Os usuários também podem desautorizar o uso de cookies, o que é mais problemático porque só as aplicações web mais simples podem fazer o que precisam sem cookies.

Os cookies comuns podem ser adulterados

Sempre que um navegador cria uma requisição para o servidor contendo um cookie associado e confiamos cegamente no conteúdo desse cookie, estamos nos arriscando a sofrer ataques. O cúmulo da insensatez, por exemplo, seria executar o código contido em um cookie. Para assegurar que os cookies não sejam adulterados, use cookies assinados.

Os cookies podem ser usados para ataques

Nos últimos anos, surgiu uma categoria de ataques chamada *cross-site scripting* (XSS). Uma técnica de XSS envolve um código JavaScript malicioso modificando o conteúdo dos cookies. Essa é uma razão adicional para não confiarmos no conteúdo dos cookies que são retornados para o servidor. Usar cookies assinados ajuda (a adulteração ficará evidente em um cookie assinado se o usuário ou um código JavaScript malicioso o modificar) e também há uma configuração que especifica que os cookies só devem ser modificados pelo servidor. Esses cookies podem ter utilidade limitada, mas certamente são mais seguros.

Os usuários notarão se você abusar dos cookies

Se você enviar vários cookies para os computadores de seus usuários ou armazenar muitos dados, isso os irritará, o que é algo que devemos evitar. Tente manter o seu uso de cookies em um nível mínimo.

Prefira as sessões aos cookies

Geralmente, podemos usar sessões para manter o estado e é inteligente fazer isso. É mais fácil, não precisamos nos preocupar com o fato de estarmos abusando do armazenamento dos usuários e pode ser mais seguro. É claro que as sessões dependem de cookies, mas, se as usarmos, o Express fará o trabalho pesado para nós.



Os cookies não são mágicos: quando o servidor pretende que o cliente armazene um cookie, ele envia um cabeçalho chamado Set-Cookie contendo pares nome/valor, e, quando um cliente envia uma requisição para um servidor do qual ele tem cookies, envia vários cabeçalhos Cookie contendo o valor dos cookies.

Exteriorizando credenciais

Para tornarmos os cookies seguros, um *segredo* é necessário. O segredo do cookie (cookie secret) é uma string que é conhecida pelo servidor e usada para criptografar cookies seguros antes que eles sejam enviados para o cliente. Não é uma senha que precise ser lembrada, logo, pode ser apenas uma string aleatória. Geralmente uso *um gerador de senha aleatória baseado no xkcd* (<http://bit.ly/2QcjuDb>) para gerar o segredo do cookie ou simplesmente gerar um número aleatório.

É uma prática comum a exteriorização de credenciais de terceiros, como o segredo de um cookie, senhas de banco de dados e tokens de API (Twitter, Facebook etc.). Isso não só facilita a manutenção (tornando fácil localizar e atualizar as credenciais) como também permite omitir o arquivo de credenciais do sistema de controle de versões. É crítico principalmente para repositórios open source hospedados no GitHub ou outros repositórios públicos de controle de código-fonte.

Para seguir essa prática, exteriorizaremos nossas credenciais em um arquivo JSON. Crie um arquivo chamado *.credentials.development.json*:

```
{
  "cookieSecret": "...your cookie secret goes here"
}
```

Esse será o arquivo de credenciais de nosso trabalho de desenvolvimento. Dessa forma, você pode ter diferentes arquivos de credenciais para produção, teste ou outros ambientes, o que será útil.

Adicionaremos uma camada de abstração acima do arquivo de credenciais para tornar mais fácil gerenciar nossas dependências à medida que a aplicação crescer. Nossa versão será muito simples. Crie um arquivo chamado *config.js*:

```
const env = process.env.NODE_ENV || 'development'
const credentials = require(`./credentials.${env}`)
module.exports = { credentials }
```

Agora, para nos certificar de não adicionarmos credenciais acidentalmente ao nosso repositório, incluiremos *.credentials.** no arquivo *.gitignore*. Para importar as credenciais para a aplicação, basta fazer o seguinte:

```
const { credentials } = require('./config')
```

Usaremos esse mesmo arquivo para armazenar outras credenciais posteriormente, mas, por enquanto, só precisamos do segredo do cookie.



Se você está acompanhando com o repositório fornecido, terá de criar seu próprio arquivo de credenciais, já que ele não foi incluído no repositório.

Cookies no Express

Antes de começar a definir e acessar cookies em sua aplicação, você precisa incluir o middleware *cookie-parser*. Primeiro, use *npm install cookie-parser*, e depois escreva o seguinte (*ch09/meadowlark.js* no repositório fornecido):

```
const cookieParser = require('cookie-parser')
app.use(cookieParser(credentials.cookieSecret))
```

Ao fazê-lo, você poderá definir um cookie, assinado ou não, em qualquer

local em que tiver acesso a um objeto de resposta:

```
res.cookie('monster', 'nom nom')
res.cookie('signed_monster', 'nom nom', { signed: true })
```



Os cookies assinados têm precedência sobre os cookies não assinados. Se você nomear seu cookie assinado com `signed_monster`, não poderá ter um cookie não assinado com o mesmo nome (ele será retornado como `undefined`).

Para recuperar o valor de um cookie (se houver) enviado pelo cliente, acesse a propriedade `cookie` ou `signedCookie` do objeto de requisição:

```
const monster = req.cookies.monster
const signedMonster = req.signedCookies.signed_monster
```



Você pode usar a string que quiser como nome de um cookie. Por exemplo, poderíamos ter usado `'signed monster'` em vez de `'signed_monster'`, mas então teríamos de usar a notação de colchete para recuperar o cookie: `req.signedCookies['signed monster']`. Logo, recomendo usar nomes de cookies sem caracteres especiais.

Para excluir um cookie, use `req.clearCookie`:

```
res.clearCookie('monster')
```

Quando você definir um cookie, poderá especificar as seguintes opções:

`domain`

Controla os domínios aos quais os cookies estão associados; essa opção permite atribuir cookies a subdomínios específicos. É bom ressaltar que não é possível definir um cookie para um domínio diferente daquele em que o servidor estiver sendo executado; ele simplesmente não funcionará.

`path`

Controla o path ao qual o cookie é aplicável. Lembre-se de que os paths têm um asterisco implícito depois deles; se você usar o path `/` (o padrão), ele será aplicável a todas as páginas de seu site. Se usar o path `/foo`, ele será aplicável aos paths `/foo`, `/foo/bar` etc.

`maxAge`

Especifica por quanto tempo, em milissegundos, o cliente deve manter o cookie antes de excluí-lo. Se essa opção for omitida, o cookie será excluído quando você fechar seu navegador. (Você também pode especificar uma data para expiração com a opção `expires`, mas a sintaxe é confusa. Recomendo usar `maxAge`.)

`secure`

Especifica que esse cookie só poderá ser enviado por uma conexão segura (HTTPS).

`httpOnly`

Configurar essa opção com `true` especifica que o cookie só poderá ser modificado pelo servidor. Ou seja, JavaScript no lado do cliente não poderá modificá-lo. Isso ajuda a evitar ataques XSS.

`signed`

Configurar essa opção com `true` assina o cookie, tornando-o disponível em `res.signedCookies` em vez de em `res.cookies`. Cookies assinados que tiverem sido adulterados serão rejeitados pelo servidor e o cookie será redefinido com seu valor original.

Examinando os cookies

Provavelmente você vai querer examinar os cookies do sistema como parte de seus testes. A maioria dos navegadores fornece uma maneira de visualizarmos os cookies individuais e os valores que eles armazenam. No Chrome, abra as ferramentas de desenvolvedor e selecione a aba Application. Na árvore à esquerda, você verá Cookies. Expanda essa opção e verá listado o site que está visitando atualmente. Clique nele e aparecerão todos os cookies associados a esse site. Você também pode clicar com o botão direito do mouse no domínio para remover todos os cookies ou clicar com o botão direito em um cookie individual para removê-lo especificamente.

Sessões

Na verdade, as sessões são apenas uma maneira mais conveniente de manter o estado. Para que sejam implementadas, *algo* tem de ser armazenado no cliente; caso contrário, o servidor não conseguirá identificá-lo entre uma requisição e outra. O método normalmente usado para isso é com um cookie contendo um identificador exclusivo. O servidor usará esse identificador para recuperar as informações de sessão apropriadas.

Os cookies não são a única maneira de fazer isso: na época em que predominava o “medo dos cookies” (quando o abuso dos cookies era desmedido), muitos usuários simplesmente os desativavam, e outras maneiras de manter o estado eram projetadas, como com a inclusão das informações de sessão nas URLs. Essas técnicas eram confusas, difíceis e ineficientes e é melhor deixá-las no passado. O HTML 5 fornece outra opção para as sessões chamada *armazenamento local*, que é melhor do que os cookies se você precisar armazenar quantidades maiores de dados. Consulte a *documentação do MDN sobre a propriedade Window.localStorage* (<https://mzl.la/2CDrGo4>) para obter mais informações sobre essa opção.

De um modo geral, há duas maneiras de implementar as sessões: armazenar tudo no cookie ou armazenar somente um identificador exclusivo no cookie e o resto no servidor. A primeira opção é chamada de *sessões baseadas em cookies* e representa apenas uma maneira mais conveniente do que usar cookies. No entanto, ainda significa que tudo que você adicionar à sessão será armazenado no navegador do cliente, o que é uma abordagem que não recomendo. Só a recomendo se você souber que armazenará somente uma pequena quantidade de informações, se não se importar que o usuário tenha acesso a elas e se a situação não crescer para além do planejado com o tempo. Se quiser usar essa abordagem, consulte a página sobre *middleware de sessão baseada em cookies* (<http://bit.ly/2qNv9h6>).

Armazenamentos na memória

Se você preferir armazenar as informações de sessão no servidor, o que recomendo, precisará de um local para armazená-las. A opção para o nível das entradas são as sessões de memória. Elas são fáceis de configurar, mas apresentam uma grande desvantagem: quando você reiniciar o servidor (o

que fará com frequência no decorrer deste livro!), suas informações de sessão desaparecerão. E o que é ainda pior é que, se você aumentar o dimensionamento com vários servidores (consulte o Capítulo 12), um servidor diferente pode atender à requisição; em certos momentos os dados da sessão estarão presentes e em outros não. É claro que essa é uma experiência de usuário inaceitável. No entanto, para nossas necessidades de desenvolvimento e teste, é o suficiente. Veremos como armazenar informações de sessão permanentemente no Capítulo 13.

Primeiro, instale `express-session` (`npm install express-session`); em seguida, após conectar o cookie parser, conecte `express-session` (*ch09/meadowalrk.js* no repositório fornecido):

```
const expressSession = require('express-session')
// certifique-se de conectar o middleware de cookies
//antes do middleware de sessões!
app.use(expressSession({
  resave: false,
  saveUninitialized: false,
  secret: credentials.cookieSecret,
}))
```

O middleware `express-session` aceita um objeto de configuração com as seguintes opções:

`resave`

Força a sessão a ser salva novamente no armazenamento mesmo se a requisição não tiver sido modificada. Geralmente é preferível configurar essa opção com `false`; consulte a documentação de `express-session` para obter mais informações.

`saveUninitialized`

Configurar essa opção com `true` faz sessões novas (não inicializadas) serem salvas no armazenamento, mesmo se não tiverem sido modificadas. Geralmente é preferível configurá-la com `false`, e isso será requerido se você precisar obter permissão do usuário antes de definir um cookie. Consulte a documentação de `express-session` para ver mais informações.

secret

A chave (ou as chaves) usada para assinar o cookie de ID de sessão. Pode ser a mesma chave usada para cookie-parser.

key

Nome do cookie que armazenará o identificador de sessão exclusivo. O padrão é connect.sid.

store

Instância de um armazenamento de sessão. Usa como padrão uma instância de MemoryStore, o que é adequado para o que queremos no momento. Veremos como usar um armazenamento de banco de dados no Capítulo 13.

cookie

Configurações do cookie da sessão (path, domain, secure etc.). Os padrões comuns dos cookies são aplicáveis.

Usando sessões

Uma vez que você tiver definido as sessões, usá-las não poderia ser mais simples; apenas empregue as propriedades da variável session do objeto de requisição:

```
req.session.userName = 'Anonymous'  
const colorScheme = req.session.colorScheme || 'dark'
```

É bom ressaltar que, com as sessões, não precisamos usar o objeto de requisição para recuperar o valor e o objeto de resposta para defini-lo; é tudo feito no objeto de requisição. (O objeto de resposta não tem uma propriedade session). Para excluir uma sessão, você pode usar o operador delete de JavaScript:

```
req.session.userName = null    // esse comando configura 'userName'  
                                // com null, mas não o remove  
  
delete req.session.colorScheme // esse comando remove 'colorScheme'
```

Usando sessões para implementar mensagens flash

As mensagens *flash* (não confundir com o Adobe Flash) são simplesmente uma maneira de dar feedback para os usuários de uma forma que não interrompa a navegação. O modo mais fácil de implementar mensagens flash é usando sessões (você também pode usar querystrings, mas, além de elas produzirem URLs de aparência inadequada, as mensagens flash são incluídas nos bookmarks, e não é isso que queremos). Primeiro definiremos nosso HTML. Usaremos as mensagens de alerta do Bootstrap para exibir nossas mensagens flash, logo, certifique-se de conectá-lo (consulte a documentação “*getting started*” (<http://bit.ly/36YxeYf>) do Bootstrap; você pode conectar os arquivos CSS e JavaScript do Bootstrap ao seu template principal – há um exemplo no repositório fornecido). No arquivo do template, em algum local de destaque (geralmente logo abaixo do cabeçalho do site), adicione o seguinte:

```
{{#if flash}}  
  <div class="alert alert-dismissible alert-{{flash.type}}">  
    <button type="button" class="close"  
      data-dismiss="alert" aria-hidden="true">&times;</button>  
    <strong>{{flash.intro}}</strong> {{{flash.message}}}  
  </div>  
{{/if}}
```

Observe que usamos três chaves para `flash.message`; isso nos permitirá fornecer alguns elementos HTML simples em nossas mensagens (podemos querer enfatizar palavras ou incluir hiperlinks). Agora adicionaremos o middleware que incluirá o objeto flash no contexto se houver um na sessão. Após exibir a mensagem flash uma vez, queremos removê-la da sessão para que ela não seja exibida na próxima requisição. Criaremos um middleware para examinar a sessão, ver se há uma mensagem flash e, se houver, transferi-la para o objeto `res.locals`, tornando-a disponível para as views. Inseriremos nosso middleware em um arquivo chamado *lib/middleware/flash.js*:

```
module.exports = (req, res, next) => {  
  // se houver uma mensagem flash, transfira-a  
  // para o contexto e depois a remova  
  res.locals.flash = req.session.flash  
  delete req.session.flash  
  next()  
}
```

```
}}
```

E, em nosso arquivo *meadowalrk.js*, conectaremos o middleware de mensagens flash, antes das rotas de view:

```
const flashMiddleware = require('./lib/middleware/flash')
app.use(flashMiddleware)
```

Agora veremos como usar a mensagem flash. Suponhamos que estivéssemos inscrevendo usuários para receber uma newsletter e quiséssemos redirecioná-los para o arquivo de newsletters após se inscreverem. Esta seria a aparência de nosso manipulador de formulário:

```
// versão um pouco modificada da regex de email HTML5 oficial do W3C:
// https://html.spec.whatwg.org/multipage/forms.html#valid-e-mail-address
const VALID_EMAIL_REGEX = new RegExp('^[a-zA-Z0-9.!#$%&\'*+\\/=?^_`{|}~-]+@' +
  '[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?' +
  '(?:\\.?[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)+$')
```

```
app.post('/newsletter', function(req, res){
  const name = req.body.name || "", email = req.body.email || ""
  // validação da entrada
  if(VALID_EMAIL_REGEX.test(email)) {
    req.session.flash = {
      type: 'danger',
      intro: 'Validation error!',
      message: 'The email address you entered was not valid.',
    }
    return res.redirect(303, '/newsletter')
  }
  // NewsletterSignup é um exemplo do objeto que você poderia criar;
  // já que cada implementação variará, fica ao seu encargo criar essas
  // interfaces específicas do projeto. Esse exemplo mostra apenas que
  // aparência teria uma implementação típica do Express.
  new NewsletterSignup({ name, email }).save((err) => {
    if(err) {
      req.session.flash = {
        type: 'danger',
        intro: 'Database error!',
        message: 'There was a database error; please try again later.',
      }
    }
  })
})
```



```

    }
    return res.redirect(303, '/newsletter/archive')
  }
  req.session.flash = {
    type: 'success',
    intro: 'Thank you!',
    message: 'You have now been signed up for the newsletter.',
  };
  return res.redirect(303, '/newsletter/archive')
})
})

```

Observe que estamos tendo o cuidado de diferenciar a validação da entrada dos erros de banco de dados. Lembre-se de que, mesmo se executarmos a validação da entrada no front-end (e devemos fazê-lo), também precisamos executá-la no back-end, porque usuários maliciosos podem conseguir passar na validação no front-end.

As mensagens flash são um excelente mecanismo para termos disponível no site, mesmo se outros métodos forem mais apropriados em certas áreas (por exemplo, nem sempre as mensagens flash são apropriadas para “assistentes” de criação de múltiplos formulários ou fluxos de checkout de carrinhos de compra). As mensagens flash também são ótimas durante o desenvolvimento, porque são uma maneira fácil de dar feedback, mesmo se você substituí-las por uma técnica diferente posteriormente. Adicionar o suporte a mensagens flash é uma das primeiras coisas que faço quando defino um site, e usaremos essa técnica no resto do livro.



Já que a mensagem flash está sendo transferida da sessão para `res.locals.flash` no middleware, você tem de executar um redirecionamento para que ela seja exibida. Se quiser exibir uma mensagem flash sem fazer o redirecionamento, use `res.locals.flash` em vez de `req.session.flash`.



O exemplo deste capítulo usou o envio de formulários pelo navegador com redirecionamentos porque normalmente o emprego de sessões para o controle da UI não é usado em aplicações que utilizam Ajax para enviar formulários. Nesse caso, é melhor indicar erros no JSON retornado pelo manipulador de formulário e fazer o front-end modificar

o DOM para exibir mensagens de erro dinamicamente. Sem mencionar que as sessões não são úteis para aplicações renderizadas no front-end, mas raramente elas são usadas para esse fim.

Para que devemos usar as sessões

As sessões serão úteis sempre que você quiser salvar uma preferência do usuário que seja aplicável a várias páginas. Normalmente, elas são usadas para fornecer informações de autenticação do usuário: fazemos login e uma sessão é criada. Depois disso, não precisaremos fazer login novamente sempre que recarregarmos a página. No entanto, as sessões podem ser úteis mesmo sem contas de usuário. É muito comum os sites lembrarem-se de como gostamos que as coisas sejam classificadas ou do formato de data que preferimos – tudo isso sem ser preciso fazer login.

Embora eu recomende que você prefira as sessões aos cookies, é importante entender como os cookies funcionam (principalmente porque eles permitem que as sessões funcionem). Isso o ajudará em problemas de diagnóstico e a entender as considerações de segurança e privacidade de sua aplicação.

Conclusão

Conhecer os cookies e as sessões nos deu uma ideia melhor de como as aplicações web mantêm a impressão de que existe um estado quando o protocolo subjacente (HTTP) é stateless. Aprendemos técnicas de manipulação de cookies e sessões para controlar a experiência do usuário.

Também criamos middleware à medida que avançávamos sem dar muita explicação. No próximo capítulo, nos aprofundaremos no middleware e aprenderemos tudo que é preciso saber sobre ele!

CAPÍTULO 10

Middleware

A essa altura, já tivemos alguma exposição ao middleware: usamos middleware existente (body-parser, cookie-parser, static e express-session, para citar alguns) e criamos os de nossa própria autoria (para adicionar dados meteorológicos ao contexto do template, para configurar mensagens flash e para o manipulador de erros 404). No entanto, o que é exatamente o middleware?

Conceitualmente, *middleware* é uma maneira de encapsular funcionalidades – para ser mais específico, funcionalidades que operem em uma requisição HTTP feita para a aplicação. Na prática, middleware é simplesmente uma função que recebe três argumentos: um objeto de requisição, um objeto de resposta e uma função `next()`, que será explicada em breve. (Também há uma forma que recebe quatro argumentos, para a manipulação de erros, que será abordada no fim deste capítulo).

O middleware é executado no que é conhecido como *pipeline*. Imagine um cano físico carregando água. A água é bombeada para dentro em uma extremidade e depois há aferidores e válvulas antes de ela chegar ao seu destino. A parte importante nessa analogia é a de que a *ordem conta*; se você inserir um aferidor de pressão antes de uma válvula, isso terá um efeito diferente da inserção do aferidor após a válvula. Da mesma forma, se você tiver uma válvula que injete algo na água, tudo que estiver no “fluxo descendente” a partir da válvula conterà o ingrediente adicionado. Em um aplicativo Express, inserimos o middleware no pipeline chamando `app.use`.

Antes do Express 4.0, o pipeline era complexo porque tínhamos de conectar o *roteador* explicitamente. Dependendo de onde conectávamos o roteador, ele poderia ficar fora da ordem, tornando a sequência do pipeline menos clara quando combinávamos o middleware e os manipuladores de rota. No Express

4.0, o middleware e os manipuladores de rota são chamados na ordem em que foram conectados, tornando a sequência muito mais clara.

É prática comum o último middleware do pipeline ser um manipulador genérico para qualquer requisição que não coincida com nenhuma outra rota. Geralmente esse middleware retorna o código de status 404 (Not Found).

No entanto, como uma requisição é “encerrada” no pipeline? É isso que a função `next` passada para cada middleware faz: se você *não* chamar `next()`, a requisição terminará no middleware atual.

Princípios do middleware

Aprender como pensar de maneira flexível sobre o middleware e os manipuladores de rota é essencial para entender como o Express funciona. Estes são os princípios que você deve memorizar:

- Os manipuladores de rota (`app.get`, `app.post` etc. – com frequência chamados coletivamente de `app.METHOD`) podem ser considerados como o middleware que manipula um verbo HTTP específico (`GET`, `POST` etc.). Inversamente, o middleware pode ser considerado como um manipulador de rota que lida com todos os verbos HTTP (o que é equivalente a `app.all`, que manipula qualquer verbo HTTP; há algumas pequenas diferenças com verbos exóticos como `PURGE`, mas, para os verbos comuns, o efeito é o mesmo).
- Os manipuladores de rota requerem um path como primeiro parâmetro. Se quiser que o path sirva para qualquer rota, use `*`. O middleware também pode receber um path como primeiro parâmetro, mas isso é opcional (se for omitido, qualquer path servirá, como se tivéssemos especificado `*`).
- Os manipuladores de rota e o middleware recebem uma função de callback que aceita dois, três ou quatro parâmetros (tecnicamente, você também poderia ter um único parâmetro ou até mesmo nenhum, mas não há uma situação que se beneficie dessas formas). Se houver dois ou três parâmetros, os dois primeiros serão os objetos de requisição e resposta, e o terceiro será a função `next`. Se houver quatro parâmetros, teremos um middleware de *manipulação de erros*, e o primeiro parâmetro será um

objeto de erro, seguido pelo objeto de requisição, pelo objeto de resposta e pela função `next`.

- Se você *não* chamar `next()`, o pipeline será encerrado e nenhum outro manipulador de rota ou middleware será processado. Caso não chame `next()`, você deve enviar uma resposta para o cliente (`res.send`, `res.json`, `res.render` etc.); se não o fizer, ele ficará esperando e seu tempo expirará.
- Quando `next()` é chamada, geralmente não é aconselhável enviar uma resposta para o cliente. Se você a enviar, o middleware ou os manipuladores de rota que estiverem mais à frente no pipeline serão executados, mas qualquer resposta que eles enviarem para o cliente será ignorada.

Exemplos de middleware

Para você ver esses princípios em ação, usaremos alguns exemplos de middleware muito simples (*ch10/00-simple-middleware.js* no repositório fornecido):

```
app.use((req, res, next) => {  
  console.log(`processing request for ${req.url}....`)  
  next()  
})
```

```
app.use((req, res, next) => {  
  console.log('terminating request')  
  res.send('thanks for playing!')  
  // observe que NÃO chamamos next() aqui...isso termina a requisição  
})
```

```
app.use((req, res, next) => {  
  console.log(`whoops, i'll never get called!`)  
})
```

Aqui temos três exemplos de middleware. O primeiro simplesmente registra uma mensagem no console antes de passar a requisição para o próximo middleware do pipeline chamando `next()`. Em seguida, o próximo middleware manipula realmente a requisição. Observe que, se omitíssemos `res.send`,

nenhuma resposta seria retornada para o cliente. Seu tempo expiraria. O último middleware nunca será executado, porque todas as requisições são encerradas no middleware anterior.

Agora consideraremos um exemplo mais complicado e completo (*ch10/01-routing-example.js* no repositório fornecido):

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

app.use((req, res, next) => {
  console.log('\n\nALLWAYS')
  next()
})
app.get('/a', (req, res) => {
  console.log('/a: route terminated')
  res.send('a')
})
app.get('/a', (req, res) => {
  console.log('/a: never called');
})
app.get('/b', (req, res, next) => {
  console.log('/b: route not terminated')
  next()
})
app.use((req, res, next) => {
  console.log('SOMETIMES')
  next()
})
app.get('/b', (req, res, next) => {
  console.log('/b (part 2): error thrown' )
  throw new Error('b failed')
})
app.use('/b', (err, req, res, next) => {
  console.log('/b error detected and passed on')
  next(err)
})
app.get('/c', (err, req) => {
  console.log('/c: error thrown')
  throw new Error('c failed')
```

```

})
app.use('/c', (err, req, res, next) => {
  console.log('/c: error detected but not passed on')
  next()
})
app.use((err, req, res, next) => {
  console.log('unhandled error detected: ' + err.message)
  res.send('500 - server error')
})
app.use((req, res) => {
  console.log('route not handled')
  res.send('404 - not found')
})

const port = process.env.PORT || 3000
app.listen(port, () => console.log( `Express started on http://localhost:${port}`
  + '; press Ctrl-C to terminate.'))

```

Antes de executar esse exemplo, pense em qual será o resultado. Quais são as diferentes rotas? O que o cliente verá? O que será exibido no console? Se você conseguir responder corretamente a todas essas perguntas, entendeu como as rotas funcionam no Express! Dê uma atenção especial à diferença entre a requisição para `/b` e a requisição para `/c`; nos dois casos, houve um erro, mas uma resulta no erro 404 e a outra no erro 500.

Observe que o middleware *deve* ser uma função. Lembre-se de que em JavaScript é muito fácil (e comum) retornarmos uma função a partir de outra função. Por exemplo, você notará que `express.static` é uma função, mas ela foi chamada, logo, deve retornar outra função. Considere o seguinte:

```

app.use(express.static)    // NÃO funcionará como esperado

console.log(express.static()) // registrará "function", indicando
                             // que express.static é uma função
                             // que também retorna uma função

```

Observe também que um módulo pode exportar uma função, que por sua vez pode ser usada diretamente como middleware. Por exemplo, a seguir temos um módulo chamado `lib/tourRequiresWaiver.js` (os pacotes de escalada da Meadowlark Travel requerem um documento de isenção de

responsabilidade):

```
module.exports = (req,res,next) => {
  const { cart } = req.session
  if(!cart) return next()
  if(cart.items.some(item => item.product.requiresWaiver)) {
    cart.warnings.push('One or more of your selected ' +
      'tours requires a waiver.')
  }
  next()
}
```

Poderíamos conectar esse middleware desta forma (*ch10/02-item-waiver.example.js* no repositório fornecido):

```
const requiresWaiver = require('./lib/tourRequiresWaiver')
app.use(requiresWaiver)
```

No entanto, o mais comum é a exportação de um objeto contendo propriedades que representem o middleware. Por exemplo, inseriremos todo o nosso código de validação de carrinho de compras em *lib/cartValidation.js*:

```
module.exports = {

  resetValidation(req, res, next) {
    const { cart } = req.session
    if(cart) cart.warnings = cart.errors = []
    next()
  },

  checkWaivers(req, res, next) {
    const { cart } = req.session
    if(!cart) return next()
    if(cart.items.some(item => item.product.requiresWaiver)) {
      cart.warnings.push('One or more of your selected ' +
        'tours requires a waiver.')
    }
    next()
  },

  checkGuestCounts(req, res, next) {
    const { cart } = req.session
```



```

    if(!cart) return next()
    if(cart.items.some(item => item.guests > item.product.maxGuests )) {
      cart.errors.push('One or more of your selected tours ' +
        'cannot accommodate the number of guests you ' +
        'have selected.')
    }
    next()
  },
}
}

```

Em seguida, você poderia conectar o middleware assim (*ch10/03-more-cart-validation.js* no repositório fornecido):

```

const cartValidation = require('./lib/cartValidation')

app.use(cartValidation.resetValidation)
app.use(cartValidation.checkWaivers)
app.use(cartValidation.checkGuestCounts)

```



No exemplo anterior, o middleware é abortado antecipadamente com a instrução `return next()`. O Express não espera que o middleware retorne um valor (e não faz nada com valores de retorno), logo, essa é apenas uma maneira abreviada de escrever `next(); return`.

Middleware comum

Embora existam milhares de projetos de middleware no npm, há os que são comuns e fundamentais, e pelo menos alguns deles são encontrados em todos os projetos Express não triviais. Parte desses projetos de middleware era tão comum que na verdade foi empacotada com o Express, sendo depois transferida para pacotes individuais. O único middleware que permanece empacotado no Express é `static`.

Essa lista tenta abranger os projetos de middleware mais comuns:

`basicauth-middleware`

Fornece autorização de acesso básica. Lembre-se de que a autorização básica oferece somente o nível de segurança mais rudimentar e você só

deve usá-la com o HTTPS (caso contrário, nomes de usuário e senhas serão transmitidos sem criptografia). Só devemos empregar a autorização básica quando precisarmos de algo rápido e fácil e estivermos usando o HTTPS.

body-parser

Fornece parsing para corpos de requisição HTTP. Provê middleware para o parsing de corpos codificados tanto em URL quanto em JSON, assim como em outros formatos.

busboy, multipart, formidable, multer

Todas essas opções de middleware fazem o parsing de corpos de requisição codificados com multipart/form-data.

compression

Compacta os dados da resposta com o gzip ou o deflate. Isso é algo bom, e seus usuários lhe agradecerão, principalmente os que estiverem em conexões lentas ou móveis. Deve ser conectado no início, antes de qualquer middleware que possa enviar uma resposta. A única coisa que recomendo conectar antes da compactação seria o middleware de depuração ou logging (que não envia respostas). É bom lembrar que, na maioria dos ambientes de produção, a compactação é manipulada por um proxy como o NGINX, tornando esse middleware desnecessário.

cookie-parser

Fornece o suporte a cookies. Consulte o Capítulo 9.

cookie-session

Fornece suporte a sessões com armazenamento de cookies. Geralmente não recomendo essa abordagem para as sessões. Deve ser conectado após cookie-parser. Consulte o Capítulo 9.

express-session

Fornece suporte a sessões com ID (armazenado em um cookie). Usa como padrão o armazenamento na memória, o que não é adequado para a produção, mas pode ser configurado para usar um armazenamento de banco

de dados. Consulte o Capítulo 9 e o Capítulo 13.

`csrf`

Fornece proteção contra ataques de falsificação de requisição entre sites (CSRF, cross-site request forgery). Usa sessões, logo, deve ser conectado depois do middleware `express-session`. Infelizmente, apenas conectar esse middleware não nos protege contra ataques CSRF; consulte o Capítulo 18 para obter mais informações.

`serve-index`

Fornece suporte à listagem de diretório para arquivos estáticos. Não há necessidade de incluir esse middleware a menos que você precise especificamente da listagem de diretório.

`errorhandler`

Fornece rastreamentos de pilha e mensagens de erro para o cliente. Não recomendo conectá-lo em um servidor de produção, porque ele expõe detalhes da implementação, o que pode ter consequências na segurança ou privacidade. Consulte o Capítulo 20 para ver mais informações.

`serve-favicon`

Serve o *favicon* (o ícone que aparece na barra de título do navegador). Não é estritamente necessário; você pode simplesmente inserir um arquivo *favicon.ico* na raiz do diretório estático, mas esse middleware pode melhorar o desempenho. Se você usá-lo, ele deve ser conectado no início da stack de middleware. Ele também permite designar um nome de arquivo diferente de *favicon.ico*.

`morgan`

Fornece suporte ao logging automatizado; todas as requisições serão registradas. Consulte o Capítulo 20 para obter mais informações.

`method-override`

Fornece suporte ao cabeçalho de requisição `x-http-method-override`, que permite que os navegadores “simulem” o uso de métodos HTTP diferentes

de GET e POST. Pode ser útil para a depuração. Só é necessário se você estiver criando APIs.

response-time

Adiciona o cabeçalho X-Response-Time à resposta, fornecendo o tempo de resposta em milissegundos. Geralmente não precisamos desse middleware a não ser quando estamos ajustando o desempenho.

static

Fornece suporte à disponibilização de arquivos estáticos (públicos). Você pode conectar esse middleware várias vezes, especificando diferentes diretórios. Consulte o Capítulo 17 para ver mais detalhes.

vhost

Os hosts virtuais (vhosts), um termo pegado emprestado do Apache, tornam os subdomínios mais fáceis de gerenciar no Express. Consulte o Capítulo 14 para obter mais informações.

Middleware de terceiros

Atualmente, não há um “armazenamento” ou índice abrangente para middleware de terceiros. No entanto, quase todos os projetos de middleware do Express estão disponíveis no npm, logo, se você procurar “Express” e “middleware” no npm, obterá uma boa lista. A documentação oficial do Express também contém uma conveniente *lista de projetos de middleware* (<http://bit.ly/36UrbnL>).

Conclusão

Neste capítulo, examinamos o que é middleware, como criar nosso próprio middleware e como ele é processado como parte de uma aplicação Express. Se agora você está achando que uma aplicação Express é simplesmente uma coleção de middleware, está começando a entender o Express! Até mesmo os manipuladores de rota que usamos anteriormente são apenas casos especializados de middleware.

No próximo capítulo, examinaremos outra necessidade comum da infraestrutura: o envio de emails (e pode ter certeza de que haverá algum middleware envolvido!).

CAPÍTULO 11

Enviando emails

Uma das principais maneiras de a aplicação se comunicar com o resto do mundo é por emails. Seja para o registro de usuários ou para instruções de redefinição de senha e mensagens promocionais, a possibilidade de enviar emails é um recurso importante. Neste capítulo, você aprenderá como formatar e enviar emails com o Node e o Express para ajudar na comunicação com seus usuários.

O Node e o Express não têm nenhum expediente interno de envio de emails, logo, temos de usar um módulo de terceiros. O pacote que recomendo é o excelente *Nodemailer* (<http://bit.ly/2Ked7vy>) de Andris Reinman. Antes de examinar a configuração do Nodemailer, veremos alguns aspectos básicos dos emails.

SMTP, MSAs e MTAs

O padrão para o envio de emails é o Simple Mail Transfer Protocol (SMTP). Embora seja possível usar o SMTP para enviar um email diretamente para o servidor de email do destinatário, em geral essa não é uma boa ideia: se você não for um “emitente confiável” como o Google ou o Yahoo!, provavelmente seu email será enviado diretamente para a caixa de spam. É melhor usar um agente de envio de emails (MSA, mail submission agent), que distribuirá o email por canais confiáveis, diminuindo as chances de ele ser marcado como spam. Além de assegurar que o email chegue, o MSA manipula incômodos como falhas temporárias e devolução de emails. A parte final da equação é o agente de transferência de emails (MTA, mail transfer agent), o serviço que envia realmente o email para seu destino final. Para os fins deste livro, o *MSA*, o *MTA* e o *servidor SMTP* são basicamente equivalentes.

Você precisará de acesso a um MSA. Poderíamos começar usando um serviço de email de consumidor gratuito como o Gmail, o Outlook ou o Yahoo!, mas esses serviços não são mais tão amigáveis a emails automatizados como eram no passado (em um esforço para eliminar o abuso). Felizmente, há dois excelentes serviços de email à nossa escolha que têm uma opção gratuita para uso de baixo volume: o *Sendgrid* (<https://sendgrid.com>) e o *Mailgun* (<https://www.mailgun.com>). Usei os dois e gosto de ambos. Os exemplos deste livro usarão o SendGrid.

Se você está trabalhando para uma empresa, talvez ela tenha o próprio MSA; entre em contato com o departamento de TI e pergunte se há um retransmissor de SMTP disponível para o envio de emails automatizados.

Se está usando o SendGrid ou o Mailgun, vá em frente e configure sua conta agora. Para o SendGrid, você terá de criar uma chave de API (que será sua senha SMTP).

Recebendo emails

A maioria dos sites só precisa do recurso de *envio* de emails, como para o envio de instruções de redefinição de senha e emails promocionais. No entanto, algumas aplicações também precisam receber emails. Um bom exemplo seria um sistema de gerenciamento de problemas que enviasse um email quando alguém atualizasse um problema, e, após o email ser respondido, o problema fosse atualizado automaticamente com a resposta.

Infelizmente, receber emails é muito mais complicado e não será abordado neste livro. Se você precisa dessa funcionalidade, deve permitir que seu provedor de email faça a manutenção na caixa de correio e é necessário ter um processo periódico para acessá-la com um agente IMAP como o *imap-simple* (<http://bit.ly/2qQK0r5>).

Cabeçalhos dos emails

Um email é composto de duas partes: o cabeçalho e o corpo (da mesma forma que uma requisição HTTP). O *cabeçalho* contém informações sobre o email: quem enviou, para quem foi enviado, a data em que foi recebido, o

assunto e assim por diante. Esses são os cabeçalhos normalmente exibidos para o usuário em uma aplicação de email, mas há muitos outros. A maioria dos clientes de email permite que examinemos os cabeçalhos; se você nunca fez isso, recomendo que o faça. Os cabeçalhos fornecem todas as informações sobre como o email chegou até nós; cada servidor e MTA pelos quais o email passou estarão listados no cabeçalho.

Com frequência, as pessoas se surpreendem por alguns cabeçalhos, como o de endereço de “emissão” (endereço “from”), poderem ser definidos arbitrariamente pelo emitente. Quando especificamos um endereço de “emissão” diferente da conta a partir da qual ocorreu o envio, geralmente isso é chamado de *spoofing*. Não há nada que nos impeça de enviar um email com o endereço de “emissão” de Bill Gates <billg@microsoft.com>. Não estou recomendando que você faça isso, apenas esclarecendo o fato de que é possível definir certos cabeçalhos com o que quisermos. Às vezes há razões legítimas para fazê-lo, mas não devemos abusar.

No entanto, qualquer email enviado *deve* ter um endereço de “emissão”. Esse requisito pode causar problemas no envio de emails automatizados, e é por isso que costumamos ver emails com endereços de retorno como DO NOT REPLY <do-not-reply@meadowlarktravel.com>. Usar essa abordagem ou fazer os emails automatizados virem de um endereço como o da Meadowlark Travel, <info@meadowlarktravel.com> é uma decisão sua; porém, se usar a última abordagem, deve estar preparado para responder a emails que vierem para *info@meadowlarktravel.com*.

Formatos dos emails

Quando a internet era novidade, todos os emails eram simplesmente texto ASCII. O mundo mudou muito desde então e as pessoas querem enviar emails em diferentes linguagens e fazer coisas mais sofisticadas como incluir texto formatado, imagens e anexos. Foi aí que tudo começou a ficar confuso: os formatos e a codificação dos emails são uma horrível mistura de técnicas e padrões.

Felizmente, não precisamos nos preocupar com essas complexidades. O Nodemailer as manipulará para nós. O que é importante saber é que o email

pode estar em texto puro (Unicode) ou HTML.

Quase todas as aplicações de email modernas dão suporte ao email HTML, logo, geralmente é muito seguro formatar os emails em HTML. Mesmo assim, há “puristas do texto” que evitam o email HTML, portanto, recomendo sempre incluir email tanto em texto quanto em HTML. Se você não quiser ter de escrever emails em texto e em HTML, o Nodemailer dá suporte a um atalho que gera automaticamente a versão em texto puro a partir do HTML.

Email HTML

O email HTML é um tópico que poderia ter seu próprio livro. Infelizmente, criar esse tipo de email não é tão simples como escrever HTML para um site: a maioria dos clientes de email dá suporte apenas a um pequeno subconjunto do HTML. Quase sempre, temos de escrever HTML como se ainda estivéssemos em 1996; não é muito divertido. Especificamente, é preciso voltar a usar tabelas para o layout (o que, na verdade, é triste).

Se você tem experiência com problemas de compatibilidade entre os navegadores e o HTML, sabe que isso pode ser uma dor de cabeça. Os problemas de compatibilidade dos emails são muito piores. Ainda bem que há algumas coisas que podem ajudar.

Primeiro, recomendo que você leia o excelente *artigo da plataforma MainChimp sobre escrever emails em HTML* (<http://bit.ly/33CsaXs>). Eles se saem bem ao abordar os aspectos básicos e explicar o que precisamos lembrar ao escrever emails em HTML.

O próximo recurso ajuda a economizar tempo: é o *HTML Email Boilerplate* (<http://bit.ly/2qJ1XIe>). É basicamente um template muito bem elaborado e rigorosamente testado para emails HTML.

Para concluir, temos a execução de testes. Você se informou sobre como escrever emails em HTML e está usando o HTML Email Boilerplate, mas os testes são a única maneira de saber com certeza se seu email será aceito no Lotus Notes 7 (sim, as pessoas ainda o usam). Está pensando em instalar 30 clientes de email diferentes para testar um único email? Não é preciso.

Felizmente, há um ótimo serviço que faz isso para nós: o *Litmus* (<http://bit.ly/2NI6JPo>). Não é um serviço barato; os planos começam em cerca de 100 dólares por mês. No entanto, para o envio de muitos emails promocionais, ele é difícil de superar.

Por outro lado, se a formatação que você usa é simples, não precisa de um serviço de teste caro como o Litmus. Se estiver usando coisas como cabeçalhos, texto em negrito/italico, regras horizontais e alguns links de imagem, está seguro.

Nodemailer

Primeiro, temos de instalar o pacote Nodemailer:

```
npm install nodemailer
```

Em seguida, demande o uso do pacote e crie uma instância do Nodemailer (o que no jargão é chamado de *transporte*):

```
const nodemailer = require('nodemailer')

const mailTransport = nodemailer.createTransport({

  auth: {
    user: credentials.sendgrid.user,
    pass: credentials.sendgrid.password,
  }
})
```

Observe que estamos usando o módulo de credenciais que definimos no Capítulo 9. Você terá de atualizar seu arquivo *.credentials.development.json* conforme apropriado:

```
{
  "cookieSecret": "your cookie secret goes here",
  "sendgrid": {
    "user": "your sendgrid username",
    "password": "your sendgrid password"
  }
}
```

As opções de configuração comuns do SMTP são a porta, o tipo de

autenticação e opções de TLS. No entanto, a maioria dos principais serviços de email usa as opções padrão. Para saber que configurações usar, consulte a documentação de seu serviço de email (tente pesquisar *envio de email SMTP*, *configuração SMTP* ou *retransmissão de SMTP*). Se tiver problemas para enviar emails com o SMTP, pode ser preciso verificar as opções; consulte a *documentação do Nodemailer* (<https://nodemailer.com/smtp>) para ver uma lista completa das opções suportadas.



Se você está acompanhando com o repositório fornecido, verá que não há configurações no arquivo de credenciais. No passado, muitos leitores entraram em contato comigo perguntando por que o arquivo está ausente ou vazio. Não forneço credenciais válidas pela mesma razão que você deve tomar cuidado com as suas credenciais! Confio em você, querido leitor, mas não a ponto de fornecer minha senha de email!

Enviando emails

Agora que temos nossa instância de transporte de email, podemos enviar emails. Começaremos com um exemplo simples que envia email em texto somente para um destinatário (*ch11/00-smtp.js* no repositório fornecido):

```
try {
  const result = await mailTransport.sendMail({
    from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
    to: 'joecustomer@gmail.com',
    subject: 'Your Meadowlark Travel Tour',
    text: 'Thank you for booking your trip with Meadowlark Travel. ' +
      'We look forward to your visit!',
  })
  console.log('mail sent successfully: ', result)
} catch(err) {
  console.log('could not send mail: ' + err.message)
}
```



Nos exemplos de código desta seção, estou usando endereços de email fictícios como *joecustomer@gmail.com*. Para fins de verificação, pode ser melhor alterar esses endereços para um email que esteja sob o seu controle para que você possa ver o que está ocorrendo. Caso contrário, o

pobre *joecustomer@gmail.com* receberá vários emails sem sentido!

Você notará que estamos manipulando erros aqui, mas é importante entender que a inexistência de erros não significa necessariamente que seu email foi distribuído com sucesso para o *destinatário*. O parâmetro *error* do callback só será usado se houver um problema de comunicação com o MSA (como um erro de rede ou de autenticação). Se o MSA não conseguir distribuir o email (por exemplo, devido a um endereço de email inválido ou um usuário desconhecido), você terá de verificar a atividade de sua conta em seu serviço de email, o que pode ser feito na interface administrativa ou por uma API.

Se você precisar que seu sistema determine automaticamente se o email foi distribuído com sucesso, terá de usar a API de seu serviço de email. Consulte a documentação da API do serviço de email para obter mais informações.

Enviando emails para vários destinatários

O Nodemail dá suporte ao envio de emails para vários destinatários com o uso de vírgulas (*ch11/01-multiple-recipients.js* no repositório fornecido):

```
try {
  const result = await mailTransport.sendMail({
    from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
    to: 'joe@gmail.com, "Jane Customer" <jane@yahoo.com>, ' +
      'fred@hotmail.com',
    subject: 'Your Meadowlark Travel Tour',
    text: 'Thank you for booking your trip with Meadowlark Travel. ' +
      'We look forward to your visit!',
  })
  console.log('mail sent successfully: ', result)
} catch(err) {
  console.log('could not send mail: ' + err.message)
}
```

Observe que, nesse exemplo, combinamos endereços de email simples (*joe@gmail.com*) com endereços que especificam o nome do destinatário (“Jane Customer” <*jane@yahoo.com*>). Essa sintaxe é permitida.

Ao enviar um email para vários destinatários, você deve tomar cuidado e observar os limites de seu MSA. O SendGrid, por exemplo, recomenda

limitar o número de destinatários (ele recomenda não mais que mil em um único email). Se você estiver enviando um email em grande quantidade, pode ser melhor distribuir várias mensagens, cada uma com vários destinatários (*ch11/02-many-recipients.js* no repositório fornecido):

```
// largeRecipientList é um array de endereços de email
const recipientLimit = 100
const batches = largeRecipientList.reduce((batches, r) => {
  const lastBatch = batches[batches.length - 1]
  if(lastBatch.length < recipientLimit)
    lastBatch.push(r)
  else
    batches.push([r])
  return batches
}, [[]])
try {
  const results = await Promise.all(batches.map(batch =>
    mailTransport.sendMail({
      from: '"Meadowlark Travel", <info@meadowlarktravel.com>',
      to: batch.join(', '),
      subject: 'Special price on Hood River travel package!',
      text: 'Book your trip to scenic Hood River now!',
    })
  ))
  console.log(results)
} catch(err) {
  console.log('at least one email batch failed: ' + err.message)
}
```

Opções melhores para o envio de email em massa

Embora você certamente possa enviar um email em massa com o Nodemailer e um MSA apropriado, deve pensar bem antes de fazê-lo. Uma campanha de email responsável deve fornecer uma maneira de as pessoas cancelarem o recebimento de emails promocionais, e essa não é uma tarefa trivial. Multiplique isso por cada lista de recebimento que você mantiver (poderia ter uma newsletter semanal e uma campanha de anúncios especial, por exemplo). Essa é uma área na qual é melhor não reinventar a roda. Serviços como o

Emma (<https://myemma.com>), o *Mailchimp* (<http://mailchimp.com>) e o *Campaign Monitor* (<http://www.campaignmonitor.com>) oferecem tudo que é necessário, inclusive ótimas ferramentas para o monitoramento do sucesso das campanhas de email. Eles são baratos e recomendo o seu uso para emails promocionais, newsletters etc.

Enviando emails HTML

Até agora, enviamos apenas emails em texto puro, mas atualmente a maioria das pessoas espera algo com uma aparência um pouco melhor. O Nodemailer permite enviar versões tanto em HTML quanto em texto puro no mesmo email, deixando que o cliente de email escolha que versão será exibida (geralmente HTML). (*ch11/03-html-email.js* no repositório fornecido):

```
const result = await mailTransport.sendMail({
  from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
  to: 'joe@gmail.com, "Jane Customer" <jane@yahoo.com>, ' +
    'fred@hotmail.com',
  subject: 'Your Meadowlark Travel Tour',
  html: '<h1>Meadowlark Travel</h1>\n<p>Thanks for book your trip with ' +
    'Meadowlark Travel. <b>We look forward to your visit!</b>',
  text: 'Thank you for booking your trip with Meadowlark Travel. ' +
    'We look forward to your visit!',
})
```

Dá muito trabalho fornecer versões em HTML e em texto, principalmente se seus usuários preferirem emails somente em texto. Se quiser economizar tempo, você pode escrever seus emails em HTML e usar um pacote como o *html-to-formatted-text* (<http://bit.ly/34RX8Lq>) para gerar texto automaticamente a partir do HTML. Apenas lembre-se de que ele não terá uma qualidade tão alta quanto a do texto criado manualmente; nem sempre o HTML fornece uma conversão clara).

Imagens em emails HTML

Embora seja possível embutir imagens em emails HTML, não recomendo fazê-lo. Elas avolumam o email, o que não é considerado boa prática. Em vez disso, você deve disponibilizar as imagens que deseja usar em seu servidor

web e vinculá-las apropriadamente a partir do email.

É melhor usar um local dedicado em sua pasta de arquivos estáticos para as imagens dos emails. É aconselhável até mesmo manter separados os arquivos que você usa tanto nos emails quanto nos sites. Isso reduz a chance de o layout dos emails ser afetado negativamente.

Adicionaremos alguns recursos de email ao nosso projeto Meadowlark Travel. Em seu diretório *public*, crie um subdiretório chamado *email*. Você poderá inserir *logo.png* nele, além de outras imagens que quiser usar no email. Assim, poderá usar essas imagens diretamente em seu email:

```

```



Isso deixa claro que você não deseja usar *localhost* ao enviar emails para outras pessoas; provavelmente elas não terão nem mesmo um servidor sendo executado, quanto mais na porta 3000! Dependendo de seu cliente de email, você deve poder usar *localhost* em emails para fins de teste, mas isso não funcionará fora de seu computador. No Capítulo 17, discutiremos algumas técnicas que suavizam a transição do desenvolvimento para a produção.

Usando views para enviar emails HTML

Até agora, inserimos nosso HTML em strings no código JavaScript, uma prática que você deve tentar evitar. Nosso código HTML tem sido muito simples, mas examine o *HTML Email Boilerplate* (<http://bit.ly/2qJ1XIe>): você deseja inserir todo o boilerplate em uma string? Claro que não.

Felizmente, podemos nos beneficiar das views para manipular isso. Considere nosso exemplo de email “Thank you for booking your trip with Meadowlark Travel”, que expandiremos um pouco. Suponhamos que tivéssemos um objeto de carrinho de compras contendo informações do pedido. Esse objeto será armazenado na sessão. Digamos que a última etapa de nosso processo de criação de pedidos fosse um formulário processado por */cart/checkout*, que envia um email de confirmação. Começaremos criando uma view para a página de agradecimento, *views/cart-thank-you.handlebars*:

```
<p>Thank you for booking your trip with Meadowlark Travel,
```

```
{{cart.billing.name}}!
```

<p>Your reservation number is {{cart.number}}, and an email has been sent to {{cart.billing.email}} for your records.</p>

Em seguida, criaremos um template para o email. Baixe o HTML Email Boilerplate e insira *views/email/cart-thank-you.handlebars*. Edite o arquivo e modifique o corpo:

```
<table cellpadding="0" cellspacing="0" border="0" id="backgroundTable">
  <tr>
    <td valign="top">
      <table cellpadding="0" cellspacing="0" border="0" align="center">
        <tr>
          <td width="200" valign="top"></td>
        </tr>
        <tr>
          <td width="200" valign="top"><p>
            Thank you for booking your trip with Meadowlark Travel,
            {{cart.billing.name}}.</p><p>Your reservation number
            is {{cart.number}}.</p></td>
        </tr>
        <tr>
          <td width="200" valign="top">Problems with your reservation?
            Contact Meadowlark Travel at
            <span class="mobile_link">555-555-0123</span>.</td>
        </tr>
      </table>
    </td>
  </tr>
</table>
```



Já que você não pode usar endereços *localhost* em emails, se seu site ainda não estiver online, poderá usar um serviço de placeholder para elementos gráficos. Por exemplo, *http://placeholder.it/100x100* serve dinamicamente um elemento gráfico quadrado de 100 pixels. Essa técnica é empregada com muita frequência em imagens apenas para posicionamento (FPO, for-placement-only) e para fins de layout.

Agora podemos criar uma rota para a página Thank-you de nosso carrinho (*ch11/04-rendering-html-email.js* no repositório fornecido):

```
app.post('/cart/checkout', (req, res, next) => {
  const cart = req.session.cart
  if(!cart) next(new Error('Cart does not exist.'))
  const name = req.body.name || "", email = req.body.email || ""
  // validação da entrada
  if(!email.match(VALID_EMAIL_REGEX))
    return res.next(new Error('Invalid email address.'))
  // atribui um ID de carrinho aleatório; normalmente
  // usaríamos um ID de banco de dados aqui
  cart.number = Math.random().toString().replace(/^0\./, "")
  cart.billing = {
    name: name,
    email: email,
  }
  res.render('email/cart-thank-you', { layout: null, cart: cart },
    (err,html) => {
      console.log('rendered email: ', html)
      if(err) console.log('error in email template')
      mailTransport.sendMail({
        from: "Meadowlark Travel: info@meadowlarktravel.com",
        to: cart.billing.email,
        subject: 'Thank You for Book your Trip with Meadowlark Travel',
        html: html,
        text: htmlToFormattedText(html),
      })
      .then(info => {
        console.log('sent! ', info)
        res.render('cart-thank-you', { cart: cart })
      })
      .catch(err => {
        console.error('Unable to send confirmation: ' + err.message)
      })
    }
  )
})
```

Observe que estamos chamando `res.render` duas vezes. Normalmente o

chamaríamos apenas uma vez (chamá-lo duas vezes exibirá apenas os resultados da primeira chamada). No entanto, nesse caso, estamos ignorando o processo de renderização normal na primeira vez que o chamamos: repare que fornecemos um callback. Ele impede que os resultados da view sejam renderizados para o navegador. Em vez disso, o callback recebe a view renderizada no parâmetro `html`: só precisamos pegar esse HTML renderizado e enviar o email! Especificamos `layout: null` para impedir que nosso arquivo de layout seja usado, porque temos tudo no template de email (uma abordagem alternativa seria criar um arquivo de layout separado para os emails e usá-lo). Para concluir, chamamos `res.render` novamente. Dessa vez, os resultados serão renderizados para a resposta HTML como de praxe.

Encapsulando a funcionalidade de uso de emails

Se você estiver usando muitos emails em seu site, pode ser melhor encapsular essa funcionalidade. Presumiremos que você deseja que seu site sempre envie emails a partir do mesmo emitente (“Meadowlark Travel” `<info@meadowlarktravel.com>`) e também que os emails sejam enviados em HTML com texto gerado automaticamente. Crie um módulo chamado *lib/email.js* (*ch11/lib/email.js* no repositório fornecido):

```
const nodemailer = require('nodemailer')
const htmlToFormattedText = require('html-to-formatted-text')

module.exports = credentials => {

  const mailTransport = nodemailer.createTransport({
    host: 'smtp.sendgrid.net',
    auth: {
      user: credentials.sendgrid.user,
      pass: credentials.sendgrid.password,
    },
  })

  const from = "Meadowlark Travel" <info@meadowlarktravel.com>
  const errorRecipient = 'youremail@gmail.com'

  return {
```

```

    send: (to, subject, html) =>
      mailTransport.sendMail({
        from,
        to,
        subject,
        html,
        text: htmlToFormattedText(html),
      }),
  },
}

}

```

Agora só precisamos executar o código a seguir para enviar um email (*ch11/05-email-library.js* no repositório fornecido):

```

const emailService = require('./lib/email')(credentials)

emailService.send(email, "Hood River tours on sale today!",
  "Get 'em while they're hot!")

```

Conclusão

Neste capítulo, você aprendeu os aspectos básicos de como o email é distribuído na internet. Se acompanhou o material, configurou um serviço de email gratuito (provavelmente o SendGrid ou o Mailgun) e o usou para enviar email em texto e HTML. Também aprendeu como usar o mesmo mecanismo de renderização por template que usamos para renderizar HTML em nossas aplicações Express a fim de renderizar o código HTML dos emails.

O email continua sendo uma maneira importante para uma aplicação se comunicar com os usuários. Cuidado para não abusar desse poder! Se você é como eu, possui uma caixa de entrada cheia de emails automatizados em grande parte ignorados. Quando se trata de emails automatizados, menos é mais. Há razões úteis e legítimas para a aplicação enviar um email para os usuários, mas você deve sempre se perguntar: “Meus usuários precisam *mesmo* desse email? Há outra maneira de transmitir essa informação?”.

Agora que abordamos alguma infraestrutura básica necessária para a criação

de aplicações, nos dedicaremos a falar sobre o conseqüente lançamento de nossa aplicação no ambiente de produção e os tipos de coisas que devemos considerar para tornar esse lançamento bem-sucedido.

Preocupações de produção

Embora eu ache prematuro começar a discutir preocupações de produção nesse momento, você pode economizar muito tempo e evitar dores de cabeça se pensar na produção desde o início. O dia do lançamento chegará sem que você sinta.

Neste capítulo, você conhecerá o suporte do Express a diferentes ambientes de execução, verá métodos para aumentar a escala de seu site e aprenderá como monitorar sua integridade. Além disso, verá como simular um ambiente de produção para teste e desenvolvimento e como executar o teste de estresse para poder identificar problemas de produção antes que ocorram.

Ambientes de execução

O Express dá suporte ao conceito de *ambientes de execução*: uma maneira de executar a aplicação no modo de produção, desenvolvimento ou teste. Na verdade, você pode ter quantos ambientes diferentes quiser. Por exemplo, poderia ter um ambiente de preparação, ou um ambiente de treinamento. No entanto, lembre-se de que o desenvolvimento, a produção e o teste são ambientes “padrão”, e com frequência os projetos de middleware tanto do Express quanto de terceiros tomam decisões de acordo com esses ambientes. Em outras palavras, se você tiver um ambiente de “preparação”, não poderá fazê-lo herdar automaticamente as propriedades de um ambiente de produção. Logo, recomendo que use os padrões de produção, desenvolvimento e teste.

Embora seja possível especificar o ambiente de execução chamando `app.set('env', 'production')`, não é aconselhável fazê-lo; se você o fizer, sua aplicação será sempre executada nesse ambiente, independentemente da situação. Pior, ela pode começar a ser executada em um ambiente e passar

para outro.

É preferível especificar o ambiente de execução usando a variável de ambiente `NODE_ENV`. Modificaremos nosso aplicativo para que ele relate o modo em que está sendo executado chamando `app.get('env')`:

```
const port = process.env.PORT || 3000
app.listen(port, () => console.log(`Express started in ` +
  `${app.get('env')}` mode at http://localhost:${port}` +
  `; press Ctrl-C to terminate.`))
```

Se você iniciar seu servidor agora, verá que a execução está ocorrendo no modo de desenvolvimento; é o padrão quando não especificamos algo diferente. Tentaremos colocá-lo no modo de produção:

```
$ export NODE_ENV=production
$ node meadowlark.js
```

Se você está usando o Unix/BSD, há uma sintaxe útil que permite modificar o ambiente somente pelo tempo de duração desse comando:

```
$ NODE_ENV=production node meadowlark.js
```

Esse código executará o servidor no modo de produção, mas, uma vez que ele terminar, a variável de ambiente `NODE_ENV` não será modificada. Gosto muito desse atalho, e ele reduz a chance de eu deixar acidentalmente as variáveis de ambiente configuradas com valores que não desejo necessariamente para tudo.



Se você iniciar o Express no modo de produção, pode ver avisos sobre componentes que não são adequados para uso nesse modo. Se está acompanhando os exemplos deste livro, verá que `connect.session` está usando um armazenamento na memória, o que não é apropriado para um ambiente de produção. Quando mudarmos para um armazenamento de banco de dados no Capítulo 13, esse aviso desaparecerá.

Configuração específica do ambiente

Apenas mudar o ambiente de execução não ajuda muito, embora o Express exiba mais avisos no console no modo de produção (por exemplo, informando sobre módulos que são obsoletos e serão removidos no futuro).

Além disso, no modo de produção o armazenamento de views em cache é ativado por padrão (consulte o Capítulo 7).

O ambiente de execução é principalmente uma ferramenta para nos ajudar, permitindo que tomemos decisões facilmente sobre como a aplicação deve se comportar nos diferentes ambientes. Recomendo que você tente reduzir as diferenças entre seus ambientes de desenvolvimento, teste e produção. Isto é, você deve usar esse recurso moderadamente. Se seus ambientes de desenvolvimento ou teste diferirem muito do de produção, você estará aumentando as chances de haver um comportamento diferente na produção, o que pode gerar mais falhas (ou falhas mais difíceis de detectar). Algumas diferenças são inevitáveis; por exemplo, se seu aplicativo é altamente baseado em banco de dados, é melhor não mexer com o banco de dados de produção durante o desenvolvimento, e essa situação seria uma boa candidata a receber uma configuração específica do ambiente. Outra área de baixo impacto seria um logging mais verboso. Há muitas coisas que você pode querer registrar no desenvolvimento que seria desnecessário registrar na produção.

Adicionaremos logging ao servidor. O detalhe é que queremos comportamentos diferentes para a produção e o desenvolvimento. Para o desenvolvimento, podemos deixar os padrões, mas, para a produção, queremos fazer o registro em um arquivo. Usaremos o morgan (não se esqueça de usar `npm install morgan`), que é o middleware de logging mais comum (*ch12/00-logging.js* no repositório fornecido):

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

switch(app.get('env')) {
  case 'development':
    app.use(morgan('dev'))
    break
  case 'production':
    const stream = fs.createWriteStream(__dirname + '/access.log',
      { flags: 'a' })
    app.use(morgan('combined', { stream }))
    break
}
```

}

Se você iniciar o servidor como faria normalmente (`node meadowlark.js`) e visitar o site, verá a atividade registrada no console. Para ver como a aplicação se comporta no modo de produção, execute-a com `NODE_ENV=production`. Agora, se você acessar a aplicação, não verá nenhuma atividade no terminal (o que é desejável para um servidor de produção), mas ela estará registrada no *Formato de Log Combinado do Apache* (<http://bit.ly/2NGC592>), que é a base de muitas ferramentas de servidor.

Fizemos isso criando um stream de gravação anexável (`{ flags: 'a' }`) e passando-o para a configuração do morgan. O morgan tem muitas opções; para ver todas, acesse *sua documentação* (<http://bit.ly/32H5wMr>).



No exemplo anterior, estamos usando `__dirname` para armazenar o log de requisição em um subdiretório do próprio projeto. Se você adotar essa abordagem, deve adicionar log ao arquivo `.gitignore`. Alternativamente, poderíamos usar uma abordagem Unix-like e salvar os logs em um subdiretório de `/var/log`, como o Apache faz por padrão.

Ressaltarei novamente que você deve pensar bem ao fazer escolhas de configuração específica do ambiente. Lembre-se sempre de que, quando o site estiver online, as instâncias de produção serão (ou deveriam ser) executadas no modo de produção. Sempre que você ficar tentado a fazer uma modificação específica do ambiente de desenvolvimento, deve pensar primeiro nas consequências de QA que ela pode ter na produção. Veremos um exemplo mais robusto de configuração específica do ambiente no Capítulo 13.

Executando seu processo Node

Até agora, executamos nossa aplicação chamando-a diretamente com `node` (por exemplo, `node meadowlark.js`). Isso funciona para o desenvolvimento e o teste, mas apresenta desvantagens para a produção. Especificamente, não haverá proteções se seu aplicativo falhar ou for encerrado. Um *gerenciador de processos* robusto pode resolver esse problema.

Dependendo de sua solução de hospedagem, talvez você não precise de um

gerenciador de processos se um for fornecido por ela. Isto é, o provedor de hospedagem lhe dará uma opção de configuração que aponte para o arquivo de sua aplicação, e ela manipulará o gerenciamento de processos.

No entanto, se você precisar gerenciar o processo por conta própria, há duas opções de gerenciadores de processos populares:

- *Forever* (<https://github.com/foreversd/forever>)
- *PM2* (<https://github.com/Unitech/pm2>)

Já que os ambientes de produção podem variar muito, não entraremos nos detalhes de instalação e configuração de um gerenciador de processos. Tanto o Forever quanto o PM2 têm uma ótima documentação, e você pode instalá-los e usá-los em sua máquina de desenvolvimento para aprender como configurá-los.

Usei os dois e não tenho uma preferência. O Forever é um pouco mais simples e fácil de entender, e o PM2 oferece mais recursos.

Se você quiser testar um gerenciador de processos sem gastar muito tempo, recomendo usar o Forever. É possível testá-lo em duas etapas. Primeiro, instale-o:

```
npm install -g forever
```

Em seguida, inicie sua aplicação com o Forever (execute esse código a partir da raiz da aplicação):

```
forever start meadowlark.js
```

Agora sua aplicação está sendo executada... e continuará em execução mesmo se você fechar a janela do terminal! Você pode reiniciar o processo com `forever restart meadowlark.js` e interrompê-lo com `forever stop meadowlark.js`.

Começar a usar o PM2 é um pouco mais complicado, mas vale a pena examiná-lo se você precisar usar seu próprio gerenciador de processos para a produção.

Dimensionando seu site

Atualmente, scaling em geral significa uma entre duas coisas: scaling up ou scaling out. *Scaling up* quer dizer tornar os servidores mais poderosos: CPUs

mais rápidas, uma arquitetura melhor, mais núcleos (cores), mais memória etc. *Scaling out*, por outro lado, significa simplesmente mais servidores. Com o aumento da popularidade da computação em nuvem e a onipresença da virtualização, o poder computacional dos servidores está se tornando menos relevante, e o *scaling out* geralmente é o método mais barato de dimensionamento de sites de acordo com nossas necessidades.

Ao desenvolver sites para o Node, você deve sempre considerar a possibilidade de fazer o *scaling out*. Mesmo se sua aplicação for pequena (poderia ser uma aplicação de intranet com público-alvo limitado) e possivelmente nunca precisar de *scaling out*, esse é um hábito que deve ser adotado. Afinal, seu próximo projeto pode ser o novo Twitter, e o *scaling out* será essencial. Felizmente, o suporte do Node ao *scaling out* é muito bom, e não será nenhum esforço criar sua aplicação com isso em mente.

O mais importante a se lembrar na construção de um site projetado para *scaling out* é a persistência. Se você está acostumado a confiar no armazenamento baseado em arquivos para manter a persistência, *desista*. Esse método o levará à loucura.

Minha primeira experiência com esse problema foi quase desastrosa. Um de nossos clientes estava promovendo um concurso baseado na web, e a aplicação web tinha sido projetada para informar aos 50 primeiros lugares que eles receberiam um prêmio. Com esse cliente específico, não conseguimos usar facilmente um banco de dados devido a algumas restrições empresariais de TI, logo, grande parte da persistência foi obtida com a gravação em arquivos simples (flat files). Dei prosseguimento como sempre faço, salvando cada entrada em um arquivo. Após o arquivo ter registrado 50 vencedores, ninguém mais seria informado de que havia vencido. O problema é que o servidor usava balanceamento de carga, portanto, metade das requisições foi servida por um servidor e a outra metade por outro. Um servidor informou a 50 pessoas que elas haviam vencido... e o outro servidor fez o mesmo. Felizmente, os prêmios eram baratos (cobertores de lã), e não algo caro como iPads, e o cliente resignou-se e entregou 100 prêmios em vez de 50 (me ofereci para pagar os 50 cobertores adicionais para compensar o erro, mas eles recusaram generosamente minha oferta).

A moral da história é que a menos que você tenha um sistema de arquivos que possa ser acessado por *todos* os seus servidores, não deve confiar no sistema de arquivos local para manter a persistência. As exceções são os dados somente de leitura, como os do logging, e os backups. Por exemplo, normalmente faço o backup de dados de envio de formulários em um arquivo simples local para o caso da conexão de banco de dados falhar. Quando há falha no banco de dados, é complicado ir até cada servidor e coletar os arquivos, mas pelo menos não há danos.

Scaling Out com clusters de aplicativo

O próprio Node dá suporte aos *clusters de aplicativo*, uma forma simples de scaling out com um único servidor. Com os clusters de aplicativo, você pode criar um servidor independente para cada núcleo (CPU) do sistema (termos mais servidores do que núcleos não melhorará o desempenho do aplicativo). Os clusters de aplicativo são bons por duas razões: em primeiro lugar, podem ajudar a melhorar o desempenho de um servidor específico (o hardware ou a máquina virtual), e, em segundo lugar, são uma maneira de testar o aplicativo com baixo overhead sob condições paralelas.

Daremos prosseguimento e adicionaremos o suporte de cluster ao nosso site. Embora seja comum que esse trabalho seja feito no arquivo principal da aplicação, criaremos um segundo arquivo que executará o aplicativo em um cluster, empregando o arquivo não pertencente ao cluster que temos usado. Para permitir isso, primeiro temos de fazer uma pequena modificação em *meadowlark.js* (consulte *ch12/01-server.js* no repositório fornecido para ver um exemplo simplificado):

```
function startServer(port) {
  app.listen(port, function() {
    console.log(`Express started in ${app.get('env')} ` +
      `mode on http://localhost:${port}` +
      `; press Ctrl-C to terminate.`)
  })
}

if(require.main === module) {
  // a aplicação é executada diretamente; inicia o servidor do aplicativo
```

```

    startServer(process.env.PORT || 3000)
  } else {
    // a aplicação é importada como um módulo com "require":
    // exporta a função que cria o servidor
    module.exports = startServer
  }

```

Você deve lembrar-se de que vimos no Capítulo 5 que, se usarmos `require.main === module`, isso significa que o script foi executado diretamente; caso contrário, ele será chamado com `require` a partir de outro script.

Em seguida, criaremos um novo script, *meadowlark-cluster.js* (consulte *ch12/01-cluster* no repositório fornecido para ver um exemplo simplificado):

```

const cluster = require('cluster')

function startWorker() {
  const worker = cluster.fork()
  console.log(`CLUSTER: Worker ${worker.id} started`)
}

if(cluster.isMaster){

  require('os').cpus().forEach(startWorker)

  // registra workers que se desconectaram; se um worker se desconectar,
  // ele deve ser encerrado, logo, esperamos o evento de encerramento
  // para gerar um novo worker para substituí-lo
  cluster.on('disconnect', worker => console.log(
    `CLUSTER: Worker ${worker.id} disconnected from the cluster.`
  ))

  // quando um worker fica inativo (é encerrado),
  // cria um worker para substituí-lo
  cluster.on('exit', (worker, code, signal) => {
    console.log(
      `CLUSTER: Worker ${worker.id} died with exit ` +
      `code ${code} (${signal})`
    )
    startWorker()
  })
}

```

```

    })

    } else {

        const port = process.env.PORT || 3000
        // inicia nosso aplicativo no worker; consulte meadowlark.js
        require('./meadowlark.js')(port)

    }

```

Quando esse código JavaScript for executado, ele estará no contexto do mestre (quando for executado diretamente, com `node meadowlark-cluster.js`) ou no contexto de um worker, quando o sistema de cluster do Node o executar. As propriedades `cluster.isMaster` e `cluster.isWorker` determinam em que contexto ocorrerá a execução. Quando executarmos esse script, sua execução ocorrerá no modo mestre, e iniciaremos um worker usando `cluster.fork` para cada CPU do sistema. Além disso, geraremos novamente qualquer worker inativo escutando os eventos `exit` emitidos pelos workers.

Para concluir, na cláusula `else`, manipulamos o caso do worker. Já que configuramos *meadowlark.js* para ser usado como um módulo, apenas o importamos e o chamamos imediatamente (lembre-se, o exportamos como uma função que inicia o servidor).

Agora iniciaremos seu novo servidor de cluster:

```
node meadowlark-cluster.js
```



Se você estiver usando virtualização (como o VirtualBox da Oracle), pode ser preciso configurar sua VM para que tenha várias CPUs. Por padrão, com frequência as máquinas virtuais têm uma única CPU.

Supondo que você esteja em um sistema multicore, deve ver alguns workers serem iniciados. Se quiser ver evidências de que diferentes workers estão manipulando diferentes requisições, adicione o middleware a seguir antes de suas rotas:

```

const cluster = require('cluster')

app.use((req, res, next) => {
  if(cluster.isWorker)

```

```
    console.log(`Worker ${cluster.worker.id} received request`)
    next()
  })
```

Agora você pode se conectar à sua aplicação com um navegador. Recarregue-o algumas vezes e veja se consegue obter um worker diferente no pool para cada requisição. (Talvez não consiga fazê-lo; o Node foi projetado para manipular um grande número de conexões, e talvez você não consiga forçá-lo suficientemente apenas recarregando seu navegador; posteriormente examinaremos o teste de estresse e você poderá ver melhor o cluster em ação).

Manipulando exceções não capturadas

No universo assíncrono do Node, as exceções não capturadas são particularmente preocupantes. Começaremos com um exemplo simples que não causa muitos problemas (recomendo que você acompanhe esses exemplos):

```
app.get('/fail', (req, res) => {
  throw new Error('Nope!')
})
```

Quando o Express executar os manipuladores de rota, ele os encapsulará em um bloco try/catch, logo, na verdade essa não é uma exceção não capturada. Ela não causará muitos problemas: o Express registrará a exceção no lado do servidor e o visitante verá um incômodo stack dump¹. No entanto, o servidor continuará estável, e outras requisições permanecerão sendo servidas corretamente. Se quiser fornecer uma página de erro “adequada”, crie um arquivo *views/500.handlebars* e adicione um manipulador de erro após todas as rotas:

```
app.use((err, req, res, next) => {
  console.error(err.message, err.stack)
  app.status(500).render('500')
})
```

É sempre boa prática fornecer uma página de erro personalizada; além de gerar uma aparência mais profissional para os usuários quando erros ocorrem, nos permite tomar medidas quando da sua ocorrência. Por exemplo, esse

manipulador de erro seria um bom local para notificarmos a equipe de desenvolvimento que um erro ocorreu. Infelizmente, isso só ajuda para exceções que o Express possa capturar. Tentaremos fazer algo pior:

```
app.get('/epic-fail', (req, res) => {  
  process.nextTick(() =>  
    throw new Error('Kaboom!')  
  )  
})
```

Vá em frente e execute esse código. O resultado será consideravelmente mais catastrófico: ele desativará totalmente o seu servidor! Além de não exibir uma mensagem de erro amigável para o usuário, agora o servidor está inativo e *nenhuma* requisição será atendida. Isso ocorreu porque `setTimeout` está sendo executado *assincronamente*; a execução da função com a exceção está sendo adiada até o Node ficar ocioso. O problema é que, quando o Node estiver ocioso e resolver executar a função, não terá mais um contexto sobre a requisição a partir do qual ela estava sendo servida, logo, não terá alternativa a não ser encerrar o servidor inteiro, porque agora ele se encontra em um estado indefinido. (O Node não tem como saber a finalidade da função ou quem é seu chamador, portanto, não pode mais supor que outras funções operarão corretamente).



`process.nextTick` é semelhante a `setTimeout` com o argumento 0, porém é mais eficiente. Estamos usando-o aqui para fins de demonstração; não é algo que usaríamos em código do lado do servidor. No entanto, nos próximos capítulos, lidaremos com muitas coisas que são executadas assincronamente, como o acesso ao banco de dados, ao sistema de arquivos e à rede, para citar algumas, e todas elas estão sujeitas a esse problema.

Há medidas que podemos tomar para manipular exceções não capturadas, mas, *se o Node não puder determinar a estabilidade da aplicação, você também não poderá*. Em outras palavras, se houver uma exceção não capturada, o único recurso é encerrar o servidor. O melhor que podemos fazer nesse caso é encerrá-lo da maneira menos problemática possível e usar um mecanismo de failover. O mecanismo de failover mais fácil de usar é o cluster. Se sua aplicação estiver operando no modo de cluster e um worker

ficar inativo, o mestre gerará outro worker para assumir seu lugar. (Não precisamos nem mesmo de vários workers; um cluster com um único worker é suficiente, embora o failover possa ser um pouco mais lento).

No entanto, como podemos executar o encerramento da maneira menos problemática possível quando confrontados com uma exceção não capturada? O mecanismo do Node que lida com essa questão é o evento `uncaughtException`. (O Node também tem um mecanismo chamado *domínios*, mas esse módulo tornou-se obsoleto e seu uso não é mais recomendado).

```
process.on('uncaughtException', err => {  
  console.error('UNCAUGHT EXCEPTION\n', err.stack);  
  // faça a limpeza que precisar aqui...feche  
  // conexões de banco de dados, etc.  
  process.exit(1)  
})
```

Não é realista esperar que a aplicação nunca apresentará exceções não capturadas, mas você deve ter um mecanismo para registrar a exceção e notificá-lo quando ela ocorrer, e deve levar isso a sério. Tente determinar por que ela ocorreu para poder corrigi-la. Serviços como o *Sentry* (<https://sentry.io>), o *Rollbar* (<https://rollbar.com>), o *Airbrake* (<https://airbrake.io/>) e o *New Relic* (<https://newrelic.com>) são uma ótima maneira de registrar esses tipos de erro para análise. Por exemplo, para usar o Sentry, primeiro você tem de se registrar para obter uma conta gratuita, momento em que receberá um nome de fonte de dados (DSN, data source name) e então poderá modificar seu manipulador de exceção:

```
const Sentry = require('@sentry/node')  
Sentry.init({ dsn: '** YOUR DSN GOES HERE **' })  
  
process.on('uncaughtException', err => {  
  // faça a limpeza que precisar aqui...feche  
  // conexões de banco de dados, etc.  
  Sentry.captureException(err)  
  process.exit(1)  
})
```

Scaling Out com vários servidores

Embora o scaling out com o uso de clusters possa melhorar o desempenho de um servidor individual, o que ocorreria se você precisasse de mais de um servidor? É aí que as coisas ficam um pouco mais complicadas. Para chegar a esse tipo de paralelismo, você precisa de um servidor *proxy*. (Com frequência ele é chamado de *reverse proxy* ou *forward-facing proxy* para diferenciá-lo dos proxies normalmente usados para acessar redes externas, mas acho esse jargão confuso e desnecessário, logo, vou chamá-lo apenas de proxy).

Duas opções muito populares são o *NGINX* (<https://www.nginx.com>) (pronuncia-se “engine X”) e o *HAProxy* (<http://www.haproxy.org>). Os servidores NGINX estão se expandindo como coelhos. Recentemente fiz uma análise de concorrência para minha empresa e descobri que mais de 80% de nossos concorrentes estavam usando o NGINX. Tanto o NGINX quanto o HAProxy são servidores proxy robustos e de alto desempenho e suportam as aplicações mais exigentes. (Se precisar de prova, considere que a Netflix, que é responsável por 15% de *todo o tráfego da internet*, usa o NGINX.)

Também há alguns servidores proxy menores baseados no Node, como o *node-http-proxy* (<http://bit.ly/34RWyNN>). Essa é uma ótima opção se suas necessidades forem modestas ou para desenvolvimento. Para produção, recomendo usar o NGINX ou o HAProxy (os dois são gratuitos, embora ofereçam suporte com o pagamento de uma taxa).

Instalar e configurar um proxy não faz parte do escopo deste livro, mas não é tão difícil quanto parece (principalmente se você usar o *node-http-proxy* ou outro proxy leve). Por enquanto, usar clusters nos dará alguma segurança de que nosso site está pronto para o scaling out.

Se você configurar um servidor proxy, certifique-se de informar ao Express que está usando um proxy e que ele é confiável:

```
app.enable('trust proxy')
```

Isso assegurará que `req.ip`, `req.protocol` e `req.secure` reflitam os detalhes da conexão entre o *cliente* e o *proxy*, e não entre o cliente e o aplicativo. Além disso, `req.ips` será um array indicando o IP do cliente original e os nomes ou endereços IP de qualquer proxy intermediário.

Monitorando seu site

Monitorar o site é uma das ações de QA mais importantes – e com frequência ignoradas – que você pode executar. A única coisa pior que estar acordado às 3 h da madrugada consertando um site danificado é ser acordado a essa hora pelo seu chefe porque o site está inativo (ou, ainda pior, chegar de manhã e saber que seu cliente perdeu 10.000 dólares em vendas porque o site ficou inativo a noite toda e ninguém notou).

Não há nada que você possa fazer com relação às falhas: elas são inevitáveis como a morte e os impostos. No entanto, se há algo que pode ser feito para convencer seu chefe e seus clientes de que você é um bom profissional é saber *sempre das falhas antes de elas ocorrerem*.

Monitores de uptime de terceiros

Um monitor de uptime sendo executado no servidor de seu site é tão eficaz quanto um detector de fumaça em uma casa na qual não viva ninguém. Ele pode ser capaz de capturar erros se uma página específica ficar inativa, mas, se o *servidor* inteiro parar de funcionar, talvez isso ocorra sem que seja enviado um simples SOS. É por essa razão que sua primeira linha de defesa devem ser os monitores de uptime de terceiros. O *UptimeRobot* (<http://uptimerobot.com>) é gratuito para até 50 monitores e é fácil de configurar. Os alertas podem seguir por email, SMS (mensagem de texto), pelo Twitter ou pelo Slack (entre outros). Você pode monitorar o código de retorno de uma única página (qualquer coisa diferente de 200 é considerado um erro) ou procurar a presença ou a ausência de uma palavra-chave na página. Lembre-se de que, se você usar um monitor de palavra-chave, isso pode afetar sua capacidade de análise (podemos excluir tráfego dos monitores de uptime na maioria dos serviços de análise).

Se suas necessidades forem mais sofisticadas, há outros serviços mais caros como o *Pingdom* (<http://pingdom.com>) e o *Site24x7* (<http://www.site24x7.com>).

Teste de estresse

O *teste de estresse* (ou *teste de carga*) foi projetado para nos dar alguma certeza de que o servidor funcionará com uma carga de centenas ou milhares de requisições simultâneas. Essa é outra área extensa que poderia ser assunto de um livro próprio: o teste de estresse pode ser arbitrariamente sofisticado, e o nível de complexidade desejado vai depender em grande parte da natureza do projeto. Se você tiver razões para achar que seu site será muito popular, pode ser melhor investir mais tempo no teste de estresse.

Adicionaremos um teste de estresse simples usando o *Artillery* (<https://artillery.io/>). Primeiro, instale o Artillery executando `npm install -g artillery`; em seguida, edite seu arquivo *package.json* e adicione o seguinte à seção `scripts`:

```
"scripts": {  
  "stress": "artillery quick --count 10 -n 20 http://localhost:3000/"  
}
```

Esse código simulará 10 “usuários virtuais” (`--count 10`), cada um deles enviando 20 requisições (`-n 20`) para o seu servidor.

Certifique-se se sua aplicação está sendo executada (em uma janela de terminal separada, por exemplo) e depois execute `npm run stress`. Você verá estatísticas como estas:

```
Started phase 0, duration: 1s @ 16:43:37(-0700) 2019-04-14  
Report @ 16:43:38(-0700) 2019-04-14  
Elapsed time: 1 second  
Scenarios launched: 10  
Scenarios completed: 10  
Requests completed: 200  
RPS sent: 147.06  
Request latency:  
  min: 1.8  
  max: 10.3  
  median: 2.5  
  p95: 4.2  
  p99: 5.4  
Codes:  
  200: 200
```

All virtual users finished
Summary report @ 16:43:38(-0700) 2019-04-14
Scenarios launched: 10
Scenarios completed: 10
Requests completed: 200
RPS sent: 145.99
Request latency:
 min: 1.8
 max: 10.3
 median: 2.5
 p95: 4.2
 p99: 5.4
Scenario counts:
 0: 10 (100%)
Codes:
 200: 200

Esse teste foi executado em meu laptop de desenvolvimento. É possível ver que o Express não levou mais do que 10,3 milissegundos para servir as requisições, e 99% delas foram servidas em menos de 5,4 milissegundos. Não posso oferecer uma orientação concreta sobre o tipo de números que você deve procurar, mas, para assegurar rapidez na aplicação, deve tentar ter tempos totais de conexão abaixo de 50 milissegundos. (Não se esqueça de que esse é apenas o tempo que o servidor demora para distribuir os dados para o cliente; este ainda tem de renderizá-los, o que leva tempo, logo, quanto menor for o tempo que você gastar transmitindo os dados, melhor).

Se você executar o teste de estresse em sua aplicação regularmente e avaliá-lo, poderá reconhecer problemas. Se tiver acabado de criar um recurso e descobrir que seus tempos de conexão triplicaram, deve fazer algum ajuste de desempenho em seu novo recurso!

Conclusão

Espero que este capítulo tenha lhe dado uma ideia das coisas que você deve considerar quando estiver perto de lançar a aplicação. Há muitos detalhes que entram em uma aplicação na produção, e, embora você não possa prever tudo o que ocorrerá no lançamento, quanto maior for o número de ocorrências que

conseguir antecipar, mais sucesso terá. Parafraseando Louis Pasteur, a sorte favorece os preparados.

1 N.T.: Um stack dump é uma listagem de tudo que está sendo executado no sistema quando um programa trava.

CAPÍTULO 13

Persistência

Todos os sites e aplicações web, exceto os mais simples, requerem algum tipo de *persistência*; isto é, alguma maneira de armazenar dados que seja mais permanente que a memória volátil para que eles sobrevivam a falhas no servidor, falta de energia, upgrades e relocações. Neste capítulo, discutiremos as opções disponíveis para a persistência e demonstraremos tanto os bancos de dados de documentos quanto os relacionais. No entanto, antes de abordar os bancos de dados, começaremos com o tipo mais básico de persistência: a persistência de sistema de arquivos.

Persistência de sistema de arquivos

Uma maneira de obtermos persistência é simplesmente salvando os dados nos chamados arquivos simples (*simples* porque os arquivos não têm uma estrutura inerente; trata-se apenas de uma sequência de bytes). O Node torna a persistência de sistema de arquivos possível por intermédio do módulo fs (filesystem).

A persistência de sistema de arquivos apresenta algumas desvantagens. Especificamente, ela não é boa para a escalabilidade. No momento em que você precisar de um servidor para atender demandas de tráfego, surgirão problemas com a persistência de sistema de arquivos, a menos que todos os seus servidores tenham acesso a um sistema de arquivos compartilhado. Além disso, já que os arquivos simples não têm estrutura inerente, o encargo de localizar, classificar e filtrar dados é transferido para a aplicação. Logo, você deve dar preferência aos bancos de dados, e não aos sistemas de arquivos para armazenar dados. A única exceção é o armazenamento de arquivos binários, como imagens, arquivos de áudio, ou vídeos. Muitos bancos de dados conseguem manipular esse tipo de dados, mas raramente o

fazem com mais eficiência que um sistema de arquivos (embora informações *sobre* os arquivos binários geralmente sejam armazenadas em um banco de dados para permitir a busca, a classificação e a filtragem).

Se você precisar armazenar dados binários, lembre-se de que o armazenamento em sistema de arquivos também apresenta o problema de não dar um bom suporte à escalabilidade. Se sua solução de hospedagem não tiver acesso a um sistema de arquivos compartilhado (que é o que geralmente ocorre), você deve considerar armazenar os arquivos binários em um banco de dados (o que costuma demandar alguma configuração para o banco de dados não ser interrompido) ou em um serviço de armazenamento baseado em nuvem, como o Amazon S3 ou o Microsoft Azure Storage.

Feitas as devidas advertências, examinaremos o suporte de sistema de arquivos do Node. Voltaremos ao concurso de fotos de férias do Capítulo 8. No arquivo de nossa aplicação, forneceremos o manipulador que processa esse formulário (*ch13/00-mongodb/lib/handlers.js* no repositório fornecido):

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

// cria diretório para armazenar fotos de férias (se ele ainda não existir)
const dataDir = pathUtils.resolve(__dirname, '..', 'data')
const vacationPhotosDir = pathUtils.join(dataDir, 'vacation-photos')
if(!fs.existsSync(dataDir)) fs.mkdirSync(dataDir)
if(!fs.existsSync(vacationPhotosDir)) fs.mkdirSync(vacationPhotosDir)

function saveContestEntry(contestName, email, year, month, photoPath) {
  // A SER CONCLUÍDO...essa parte virá depois
}

// precisaremos dessas versões de funções do fs baseadas
// em promessas posteriormente
const { promisify } = require('util')
const mkdir = promisify(fs.mkdir)
const rename = promisify(fs.rename)

exports.api.vacationPhotoContest = async (req, res, fields, files) => {
  const photo = files.photo[0]
```

```
const dir = vacationPhotosDir + '/' + Date.now()
const path = dir + '/' + photo.originalFilename
await mkdir(dir)
await rename(photo.path, path)
saveContestEntry('vacation-photo', fields.email,
  req.params.year, req.params.month, path)
res.send({ result: 'success' })
}
```

Há muita coisa acontecendo aqui, logo, analisaremos por partes. Primeiro criamos um diretório para armazenar os arquivos enviados por upload (caso ele ainda não exista). É aconselhável adicionar o diretório *data* ao arquivo *.gitignore* para você não fazer o commit acidental dos arquivos. Conforme vimos no Capítulo 8 estamos manipulando o upload em *meadowlark.js* e chamando nosso manipulador com os arquivos já decodificados. O que obteremos será um objeto (*files*) contendo as informações desses arquivos. Já que queremos evitar colisões, não podemos usar o nome do arquivo que o usuário enviou (para o caso, por exemplo, de dois usuários fazerem o upload de *portland.jpg*). Para evitar esse problema, criamos um diretório exclusivo baseado no timestamp; é improvável que dois usuários façam o upload de *portland.jpg* no mesmo milissegundo! Em seguida, mudamos (movemos) o nome do arquivo enviado por upload (nosso processador de arquivos fornecerá um nome temporário para ele, que pode ser acessado na propriedade *path*) para o nome construído.

Para concluir, precisamos de alguma maneira de associar os arquivos que os usuários enviaram por upload aos seus endereços de email (e ao mês e ano do envio). Poderíamos codificar essas informações nos nomes de arquivo ou diretório, mas daremos preferência a armazená-las em um banco de dados. Como ainda não aprendemos a fazer isso, encapsularemos essa funcionalidade na função *vacationPhotoContest* e a terminaremos posteriormente neste capítulo.



Em geral, não devemos nunca confiar em algo que o usuário envie por upload porque pode ser um vetor para o site ser atacado. Por exemplo, um usuário malicioso poderia pegar facilmente um executável prejudicial, renomeá-lo com uma extensão *.jpg* e enviá-lo por upload

como a primeira etapa de um ataque (esperando encontrar alguma maneira de executá-lo em um momento posterior). Da mesma forma, estamos nos arriscando aqui ao nomear o arquivo usando a propriedade `name` fornecida pelo navegador; alguém também poderia explorar isso inserindo caracteres especiais no nome do arquivo. Para tornar esse código totalmente seguro, teríamos de dar ao arquivo um nome aleatório, pegando apenas a extensão (certificando-nos de que ela seja composta somente de caracteres alfanuméricos).

Ainda que a persistência de sistema de arquivos apresente desvantagens, ela é usada com frequência para o armazenamento intermediário de arquivos, e é útil saber como usar a biblioteca `filesystem` do Node. No entanto, para resolver as deficiências do armazenamento em sistema de arquivos, voltaremos nossa atenção para a persistência de nuvem.

Persistência de nuvem

O armazenamento em nuvem está se tornando cada vez mais popular e recomendo que você se beneficie de um desses baratos e robustos serviços.

Antes de usarmos os serviços de nuvem, há trabalho a fazer. É claro que temos de criar uma conta, mas também precisamos entender como a aplicação faz a autenticação no serviço de nuvem, e é útil conhecer a terminologia básica (por exemplo, a AWS chama seu mecanismo de armazenamento de arquivos de *bucket*, enquanto o Azure o chama de *container*). Não faz parte do escopo deste livro detalharmos todas essas informações, e elas têm uma boa documentação:

- *AWS: Getting Started in Node.js* (<https://amzn.to/2CCYk9s>)
- *Azure for JavaScript and Node.js Developers* (<http://bit.ly/2NEkTku>)

A boa notícia é que, quando deixamos para trás essa configuração inicial, é muito fácil usar a persistência de nuvem. Aqui está um exemplo de como é fácil salvar um arquivo em uma conta do Amazon S3:

```
const filename = 'customerUpload.jpg'
```

```
s3.putObject({  
  Bucket: 'uploads',
```

```
Key: filename,  
Body: fs.readFileSync(__dirname + '/tmp/' + filename),  
})
```

Consulte a *documentação do AWS SDK* (<https://amzn.to/2O3e1MA>) para obter mais informações.

Veja um exemplo de como fazer o mesmo com o Microsoft Azure:

```
const filename = 'customerUpload.jpg'  
  
const blobService = azure.createBlobService()  
blobService.createBlockBlobFromFile('uploads', filename, __dirname +  
  '/tmp/' + filename)
```

Examine a *documentação do Microsoft Azure* (<http://bit.ly/2Kd3rRK>) para ver mais detalhes.

Agora que conhecemos algumas técnicas de armazenamento de arquivos, consideraremos o armazenamento de dados estruturados com bancos de dados.

Persistência de banco de dados

Todos os sites e aplicações web, exceto os mais simples, precisam de um banco de dados. Mesmo se grande parte de seus dados for binária e você estiver usando um sistema de arquivos compartilhado ou o armazenamento em nuvem, provavelmente precisará de um banco de dados para ajudar a catalogar esses dados binários.

Tradicionalmente, o termo *banco de dados* tem sido usado para abreviar a expressão *sistema de gerenciamento de bancos de dados relacionais* (RDBMS, relational database management system). Os bancos de dados relacionais, como o Oracle, MySQL, PostgreSQL ou SQL Server, são baseados em décadas de pesquisa e na teoria formal dos bancos de dados. É uma tecnologia que a essa altura já está bem madura, e o poder desses bancos de dados é inquestionável. No entanto, agora podemos nos dar ao luxo de expandir nossas ideias do que compõe um banco de dados. Os bancos de dados NoSQL tornaram-se populares nos últimos anos e estão desafiando o status quo do armazenamento de dados na internet.

Seria tolo alegar que os bancos de dados NoSQL são melhores que os relacionais, mas eles apresentam certas vantagens (e vice-versa). Embora seja muito fácil integrar um banco de dados relacional a aplicativos Node, não há um banco de dados NoSQL que não pareça ter sido projetado para o Node.

Os dois tipos de bancos de dados NoSQL mais populares são os *bancos de dados de documentos* e os bancos de dados de *chave-valor*. Os bancos de dados de documentos se sobressaem no armazenamento de objetos, o que os torna uma opção natural para o Node e o JavaScript. Os bancos de dados de chave-valor, como o nome sugere, são extremamente simples e constituem uma ótima opção para aplicações com esquemas de dados que sejam facilmente mapeados para pares de chave-valor.

Acredito que os bancos de dados de documentos representem o equilíbrio ótimo entre as restrições dos bancos de dados relacionais e a simplicidade dos bancos de dados de chave-valor, portanto, vamos usá-lo em nosso primeiro exemplo. O MongoDB é o principal banco de dados de documentos, é robusto e já está consolidado.

Para nosso segundo exemplo, usaremos o PostgreSQL, um RDBMS open source popular e robusto.

Observação sobre o desempenho

A simplicidade dos bancos de dados NoSQL é uma faca de dois gumes. Projetar cuidadosamente um banco de dados relacional pode ser uma tarefa complicada, mas o benefício obtido com esse planejamento cuidadoso é um banco de dados que oferece um ótimo desempenho. Não se iluda pensando que, já que geralmente os bancos de dados NoSQL são mais simples, não é preciso astúcia e conhecimento para ajustá-los para que forneçam o melhor nível de desempenho.

Tradicionalmente, os bancos de dados relacionais têm confiado em suas rígidas estruturas de dados e em décadas de pesquisas de otimização para oferecer um alto desempenho. Por outro lado, os bancos de dados NoSQL abraçaram a natureza distribuída da internet e, como o Node, deram ênfase à concorrência para melhorar o desempenho (os bancos de dados relacionais também dão suporte à concorrência, mas ela costuma ser reservada para

aplicações mais exigentes).

O planejamento do desempenho e da escalabilidade do banco de dados é um tópico extenso e complexo que não faz parte do escopo deste livro. Se sua aplicação demandar um alto nível de desempenho do banco de dados, recomendo que você comece pela leitura de *MongoDB: The Definitive Guide* (O'Reilly) de Kristina Chodorow e Michael Dirolf.

Abstraindo a camada de banco de dados

Neste livro, implementaremos os mesmos recursos e demonstraremos como fazê-lo com dois bancos de dados (que não são apenas dois bancos de dados, e sim duas arquiteturas de banco de dados significativamente diferentes). Embora o objetivo do livro seja abordar duas opções populares de arquitetura de banco de dados, na verdade estamos refletindo um cenário do mundo real: trocar um componente importante no meio de um projeto de aplicação web. Isso pode ocorrer por muitas razões. Geralmente se resume a descobrir que uma tecnologia diferente será mais barata ou que ela lhe permitirá implementar os recursos necessários com mais rapidez.

Sempre que possível, é útil *abstrair* as opções de tecnologia, o que significa criar algum tipo de camada de API para generalizar as opções de tecnologia subjacentes. Se feito corretamente, isso reduz o custo da troca do componente em questão. No entanto, há um preço a pagar: a camada de abstração é mais uma coisa que precisará de desenvolvimento e manutenção.

Felizmente, nossa camada de abstração será muito pequena, já que estamos suportando apenas alguns recursos para os fins deste livro. Por enquanto, os recursos serão os seguintes:

- Retornar uma lista dos pacotes de férias a partir do banco de dados
- Armazenar o endereço de email dos usuários que quiserem ser notificados quando for época de certos pacotes de férias

Embora pareça muito simples, há vários detalhes aqui. Qual é a aparência de um pacote de férias? Vamos acessar sempre todos os pacotes de férias no banco de dados ou queremos poder filtrá-los ou organizá-los em páginas? Como identificar os pacotes? E assim por diante.

Para os fins deste livro, teremos uma camada de abstração simples. Vamos armazená-la em um arquivo chamado *db.js* que exportará dois métodos para os quais começaremos fornecendo apenas implementações fictícias:

```
module.exports = {
  getVacations: async (options = {}) => {
    // simularemos alguns dados de férias:
    const vacations = [
      {
        name: 'Hood River Day Trip',
        slug: 'hood-river-day-trip',
        category: 'Day Trip',
        sku: 'HR199',
        description: 'Spend a day sailing on the Columbia and ' +
          'enjoying craft beers in Hood River!',
        location: {
          // usaremos essa parte para a geocodificação
          // posteriormente no livro
          search: 'Hood River, Oregon, USA',
        },
        price: 99.95,
        tags: ['day trip', 'hood river', 'sailing', 'windsurfing', 'breweries'],
        inSeason: true,
        maximumGuests: 16,
        available: true,
        packagesSold: 0,
      }
    ]
    // se a opção "available" for especificada, somente as
    // férias correspondentes serão retornadas
    if(options.available !== undefined)
      return vacations.filter(({ available }) => available === options.available)
    return vacations
  },
  addVacationInSeasonListener: async (email, sku) => {
    // vamos apenas fingir que fizemos isso...já que essa é
    // uma função async, uma nova promessa será retornada automaticamente
    // e será resolvida como undefined
  },
}
```

}

Esse código define a expectativa de como será a aparência de nossa implementação de banco de dados na aplicação... só precisamos fazer com que os bancos de dados estejam de acordo com essa interface. Observe que estamos introduzindo o conceito de “disponibilidade” do pacote de férias; fizemos isso para poder desativar os pacotes de férias temporariamente em vez de excluí-los do banco de dados. Um exemplo dessa situação seria um alojamento do tipo bed and breakfast¹ que nos avisasse que estará fechado por vários meses para reforma. Estamos mantendo esse caso separado do conceito de “estar na época” porque podemos querer listar os pacotes de férias que estejam fora de época no site já que as pessoas gostam de se planejar com antecedência.

Também incluímos algumas informações de “localização” muito genéricas; seremos mais específicos com relação a isso no Capítulo 19.

Agora que temos uma base de abstração para nossa camada de banco de dados, examinaremos como podemos implementar o armazenamento de bando de dados com o MongoDB.

Instalando o MongoDB

As dificuldades envolvidas na instalação de uma instância do MongoDB variam com o sistema operacional. Logo, evitaremos esse problema usando um excelente serviço de hospedagem gratuito do MongoDB, o mLab.



O mLab não é o único serviço de MongoDB disponível. Atualmente a própria empresa MongoDB está oferecendo hospedagem em banco de dados gratuita e de baixo custo com seu produto *MongoDB Atlas* (<https://www.mongodb.com>). No entanto, as contas gratuitas não são recomendadas para fins de produção. Tanto o mLab quanto o MongoDB Atlas oferecem contas prontas para serem usadas na produção, logo, você deve examinar seus preços antes de escolher. Será menos complicado se você continuar usando o mesmo serviço de hospedagem ao fazer a passagem para a produção.

É fácil começar a usar o mLab. Simplesmente acesse <https://mlab.com> e clique em Sign Up. Preencha o formulário de registro, faça login e será

levado para a tela inicial. Em Databases, você verá “no databases at this time”. Clique em “Create new” e será levado a uma página com algumas opções para o novo banco de dados. A primeira coisa que deve ser selecionada é um provedor de nuvem. Para uma conta gratuita (sandbox), essa opção é em grande parte irrelevante, embora você deva procurar um data center próximo (no entanto, nem todo data center oferece contas sandbox). Selecione SANDBOX e escolha uma região. Em seguida, selecione um nome de banco de dados e clique em Submit Order (mesmo sendo gratuito, é um pedido de aquisição!). Você será levado de volta para a lista de bancos de dados, e, após alguns segundos, seu banco de dados estará disponível para uso.

Ter um banco de dados instalado é apenas metade da empreitada. Agora temos de acessá-lo com o Node, e é aí que o Mongoose entra em cena.

Mongoose

Embora haja um driver de baixo nível disponível para o *MongoDB* (<http://bit.ly/2Kfw0hE>), provavelmente você vai querer usar um mapeador objeto-documento (ODM, object document mapper). O ODM mais popular para o MongoDB é o *Mongoose*.

Uma das vantagens do JavaScript é que seu modelo de objeto é extremamente flexível. Se você quiser adicionar uma propriedade ou um método a um objeto, basta fazê-lo, sem precisar se preocupar com a modificação de uma classe. Infelizmente, esse tipo de flexibilidade espontânea pode ter um impacto negativo sobre os bancos de dados porque eles podem tornar-se fragmentados e difíceis de otimizar. O Mongoose tenta obter um equilíbrio introduzindo *esquemas* e *modelos* (combinados, os esquemas e modelos são semelhantes às classes da programação orientada a objetos tradicional). Os esquemas são flexíveis, mas mesmo assim fornecem alguma estrutura necessária para o banco de dados.

Antes de começar, temos de instalar o módulo Mongoose:

```
npm install mongoose
```

Em seguida, adicionaremos nossas credenciais de banco de dados ao arquivo *.credentials.development.json*:

```
"mongo": {  
  "connectionString": "your_dev_connection_string"  
}
```

Você encontrará a string da conexão na página de bancos de dados do mLab. Na tela inicial, clique no banco de dados apropriado. Você verá uma caixa com a URI da conexão do MongoDB (ela começa com *mongodb://*). Também será preciso criar um usuário para o banco de dados. Para criá-lo, clique em Users e depois em “Add database user”.

Poderíamos estabelecer um segundo conjunto de credenciais para a produção criando um arquivo *.credentials.production.js* e usando `NODE_ENV=production`; é recomendável que você o faça quando chegar a hora de operar online!

Agora que terminamos de configurar, estabeleceremos uma conexão com o banco de dados e faremos algo útil!

Conexões de banco de dados com o Mongoose

Começaremos criando uma conexão com nosso banco de dados. Inseriremos o código de inicialização do banco de dados em *db.js*, junto com a API fictícia que criamos anteriormente (*ch13/00-mongodb/db.js* no repositório fornecido):

```
const mongoose = require('mongoose')  
const { connectionString } = credentials.mongo  
if(!connectionString) {  
  console.error('MongoDB connection string missing!')  
  process.exit(1)  
}  
mongoose.connect(connectionString)  
const db = mongoose.connection  
db.on('error' err => {  
  console.error('MongoDB error: ' + err.message)  
  process.exit(1)  
})  
db.once('open', () => console.log('MongoDB connection established'))
```



```

module.exports = {
  getVacations: async () => {
    //...retorna dados de férias fictícios
  },
  addVacationInSeasonListener: async (email, sku) => {
    //...não faz nada
  },
}

```

Qualquer arquivo que precisar acessar o banco de dados pode simplesmente importar *db.js*. No entanto, queremos que a inicialização ocorra imediatamente, antes de precisarmos da API, logo, faremos a importação a partir de *meadowlark.js* (em que não precisamos fazer nada com a API):

```
require('./db')
```

Agora que estamos nos conectando com o banco de dados, é hora de considerar como estruturaremos os dados que estamos transferindo de e para ele.

Criando esquemas e modelos

Criaremos um banco de dados de pacotes de férias para a Meadowlark Travel. Começaremos definindo um esquema e criando um modelo a partir dele. Crie o arquivo *models/vacation.js* (*ch13/00-mongodb/models/vacation.js* no repositório fornecido):

```

const mongoose = require('mongoose')

const vacationSchema = mongoose.Schema({
  name: String,
  slug: String,
  category: String,
  sku: String,
  description: String,
  location: {
    search: String,
    coordinates: {
      lat: Number,
      lng: Number,
    },
  },
})

```

```
},  
price: Number,  
tags: [String],  
inSeason: Boolean,  
available: Boolean,  
requiresWaiver: Boolean,  
maximumGuests: Number,  
notes: String,  
packagesSold: Number,  
}))
```

```
const Vacation = mongoose.model('Vacation', vacationSchema)  
module.exports = Vacation
```

Esse código declara as propriedades que compõem o modelo de férias e seu tipo. Como você pode ver, há várias propriedades que são strings, algumas são propriedades numéricas, duas são propriedades booleanas e há um array de strings (representado por [String]). Nesse momento, também podemos definir métodos em nosso esquema. Cada produto tem uma unidade de manutenção de estoque (SKU, stock keeping unit); ainda que não consideremos férias como “itens de estoque”, o conceito de SKU é padrão em contabilidade, mesmo se o que estiver sendo vendido não forem mercadorias tangíveis.

Tendo definido o esquema, criaremos um modelo usando `mongoose.model`: nesse caso, `Vacation` é semelhante a uma classe da programação tradicional orientada a objetos. Lembre-se de que precisamos definir nossos métodos antes de criar o modelo.



Devido à natureza dos números de ponto flutuante, você deve sempre tomar cuidado com cálculos financeiros em JavaScript. Poderíamos armazenar nossos preços em cents em vez de em dólares, mas isso não resolveria o problema. Para os modestos fins de nosso site de viagens, não nos preocuparemos com essa questão, mas, se sua aplicação envolver quantias financeiras muito altas ou muito baixas (por exemplo, cents fracionários provenientes de juros ou de vendas volumosas), você deve considerar o uso de uma biblioteca como o *currency.js* (<https://currency.js.org>) ou o *decimal.js-light* (<http://bit.ly/2X6kbQ5>). O objeto

interno *BigInt* (<https://mzl.la/2Xhs45r>) do JavaScript, que está disponível a partir do Node 10 (com suporte de navegador limitado quando este texto foi escrito), também pode ser usado para essa finalidade.

Estamos exportando o objeto de modelo *Vacation* criado pelo Mongoose. Embora pudéssemos usar esse modelo diretamente, isso prejudicaria nosso esforço de fornecer uma camada de abstração de banco de dados. Logo, preferimos importá-lo do arquivo *db.js* e deixar a aplicação usar seus métodos. Adicione o modelo *Vacation* a *db.js*:

```
const Vacation = require('./models/vacation')
```

Agora todas as estruturas estão definidas, mas nosso banco de dados não é muito interessante porque não há nada nele. Vamos torná-lo útil inserindo alguns dados.

Fornecendo os dados iniciais

Ainda não temos nenhum pacote de férias em nosso banco de dados, logo, adicionaremos alguns. Se quiser, você pode usar outra maneira de gerenciar os produtos, mas, para os fins deste livro, faremos isso em código (*ch13/00-mongodb/db.js* no repositório fornecido):

```
Vacation.find((err, vacations) => {
  if(err) return console.error(err)
  if(vacations.length) return

  new Vacation({
    name: 'Hood River Day Trip',
    slug: 'hood-river-day-trip',
    category: 'Day Trip',
    sku: 'HR199',
    description: 'Spend a day sailing on the Columbia and ' +
      'enjoying craft beers in Hood River!',
    location: {
      search: 'Hood River, Oregon, USA',
    },
    price: 99.95,
    tags: ['day trip', 'hood river', 'sailing', 'windsurfing', 'breweries'],
    inSeason: true,
```

```
    maximumGuests: 16,  
    available: true,  
    packagesSold: 0,  
  }).save()
```

```
new Vacation({  
  name: 'Oregon Coast Getaway',  
  slug: 'oregon-coast-getaway',  
  category: 'Weekend Getaway',  
  sku: 'OC39',  
  description: 'Enjoy the ocean air and quaint coastal towns!',  
  location: {  
    search: 'Cannon Beach, Oregon, USA',  
  },  
  price: 269.95,  
  tags: ['weekend getaway', 'oregon coast', 'beachcombing'],  
  inSeason: false,  
  maximumGuests: 8,  
  available: true,  
  packagesSold: 0,  
}).save()
```

```
new Vacation({  
  name: 'Rock Climbing in Bend',  
  slug: 'rock-climbing-in-bend',  
  category: 'Adventure',  
  sku: 'B99',  
  description: 'Experience the thrill of climbing in the high desert.',  
  location: {  
    search: 'Bend, Oregon, USA',  
  },  
  price: 289.95,  
  tags: ['weekend getaway', 'bend', 'high desert', 'rock climbing'],  
  inSeason: true,  
  requiresWaiver: true,  
  maximumGuests: 4,  
  available: false,  
  packagesSold: 0,  
  notes: 'The tour guide is currently recovering from a skiing accident.',  
}).save()
```

```
}).save()  
})
```

Dois métodos do Mongoose estão sendo usados aqui. O primeiro, `find`, faz o que seu nome sugere. Nesse caso, ele está procurando todas as instâncias de `Vacation` no banco de dados e chamando o `callback` com essa lista. Estamos fazendo isso porque não queremos precisar adicionar novamente os pacotes de férias: se já houver pacotes de férias no banco de dados, ele terá sido alimentado, e só temos de seguir em frente. No entanto, na primeira vez que esse código for executado, `find` retornará uma lista vazia, logo, criamos dois pacotes de férias e chamamos o método `save` para salvar esses novos objetos no banco de dados.

Agora que os dados estão *no* banco de dados, é hora de recuperá-los!

Recuperando dados

Já vimos o método `find`, que usaremos para exibir uma lista de pacotes de férias. Porém, dessa vez passaremos uma opção para `find` que filtrará os dados. Especificamente, só queremos exibir pacotes de férias que estejam disponíveis atualmente.

Crie uma view para a página de produtos, `views/vacations.handlebars`:

```
<h1>Vacations</h1>  
{{#each vacations}}  
  <div class="vacation">  
    <h3>{{name}}</h3>  
    <p>{{description}}</p>  
    {{#if inSeason}}  
      <span class="price">{{price}}</span>  
      <a href="/cart/add?sku={{sku}}" class="btn btn-default">Buy Now!</a>  
    {{else}}  
      <span class="outOfSeason">We're sorry, this vacation is currently  
      not in season.  
      {{! The "notify me when this vacation is in season"  
      page will be our next task. }}  
      <a href="/notify-me-when-in-season?sku={{sku}}">Notify me when  
      this vacation is in season.</a>  
    {{/if}}  
  </div>  
{{/each}}
```

```
</div>
{{/each}}
```

Agora podemos criar manipuladores de rota e conectar tudo. Em *lib/handlers.js* (não se esqueça de importar *../db*), criaremos o manipulador a seguir:

```
exports.listVacations = async (req, res) => {
  const vacations = await db.getVacations({ available: true })
  const context = {
    vacations: vacations.map(vacation => ({
      sku: vacation.sku,
      name: vacation.name,
      description: vacation.description,
      price: '$' + vacation.price.toFixed(2),
      inSeason: vacation.inSeason,
    }))
  }
  res.render('vacations', context)
}
```

Adicionaremos uma rota que chame o manipulador em *meadowlark.js*:

```
app.get('/vacations', handlers.listVacations)
```

Se você executar esse exemplo, verá um único pacote de férias em nossa implementação de banco de dados fictícia. Isso ocorrerá porque inicializamos o banco de dados e o alimentamos, mas não substituímos a implementação fictícia por uma real. Vamos fazê-lo agora. Abra *db.js* e modifique *getVacations*:

```
module.exports = {
  getVacations: async (options = {}) => Vacation.find(options),
  addVacationInSeasonListener: async (email, sku) => {
    //...
  },
}
```

Foi fácil! Só tivemos de mexer em uma linha. Foi tão fácil em parte porque o Mongoose está fazendo o trabalho pesado para nós, e a maneira como projetamos nossa API é semelhante a como ele funciona. Quando fizermos a adaptação para o PostgreSQL posteriormente, você verá que será preciso um

pouco mais de trabalho.



O leitor astuto pode ficar preocupado, já que nossa camada de abstração de banco de dados não está se esforçando muito para “manter” seu objetivo de permanecer neutra quanto à tecnologia. Por exemplo, um desenvolvedor poderia ler esse código e ver que é possível passar qualquer opção do Mongoose para o modelo de férias, caso em que a aplicação usaria recursos específicos do Mongoose, tornando mais difícil a troca de banco de dados. Poderíamos tomar algumas medidas para impedir isso. Em vez de passar coisas para o Mongoose, poderíamos procurar opções específicas e manipulá-las explicitamente, deixando claro que qualquer implementação teria de fornecer essas opções. No entanto, nesse exemplo deixaremos essa solução de fora e manteremos o código simples.

Grande parte disso tudo deve parecer familiar, mas algumas coisas podem surpreendê-lo. Por exemplo, a maneira como estamos manipulando o contexto da view para a listagem de férias pode parecer estranha. Por que mapeamos os produtos retornados do banco de dados para um objeto quase idêntico? Uma razão é que queremos exibir o preço formatado de maneira clara, logo, temos de convertê-lo para uma string formatada.

Poderíamos digitar menos fazendo o seguinte:

```
const context = {  
  vacations: products.map(vacations => {  
    vacation.price = '$' + vacation.price.toFixed(2)  
    return vacation  
  })  
}
```

Isso certamente nos pouparia algumas linhas de código, mas, por experiência própria, há boas razões para não passarmos objetos de banco de dados não mapeados diretamente para as views. A view recebe várias propriedades que podem ser desnecessárias, muitas vezes em formatos que são incompatíveis com ela. Até agora temos um exemplo muito simples, mas, quando ele ficar mais complicado, provavelmente você vai querer aplicar uma personalização ainda maior aos dados passados para a view. Também facilita expor acidentalmente informações confidenciais ou que possam comprometer a

segurança do site. Logo, recomendo mapear os dados que forem retornados do banco de dados e passar somente o que for necessário para a view (transformando quando preciso, como fizemos com price).



Em algumas variações da arquitetura MVC, um terceiro componente chamado *view model* é introduzido. Basicamente, um view model refina e transforma um modelo (ou modelos) para que fique mais apropriado para exibição em uma view. O que estamos fazendo aqui é criando um view model dinamicamente.

Já avançamos bastante. Estamos usando com sucesso um banco de dados para armazenar informações sobre nossas férias. No entanto, os bancos de dados não seriam muito úteis se não pudéssemos atualizá-los. Voltaremos nossa atenção para esse aspecto da interação com os bancos de dados.

Adicionando dados

Vimos como adicionar dados (adicionamos dados quando fornecemos a coleção de pacotes de férias) e como atualizá-los (atualizamos a contagem de pacotes vendidos quando reservarmos um pacote de férias), mas examinaremos um cenário um pouco mais complicado que realce a flexibilidade dos bancos de dados de documentos.

Quando não for a época certa de um pacote de férias, exibiremos um link convidando o cliente a ser notificado quando a época chegar novamente. Vamos incluir essa funcionalidade. Primeiro, criaremos o esquema e o modelo (*models/vacationInSeasonListener.js*):

```
const mongoose = require('mongoose')

const vacationInSeasonListenerSchema = mongoose.Schema({
  email: String,
  skus: [String],
})
const VacationInSeasonListener = mongoose.model('VacationInSeasonListener',
  vacationInSeasonListenerSchema)

module.exports = VacationInSeasonListener
```

Em seguida, criaremos nossa view, *views/notify-me-when-in-*

season.handlebars:

```
<div class="formContainer">
  <form class="form-horizontal newsletterForm" role="form"
    action="/notify-me-when-in-season" method="POST">
    <input type="hidden" name="sku" value="{{sku}}">
    <div class="form-group">
      <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
      <div class="col-sm-4">
        <input type="email" class="form-control" required
          id="fieldEmail" name="email">
      </div>
    </div>
    <div class="form-group">
      <div class="col-sm-offset-2 col-sm-4">
        <button type="submit" class="btn btn-default">Submit</button>
      </div>
    </div>
  </form>
</div>
```

Depois temos os manipuladores de rota:

```
exports.notifyWhenInSeasonForm = (req, res) =>
  res.render('notify-me-when-in-season', { sku: req.query.sku })

exports.notifyWhenInSeasonProcess = (req, res) => {
  const { email, sku } = req.body
  await db.addVacationInSeasonListener(email, sku)
  return res.redirect(303, '/vacations')
}
```

Para concluir, adicionaremos uma implementação real a *db.js*:

```
const VacationInSeasonListener = require('./models/vacationInSeasonListener')

module.exports = {
  getVacations: async (options = {}) => Vacation.find(options),
  addVacationInSeasonListener: async (email, sku) => {
    await VacationInSeasonListener.updateOne(
      { email },
      { $push: { skus: sku } },
    )
  }
}
```

```
    { upsert: true }  
  )  
},  
}
```

Que mágica é essa? Como podemos “atualizar” um registro na coleção `VacationInSeasonListener` antes mesmo de ele existir? A resposta está em uma conveniência do Mongoose chamada *upsert* (fusão das palavras “update” e “insert”). Basicamente, se um registro com o endereço de email fornecido não existir, ele será criado. Se existir, será atualizado. Depois usamos a variável mágica `$push` para indicar que queremos adicionar um valor a um array.



Esse código não impede que múltiplos SKUs sejam adicionados ao registro se o usuário preencher o formulário várias vezes. Quando chegar a época de um pacote de férias e encontrarmos todos os clientes que quiserem ser notificados, teremos de tomar cuidado para não os notificar várias vezes.

Certamente, a essa altura, já abordamos os aspectos básicos importantes! Aprendemos a nos conectar com uma instância do MongoDB, a alimentá-la com dados, a ler esses dados e a gravar atualizações! Contudo, você pode querer usar um RDBMS, portanto, examinaremos como fazer o mesmo com o PostgreSQL.

PostgreSQL

Bancos de dados orientados a objetos como o MongoDB são ótimos e costumam ser melhores para iniciantes, mas, se você está tentando construir uma aplicação robusta, pode ter de dedicar à estruturação de seus bancos de dados orientados a objetos a mesma quantidade de trabalho – ou mais – que dedicaria ao planejamento de um banco de dados relacional tradicional. Além disso, talvez você já tenha experiência com bancos de dados relacionais ou pode querer se conectar com um já existente.

Felizmente, há um suporte robusto para cada um dos principais bancos de dados relacionais no ecossistema JavaScript, e, se você quiser ou precisar usar um banco de dados relacional, não deve ter problemas.

Pegaremos nosso banco de dados de férias e o reimplementaremos usando um banco de dados relacional. Para esse exemplo, usaremos o PostgreSQL, um banco de dados relacional open source popular e sofisticado. As técnicas e princípios que empregaremos serão semelhantes aos de qualquer banco de dados relacional.

Como o ODM que usamos para o MongoDB, há ferramentas de mapeamento objeto-relacional (ORM, object-relational mapping) disponíveis para bancos de dados relacionais. No entanto, provavelmente quase todos os leitores interessados nesse tópico já devem estar familiarizados com os bancos de dados relacionais e o SQL, logo, empregaremos diretamente um cliente PostgreSQL para Node.

Semelhante ao que fizemos no caso do MongoDB, usaremos um serviço de PostgreSQL gratuito online. É claro que, se você se sentir à vontade para instalar e configurar seu próprio banco de dados PostgreSQL, também pode fazê-lo. A única coisa que mudará é a string da conexão. Se você usar sua própria instância do PostgreSQL, certifique-se de que seja a 9.4 ou posterior, porque utilizaremos o tipo de dado JSON, que foi introduzido na versão 9.4 (enquanto escrevo este texto, estou usando a versão 11.3).

Há muitas opções de PostgreSQL online; nesse exemplo, empregaremos o *ElephantSQL* (<https://www.elephantsql.com>). Não poderia ser mais fácil começar a usá-lo: crie uma conta (você pode utilizar sua conta do GitHub para fazer login) e clique em Create New Instance. Basta fornecer um nome (por exemplo, “meadowlark”) e selecionar um plano (é possível usar o plano gratuito). Você também deve especificar uma região (tente selecionar a mais próxima). Feito isso, você encontrará uma seção Details listando informações sobre sua instância. Copie a URL (a string da conexão), que inclui o nome de usuário, a senha e a localização da instância na mesma string.

Insira esta string em seu arquivo *.credentials.development.json*:

```
"postgres": {  
  "connectionString": "your_dev_connection_string"  
}
```

Uma das diferenças entre os bancos de dados orientados a objetos e os RDBMSs é que estes requerem um trabalho preparatório maior para a

definição de seu esquema e um código SQL de definição de dados que crie o esquema antes da inclusão ou recuperação de dados. Para manter esse paradigma, faremos isso como uma etapa separada em vez de deixar que o ODM ou o ORM a manipule, como fizemos com o MongoDB.

Poderíamos criar scripts SQL e usar um cliente de linha de comando para executar os scripts de definição de dados que criarão nossas tabelas ou fazer esse trabalho em JavaScript com a API cliente do PostgreSQL, mas em uma etapa separada isso é feito uma única vez. Já que este é um livro sobre o Node e o Express, adotaremos a segunda abordagem.

Primeiro, teremos de instalar a biblioteca cliente pg (npm install pg). Em seguida, criaremos *db-init.js*, que só será executado para inicializar nosso banco de dados e é diferente do arquivo *db.js*, usado sempre que o servidor é iniciado (*ch13/01-postgres/db.js* no repositório fornecido):

```
const { credentials } = require('./config')

const { Client } = require('pg')
const { connectionString } = credentials.postgres
const client = new Client({ connectionString })

const createScript = `
CREATE TABLE IF NOT EXISTS vacations (
  name varchar(200) NOT NULL,
  slug varchar(200) NOT NULL UNIQUE,
  category varchar(50),
  sku varchar(20),
  description text,
  location_search varchar(100) NOT NULL,
  location_lat double precision,
  location_lng double precision,
  price money,
  tags jsonb,
  in_season boolean,
  available boolean,
  requires_waiver boolean,
  maximum_guests integer,
  notes text,
```

```
    packages_sold integer
  );
`
```

```
const getVacationCount = async client => {
  const { rows } = await client.query('SELECT COUNT(*) FROM VACATIONS')
  return Number(rows[0].count)
}
```

```
const seedVacations = async client => {
  const sql = `
    INSERT INTO vacations(
      name,
      slug,
      category,
      sku,
      description,
      location_search,
      price,
      tags,
      in_season,
      available,
      requires_waiver,
      maximum_guests,
      notes,
      packages_sold
    ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14)
  `
```

```
  await client.query(sql, [
    'Hood River Day Trip',
    'hood-river-day-trip',
    'Day Trip',
    'HR199',
    'Spend a day sailing on the Columbia and enjoying craft beers in Hood River!',
    'Hood River, Oregon, USA',
    99.95,
    `["day trip", "hood river", "sailing", "windsurfing", "breweries"]`,
    true,
    true,
```

```

    false,
    16,
    null,
    0,
  ])
  // podemos usar o mesmo padrão para inserir
  // outros dados de férias aqui...
}

client.connect().then(async () => {
  try {
    console.log('creating database schema')
    await client.query(createScript)
    const vacationCount = await getVacationCount(client)
    if(vacationCount === 0) {
      console.log('seeding vacations')
      await seedVacations(client)
    }
  } catch(err) {
    console.log('ERROR: could not initialize database')
    console.log(err.message)
  } finally {
    client.end()
  }
})

```

Começaremos pelo fim do arquivo. Pegamos nosso cliente de banco de dados (client) e chamamos connect() nele; esse método estabelece uma conexão de banco de dados e retorna uma promessa. Quando essa promessa for resolvida, poderemos executar ações no banco de dados.

A primeira coisa que fizemos foi chamar client.query(createScript), que criará nossa tabela vacations (o que também é conhecido como *relação*). Se examinarmos createScript, veremos que esse é um SQL de definição de dados. O estudo de SQL não faz parte do escopo deste livro, mas, se você está lendo esta seção, presumo que tenha pelo menos algum conhecimento de SQL. Algo que você deve ter notado é que usamos snake_case para nomear os campos em vez de camelCase. Isto é, onde tínhamos “inSeason” agora temos “in_season”. Embora seja possível usar camelCase para nomear estruturas no

PostgreSQL, qualquer identificador precisa usar letras maiúsculas, o que acaba trazendo mais problemas do que benefícios. Voltaremos a essa questão um pouco mais à frente.

Você deve ter notado que tivemos de dar uma atenção maior ao esquema. Que tamanho o nome de um pacote de férias pode ter? (aqui lhe demos uma abrangência de 200 caracteres). Que tamanho os nomes de categorias e a SKU podem ter? Observe que estamos usando o tipo `money` do PostgreSQL para o preço e tornamos `slug` nossa chave primária (em vez de adicionar um ID separado).

Se você conhece bancos de dados relacionais, não há nada de novo nesse esquema simples. No entanto, a maneira como manipulamos as “tags” (identificações) pode ter lhe chamado a atenção.

No design de banco de dados tradicional, provavelmente criaríamos uma nova tabela para estabelecer relacionamentos entre as férias e as tags (isso se chama *normalização*). Poderíamos fazer o mesmo aqui. Porém, é nesse momento que precisamos decidir se será melhor adotar o design de um banco de dados relacional tradicional ou agir da “maneira JavaScript”. Se usarmos duas tabelas (`vacations` e `vacation_tags`, por exemplo), teremos de consultar dados em ambas para criar um único objeto contendo todas as informações sobre um pacote de férias, como fizemos no caso do MongoDB. Pode haver razões para essa complexidade adicional relacionadas ao desempenho, mas suponhamos que não haja; queremos apenas poder determinar rapidamente as tags de um pacote de férias específico. Poderíamos usar um campo de texto e separar nossas tags com vírgulas, mas teríamos de fazer o parsing das tags, e o PostgreSQL fornece uma maneira melhor com os tipos de dados JSON. Veremos em breve que, usando JSON (`jsonb`, uma representação binária que geralmente fornece um desempenho melhor), podemos armazenar esses dados como um array JavaScript, como fizemos no MongoDB.

Para concluir, inserimos nossos dados iniciais (seed data) no banco de dados usando o mesmo conceito básico de antes: se a tabela `vacations` estiver vazia, adicionaremos alguns dados iniciais; caso contrário, vamos presumir que já o fizemos.

Observe que a inserção de dados é um pouco mais complicada do que com o

MongoDB. Há algumas maneiras de resolver esse problema, mas, nesse exemplo, queremos ser explícitos no uso de SQL. Poderíamos criar uma função para executar instruções de inserção mais naturalmente ou usar um ORM (veremos mais sobre isso posteriormente). Contudo, por enquanto o SQL atende bem, e essa abordagem deve ser fácil para qualquer pessoa que já conheça SQL.

Repare também que, embora esse script tenha sido projetado para ser executado somente uma vez para inicializar e alimentar nosso banco de dados, o criamos de um modo que seja seguro executá-lo várias vezes. Incluímos a opção `IF NOT EXISTS` e verificamos se a tabela `vacations` está vazia antes de adicionar dados iniciais.

Já podemos executar o script para inicializar nosso banco de dados:

```
$ node db-init.js
```

Agora que o banco de dados está definido, podemos escrever o código que nos permitirá *usá-lo* em nosso site.

Normalmente, os servidores de banco de dados só conseguem manipular um número limitado de conexões simultâneas, logo, os servidores web costumam implementar uma estratégia chamada *pooling de conexões* para balancear o overhead do estabelecimento de uma conexão e o perigo de deixar conexões abertas por muito tempo e travar o servidor. Felizmente, os detalhes dessa operação serão manipulados para nós pelo cliente PostgreSQL para Node.

Dessa vez usaremos uma estratégia um pouco diferente em nosso arquivo `db.js`. Em vez de o arquivo estabelecer a conexão com o banco de dados, ele retornará uma API criada por nós para manipular os detalhes dessa comunicação.

Também temos uma decisão a tomar sobre nosso modelo de férias. Lembre-se de que, quando criamos nosso modelo, usamos `snake_case` para o esquema do banco de dados, mas todo o código JavaScript está usando `camelCase`. De um modo geral, temos três opções:

- Refatorar o esquema para que use `camelCase`. Isso tornará nosso SQL mais deselegante porque teremos de nos lembrar de escrever os nomes das propriedades corretamente.

- Usar snake_case no JavaScript. Não é a opção ideal porque gostamos de padrões (certo?).
- Usar snake_case no lado do banco de dados e converter para camelCase no lado JavaScript. Dará mais trabalho, porém manterá o SQL e o JavaScript com seu aspecto genuíno.

Felizmente, a terceira opção pode ser manipulada automaticamente. Poderíamos criar nossa própria função para fazer essa conversão, mas usaremos uma biblioteca utilitária popular chamada *Lodash* (<https://lodash.com>), que facilita sua execução. Para instalá-la, basta executar `npm install lodash`.

No momento, as necessidades de nosso banco de dados são muito modestas. Só precisamos buscar todos os pacotes de férias disponíveis, logo, o arquivo *db.js* terá esta aparência (*ch13/01-postgres/db.js* no repositório fornecido):

```
const { Pool } = require('pg')
const _ = require('lodash')

const { credentials } = require('./config')

const { connectionString } = credentials.postgres
const pool = new Pool({ connectionString })

module.exports = {
  getVacations: async () => {
    const { rows } = await pool.query('SELECT * FROM VACATIONS')
    return rows.map(row => {
      const vacation = _.mapKeys(row, (v, k) => _.camelCase(k))
      vacation.price = parseFloat(vacation.price.replace(/^\$/, ''))
      vacation.location = {
        search: vacation.locationSearch,
        coordinates: {
          lat: vacation.locationLat,
          lng: vacation.locationLng,
        },
      }
      return vacation
    })
  }
}
```

```
}}}
```

Bem conciso! Estamos exportando um único método chamado `getVacations` que faz o que o seu nome sugere. Ele também usa as funções `mapKeys` e `camelCase` do `Lodash` para converter as propriedades do banco de dados para `camelCase`.

Não podemos nos esquecer de que temos de manipular o atributo `price` com cuidado. O tipo `money` do PostgreSQL é convertido pela biblioteca `pg` em uma string já formatada. E por uma boa razão: como discutimos, só recentemente o JavaScript adicionou o suporte a tipos numéricos de precisão arbitrária (`BigInt`), mas ainda não há um adaptador para PostgreSQL que se beneficie disso (e ele pode não ser o tipo de dado mais eficiente para todos os casos). Poderíamos alterar o esquema do banco de dados para usar um tipo numérico em vez do tipo `money`, mas não devemos deixar nossas opções de front-end guiar o esquema. Também poderíamos lidar com as strings pré-formatadas retornadas pela biblioteca `pg`, mas teríamos de alterar todo o código existente, que espera que `price` seja um número. Além disso, essa abordagem prejudicaria nossa capacidade de fazer cálculos numéricos no front-end (como somar os preços dos itens do carrinho de compras). Por todas essas razões, optamos por converter a string em um número quando a recuperarmos no banco de dados.

Também pegamos nossas informações de localização – que não têm uma “estrutura” na tabela – e as convertemos para uma estrutura mais típica do JavaScript. Estamos fazendo isso apenas para que fique igual ao exemplo do MongoDB; poderíamos usar os dados na estrutura em que se encontram (ou modificar o exemplo do MongoDB para usar uma estrutura plana).

A última coisa que precisamos aprender a fazer com o PostgreSQL é atualizar dados, logo, forneceremos o conteúdo do recurso de ouvinte de “férias da temporada” (`vacation in season listener`).

Adicionando dados

Como no caso do MongoDB, usaremos nosso exemplo de ouvinte de “férias da temporada”. Começaremos adicionando a definição de dados a seguir à string de `createScript` em `db-init.js`:

```
CREATE TABLE IF NOT EXISTS vacation_in_season_listeners (  
  email varchar(200) NOT NULL,  
  sku varchar(20) NOT NULL,  
  PRIMARY KEY (email, sku)  
);
```

Lembre-se de que tomamos o cuidado de gravar *db-init.js* de maneira não destrutiva para poder executá-lo a qualquer momento. Logo, podemos executá-lo novamente para criar a tabela `vacation_in_season_listeners`.

Agora podemos modificar *db.js* para incluir o método que atualizará essa tabela:

```
module.exports = {  
  //...  
  addVacationInSeasonListener: async (email, sku) => {  
    await pool.query(  
      'INSERT INTO vacation_in_season_listeners (email, sku) ' +  
      'VALUES ($1, $2) ' +  
      'ON CONFLICT DO NOTHING',  
      [email, sku]  
    )  
  },  
}
```

A cláusula `ON CONFLICT` do PostgreSQL ativa os upserts. Nesse caso, se a combinação exata de email e SKU estiver presente, o usuário já terá se registrado para receber notificações, portanto, não precisamos fazer mais nada. Se tivéssemos outras colunas nessa tabela (como a data do último registro), poderíamos usar uma cláusula `ON CONFLICT` mais sofisticada (consulte a *documentação do PostgreSQL sobre INSERT* [<http://bit.ly/3724FJI>] para obter informações adicionais). Observe também que esse comportamento depende da maneira como a tabela é definida. Tornamos o email e a SKU uma chave primária composta, significando que não pode haver duplicatas, o que por sua vez demandou o uso da cláusula `ON CONFLICT` (caso contrário, o comando `INSERT` resultaria em um erro na segunda vez que um usuário tentasse se registrar para ser notificado sobre o mesmo pacote de férias).

Vimos um exemplo completo do uso de dois tipos de bancos de dados, um

banco de dados orientado a objetos e um RDBMS. Deve ter ficado claro que a função do banco de dados é a mesma: armazenar, recuperar e atualizar dados de maneira consistente e escalável. Já que a função é a mesma, conseguimos criar uma camada de abstração para poder selecionar uma tecnologia de banco de dados diferente. A última coisa para a qual podemos precisar de um banco de dados é o armazenamento persistente de sessões, que mencionamos no Capítulo 9.

Usando um banco de dados para o armazenamento de sessões

Como discutimos no Capítulo 9, não é adequado usar um armazenamento na memória para dados de sessões em um ambiente de produção. Felizmente, é fácil usar um banco de dados como armazenamento de sessões.

Poderíamos usar o banco de dados MongoDB ou PostgreSQL já existente para o armazenamento de sessões, mas um banco de dados maduro é essencial para esse tipo de armazenamento, sendo um caso de uso perfeito para um banco de dados de chave-valor. Quando redigi esse texto, os bancos de dados de chave-valor mais populares para o armazenamento de sessões eram o *Redis* (<https://redis.io>) e o *Memcached* (<https://memcached.org>). Para manter a consistência com os outros exemplos deste capítulo, usaremos um serviço online gratuito para fornecer um banco de dados Redis.

Começaremos acessando o site da *Redis Labs* (<https://redislabs.com>) e criando uma conta. Em seguida, crie uma assinatura e um plano gratuitos. Selecione Cache para o plano e dê um nome para o banco de dados; deixe o resto das configurações com seus padrões.

Você será levado para a tela View Database e, pelo menos quando escrevi este texto, as informações críticas não eram preenchidas em alguns segundos, logo, tenha paciência. O que queremos é o campo Endpoint e a senha do Redis (Redis Password) que fica em Access Control & Security (ela fica oculta por padrão, mas há um pequeno botão próximo que a exhibe). Pegue essas informações e insira-as em seu arquivo `.credentials.development.json`:

```
"redis": {  
  "url": "redis://:<SUA SENHA >@<SEU ENDPOINT>"
```

```
}
```

Observe que a URL é um pouco estranha: normalmente há um nome de usuário antes dos dois pontos precedendo a senha, mas o Redis permite que a conexão seja estabelecida somente com a senha; no entanto, os dois pontos que separam o nome de usuário e a senha continuam sendo obrigatórios.

Usaremos um pacote chamado `connect-redis` para fornecer o armazenamento de sessões do Redis. Uma vez que você o instalar (`npm install connect-redis`), poderemos configurá-lo no arquivo principal da aplicação. Continuaremos usando `express-session`, mas agora passaremos uma nova propriedade para ele, `store`, que o configurará para usar um banco de dados. Repare que é preciso passar `expressSession` para a função retornada por `connect-redis` para obtermos o construtor: essa é uma particularidade muito comum dos armazenamentos de sessões (*ch13/00-mongodb/meadowlark.js* ou *ch13/01-postgres/meadowlark.js* no repositório fornecido):

```
const expressSession = require('express-session')
const RedisStore = require('connect-redis')(expressSession)

app.use(cookieParser(credentials.cookieSecret))
app.use(expressSession({
  resave: false,
  saveUninitialized: false,
  secret: credentials.cookieSecret,
  store: new RedisStore({
    url: credentials.redis.url,
    logErrors: true, // altamente recomendado!
  }),
}))
```

Usaremos o armazenamento de sessão recém-criado para algo útil. Suponhamos que quiséssemos exibir os preços dos pacotes de férias em diferentes moedas. Também queremos que o site lembre-se da moeda selecionada pelo usuário.

Começaremos adicionando um selecionador de moeda na parte inferior de nossa página de pacotes de férias:

```
<hr>
<p>Currency:
```

```

<a href="/set-currency/USD" class="currency {{currencyUSD}}">USD</a> |
<a href="/set-currency/GBP" class="currency {{currencyGBP}}">GBP</a> |
<a href="/set-currency/BTC" class="currency {{currencyBTC}}">BTC</a>
</p>

```

Agora forneceremos um pouco de estilo CSS (você pode inserir o estilo inline em seu arquivo *views/layouts/main.handlebars* ou vinculá-lo a um arquivo CSS no diretório *public*):

```

a.currency {
  text-decoration: none;
}
.currency.selected {
  font-weight: bold;
  font-size: 150%;
}

```

Para concluir, adicionaremos um manipulador de rota para definir a moeda e o modificaremos para */vacations* para exibir os preços na moeda atual (*ch13/00-mongodb/lib/handlers.js* ou *ch13/01-postgres/lib/handlers.js* no repositório fornecido):

```

exports.setCurrency = (req, res) => {
  req.session.currency = req.params.currency
  return res.redirect(303, '/vacations')
}

function convertFromUSD(value, currency) {
  switch(currency) {
    case 'USD': return value * 1
    case 'GBP': return value * 0.79
    case 'BTC': return value * 0.000078
    default: return NaN
  }
}

exports.listVacations = (req, res) => {
  Vacation.find({ available: true }, (err, vacations) => {
    const currency = req.session.currency || 'USD'
    const context = {
      currency: currency,
      vacations: vacations.map(vacation => {

```

```

    return {
      sku: vacation.sku,
      name: vacation.name,
      description: vacation.description,
      inSeason: vacation.inSeason,
      price: convertFromUSD(vacation.price, currency),
      qty: vacation.qty,
    }
  })
}
switch(currency){
  case 'USD': context.currencyUSD = 'selected'; break
  case 'GBP': context.currencyGBP = 'selected'; break
  case 'BTC': context.currencyBTC = 'selected'; break
}
res.render('vacations', context)
})
}

```

Também temos de adicionar uma rota para a definição da moeda em *meadowlark.js*:

```
app.get('/set-currency/:currency', handlers.setCurrency)
```

Claro que essa não é uma boa maneira de executar a conversão da moeda. Seria melhor utilizar uma API de conversão de moeda de terceiros para assegurar que nossas taxas estejam atualizadas. No entanto, isso será suficiente para fins de demonstração. Agora você pode alternar-se entre várias moedas e – vá em frente e tente fazê-lo – interromper e reiniciar o servidor. Verá que ele se lembra da moeda de sua preferência! Se você remover os cookies, a moeda preferencial será esquecida. Observe que perdemos a formatação da moeda; ela ficou mais complicada e vou deixá-la como exercício para o leitor.

Outro exercício interessante para o leitor seria transformar set-currency em uma rota de uso geral para torná-la mais útil. Atualmente, ela sempre fará o redirecionamento para a página de férias, mas e se quiséssemos usá-la em uma página de carrinho de compras? Veja se consegue pensar em uma ou duas maneiras de resolver esse problema.

Se você examinar seu banco de dados, verá que há uma nova coleção chamada *sessions*. Se explorá-la, encontrará um documento com seu ID de sessão (propriedade *sid*) e a moeda de sua preferência.

Conclusão

Sem dúvida abordamos um material bastante extenso neste capítulo. O banco de dados é a base que torna a maioria das aplicações web úteis. O design e o ajuste de bancos de dados compõem um tópico vasto que poderia se estender por vários livros, mas espero que essa explanação tenha fornecido as ferramentas básicas necessárias para você conectar dois tipos de bancos de dados e fazer transferências entre eles.

Agora que examinamos essa peça fundamental, revisitaremos o roteamento e o importante papel que ele desempenha nas aplicações web.

¹ N.T.: Bed and Breakfast (cama e café da manhã, em português) é o aluguel de uma cama na casa de um habitante local, com direito a uma refeição inclusa na diária, geralmente o café da manhã.

CAPÍTULO 14

Roteamento

O roteamento é um dos aspectos mais importantes de um site ou web service; felizmente, no Express ele é simples, flexível e robusto. *Roteamento* é o mecanismo pelo qual as requisições (como especificado por uma URL e um método HTTP) são roteadas para o código que as manipula. Como já mencionado, ele costumava ser simples e baseado em arquivo. Por exemplo, se você inserisse o arquivo *foo/about.html* em seu site, poderia acessá-lo a partir do navegador com o path */foo/about.html*. Simples, mas inflexível. E, caso você não tenha percebido, atualmente é ultrapassado usar *html* na URL.

Antes de examinar os aspectos técnicos do roteamento com o Express, precisamos discutir o conceito de *arquitetura da informação* (AI). A AI é a organização conceitual do conteúdo. A existência de uma AI extensível (porém não excessivamente complicada) antes de começarmos a pensar no roteamento trará muitos benefícios mais à frente.

Um dos ensaios mais inteligentes e atemporais sobre AI foi escrito por Tim Berners-Lee, que praticamente *inventou a internet*. Você pode (e deve) lê-lo agora: <http://www.w3.org/Provider/Style/URI.html>. Ele foi escrito em 1998. Pense bem, não há muita coisa que foi escrita sobre a tecnologia da internet em 1998 que continue sendo verdade como o era na época.

Segundo esse ensaio, a grande responsabilidade que devemos assumir é a seguinte:

É tarefa do webmaster alocar URIs que possamos usar em 2, 20 ou 200 anos. Isso demanda reflexão, organização e compromisso.

TIM BERNERS-LEE

Gosto de imaginar uma época em que para a classe de web design permitir a atuação do profissional, como em outras áreas da engenharia, fosse preciso fazer um juramento. (O leitor astuto desse artigo achará engraçado o fato de sua URL terminar com *.html*).

Para fazer uma comparação (que infelizmente talvez os mais jovens não entendam), suponhamos que a cada dois anos nossa biblioteca favorita reordenasse totalmente o sistema decimal de Dewey. Entraríamos na biblioteca certo dia e não conseguiríamos encontrar nada. É exatamente isso que acontece quando reprojecemos nossa estrutura de URL.

Pense com cuidado em suas URLs. Elas ainda farão sentido daqui a 20 anos? (200 anos pode ser muito tempo: quem sabe se estaremos usando URLs até lá? Mesmo assim, admiro a dedicação de quem pensa considerando um futuro tão distante). Pondere cuidadosamente a divisão do conteúdo. Categorize as coisas logicamente e não se perca. É ciência, mas também é arte.

Talvez o mais importante seja trabalhar com outras pessoas ao projetar suas URLs. Mesmo se você for o melhor arquiteto da informação das redondezas, pode ficar surpreso com como as pessoas olham para o mesmo conteúdo com uma visão totalmente diferente. Não estou dizendo que você deva tentar buscar uma AI que faça sentido para *todo mundo* (porque geralmente isso não é possível), mas conseguir enxergar o problema a partir de vários pontos de vista lhe trará ideias melhores e exporá as falhas da AI que imaginou.

Aqui estão algumas sugestões que o ajudarão a chegar a uma AI duradoura:

Nunca exponha detalhes técnicos em suas URLs

Já acessou um site, notou que a URL terminava com *.asp* e o considerou totalmente desatualizado? Lembre-se de que no passado ASP era inovador. Embora me entristeça dizê-lo, o mesmo ocorrerá com JavaScript, JSON, Node e Express. Espero que não por longos e férteis anos, mas o tempo não

costuma ser complacente com a tecnologia.

Evite informações irrelevantes em suas URLs

Pense com cuidado em cada palavra de sua URL. Se alguma não significar nada, deixe-a de fora. Por exemplo, é constrangedor ver sites usarem a palavra *home* nas URLs. A URL raiz é a home page. Não precisamos de URLs como */home/directions* e */home/contact*.

Evite URLs desnecessariamente longas

Sob as mesmas condições, uma URL curta é melhor que uma mais longa. No entanto, você não deve tentar criar URLs curtas se elas prejudicarem a clareza ou para se beneficiar da SEO. É tentador usar abreviações, mas pense bem ao fazê-lo. Elas devem ser comuns e conhecidas para que você possa imortalizá-las em uma URL.

Seja consistente com os separadores de palavras

É muito comum a separação de palavras com hifens, e um pouco menos comum com underscores. Geralmente os hifens são considerados esteticamente mais agradáveis que os underscores, e quase todos os especialistas em SEO recomendam seu uso. Preferindo os hifens ou os underscores, seja consistente ao usá-los.

Nunca use espaço em branco ou caracteres não digitáveis

Não é recomendável usar espaço em branco em uma URL. Geralmente ele é convertido em um sinal de adição (+), gerando confusão. É claro que você deve evitar caracteres não digitáveis, e devo preveni-lo contra o uso de algo diferente de caracteres alfanuméricos, números, travessões e underscores. Pode parecer inteligente, mas o que é “inteligente” não costuma se sustentar com o passar do tempo. Obviamente, se seu site não for para um público-alvo do idioma português, você pode usar caracteres não pertencentes a ele (com codificação por cento), mas isso costuma causar problemas na localização do site.

Use letras minúsculas em suas URLs

Essa sugestão pode provocar debates. Há quem ache que combinar maiúsculas e minúsculas nas URLs é não só aceitável, mas também preferível. Não quero começar um debate rigoroso sobre isso, mas devo enfatizar que a vantagem das letras minúsculas é que elas sempre podem ser geradas automaticamente em código. Se você já teve de examinar um site e sanitizar milhares de links ou fazer comparações de strings, concordará com esse argumento. Pessoalmente acho as URLs em minúsculas esteticamente mais agradáveis, mas a decisão é sua.

As rotas e a SEO

Se você quiser que seu site seja facilmente descoberto (que é o que a maioria das pessoas quer), tem de considerar o uso da SEO e como suas URLs podem afetá-la. Especificamente, se certas palavras-chave forem importantes – *e fizer sentido* –, considere torná-las parte da URL. Por exemplo, a Meadowlark Travel oferece vários pacotes de férias para Oregon Coast. Para assegurar que o mecanismo de busca dê uma classificação (ranking) mais alta a esses pacotes, usaremos a string “Oregon Coast” no título, no cabeçalho, no corpo e na metadescrição, e as URLs começarão com `/vacations/oregon-coast`. O pacote de férias para Manzanita pode ser encontrado em `/vacations/oregon-coast/manzanita`. Se, para encurtar a URL, usássemos apenas `/vacations/manzanita`, perderíamos uma SEO valiosa.

Dito isso, resista à tentação de encher desatentamente as URLs com palavras-chave na tentativa de melhorar a classificação. Não vai dar certo. Por exemplo, seria errado alterar a URL do pacote de férias de Manzanita para `/vacations/oregon-coast-portland-and-hood-river/oregon-coast/manzanita` em um esforço para dizer “Oregon Coast” mais uma vez e inserir ao mesmo tempo as palavras-chave “Portland” e “Hood River”. Vai contra as regras de uma boa AI e será contraproducente.

Subdomínios

Além de path, os subdomínios são a outra parte da URL normalmente usada para rotear requisições. Os subdomínios devem ser reservados para partes significativamente diferentes da aplicação – como para uma API REST

(*api.meadowlarktravel.com*) ou uma interface administrativa (*admin.meadowlarktravel.com*). À vezes eles são usados por razões técnicas. Por exemplo, se construíssemos nosso blog com o WordPress (enquanto o resto do site usa o Express), seria mais fácil utilizar *blog.meadowlarktravel.com* (uma solução melhor seria empregar um servidor proxy, como o NGINX). Geralmente, a divisão do conteúdo com o uso de subdomínios traz consequências para a SEO, e é por isso que você deve reservá-los para áreas do site que não sejam importantes para a SEO, como áreas administrativas e APIs. Lembre-se disso e verifique se não há outra opção antes de usar um subdomínio para conteúdo que seja importante para seu plano de SEO.

Por padrão, o mecanismo de roteamento do Express não leva os subdomínios em consideração: `app.get(/about)` manipulará requisições para *http://meadowlarktravel.com/about*, *http://www.meadowlarktravel.com/about* e *http://admin.meadowlarktravel.com/about*. Se você quiser manipular um subdomínio separadamente, pode usar um pacote chamado *vhost* (abreviação de “virtual host,” que vem de um mecanismo do Apache usado para a manipulação de subdomínios). Primeiro, instale o pacote (`npm install vhost`). Para testar o roteamento baseado em domínio em sua máquina de desenvolvimento, você precisará “simular” nomes de domínio. Felizmente, é para isso que serve o *arquivo hosts*. Em máquinas macOS e Linux, ele pode ser encontrado em `/etc/hosts`, e no Windows em `c:\windows\system32\drivers\etc\hosts`. Adicione as linhas a seguir ao seu arquivo *hosts* (você precisará de privilégios administrativos para editá-lo):

```
127.0.0.1 admin.meadowlark.local
```

```
127.0.0.1 meadowlark.local
```

Isso dirá ao seu computador que ele deve tratar *meadowlark.local* e *admin.meadowlark.local* da mesma forma que os domínios comuns da internet, mas deve mapeá-los para *localhost* (127.0.0.1). Usamos o domínio de nível superior *.local* para não nos confundirmos (você poderia usar *.com* ou qualquer outro domínio da internet, mas ele sobreporia o domínio real, o que pode levar a falhas).

Em seguida, podemos usar o middleware *vhost* para trabalhar com o

roteamento baseado em domínio (*ch14/00-subdomains.js* no repositório fornecido):

```
// cria o subdomínio "admin"...essa parte deve vir
// antes de todas as suas outras rotas
var admin = express.Router()
app.use(vhost('admin.meadowlark.local', admin))

// cria rotas administrativas; elas podem ser definidas em qualquer local
admin.get('*', (req, res) => res.send('Welcome, Admin!'))

// rotas comuns
app.get('*', (req, res) => res.send('Welcome, User!'))
```

`express.Router()` criará uma nova instância do roteador do Express. Você pode tratá-la como sua instância original (`app`). Assim poderá adicionar rotas e middleware como faria com `app`. No entanto, ela não fará nada até você a adicionar a `app`. Fizemos isso com `vhost`, que vinculou essa instância do roteador a esse subdomínio.



`express.Router` também é útil para a divisão de rotas, permitindo que usemos vários manipuladores de rota ao mesmo tempo. Consulte a *documentação de roteamento do Express* (<http://bit.ly/2X8VC59>) para obter mais informações.

Manipuladores de rota são middleware

Já vimos o roteamento básico correspondente a um path específico. Porém, o que `app.get('/foo', ...)` faz realmente? Como vimos no Capítulo 10, é apenas um middleware especializado, que pode receber um método `next`. Examinaremos alguns exemplos mais sofisticados (*ch14/01-fifty-fifty.js* no repositório fornecido):

```
app.get('/fifty-fifty', (req, res, next) => {
  if(Math.random() < 0.5) return next()
  res.send('sometimes this')
})
app.get('/fifty-fifty', (req, res) => {
  res.send('and sometimes that')
```

```
})
```

No exemplo anterior, temos dois manipuladores para a mesma rota. Normalmente, o primeiro teria prioridade, mas, nesse caso, ele será usado durante aproximadamente metade do tempo, dando chance ao segundo. Não precisamos nem mesmo usar `app.get` duas vezes: podemos usar quantos manipuladores quisermos para uma única chamada a `app.get`. Aqui está um exemplo com chance aproximadamente igual de fornecer três respostas diferentes (*ch14/02-red-green-blue.js* no repositório fornecido):

```
app.get('/rgb',
  (req, res, next) => {
    // cerca de um terço das requisições retornará "red"
    if(Math.random() < 0.33) return next()
    res.send('red')
  },
  (req, res, next) => {
    // metade dos 2/3 das requisições restantes
    // (ou seja, outro terço) retornará "green"
    if(Math.random() < 0.5) return next()
    res.send('green')
  },
  function(req, res){
    // e o último terço retornará "blue"
    res.send('blue')
  },
)
```

Embora inicialmente talvez isso não pareça muito útil, permite criar funções genéricas que possam ser usadas em qualquer rota. Por exemplo, digamos que tivéssemos um mecanismo que exibisse ofertas especiais em certas páginas. As ofertas especiais mudam sempre e não são exibidas em todas as páginas. Podemos criar uma função para injetar essas ofertas na propriedade `res.locals` (que vimos no Capítulo 7) (*ch14/03-specials.js* no repositório fornecido):

```
async function specials(req, res, next) {
  res.locals.special = await getSpecialsFromDatabase()
  next()
}
```

```
app.get('/page-with-specials', specials, (req, res) =>
  res.render('page-with-specials')
)
```

Também poderíamos implementar um mecanismo de autorização com essa abordagem. Suponhamos que nosso código de autorização definisse uma variável de sessão `req.session.authorized`. Podemos usar o código a seguir para criar um filtro de autorização reutilizável (*ch14/04-authorizer.js* no repositório fornecido):

```
function authorize(req, res, next) {
  if(req.session.authorized) return next()
  res.render('not-authorized')
}
```

```
app.get('/public', () => res.render('public'))
```

```
app.get('/secret', authorize, () => res.render('secret'))
```

Paths de rota e expressões regulares

Quando especificamos um path (como `/foo`) em uma rota, ele é convertido em uma expressão regular pelo Express. Certos metacaracteres de expressão regular estão disponíveis em paths de rota: `+`, `?`, `*`, `(` e `)`. Examinaremos alguns exemplos. Digamos que você quisesse que as URLs `/user` e `/username` fossem manipuladas pela mesma rota:

```
app.get('/user(name)?', (req, res) => res.render('user'))
```

Um de meus sites de novidades favoritos – infelizmente já extinto – era o <http://khaaan.com>. Tratava-se apenas do capitão de nave estelar preferido de todas as pessoas dizendo sua frase mais icônica. Inútil, mas sempre me fazia rir. Suponhamos que quiséssemos criar nossa própria página “KHAANAAAAAN”, mas não quiséssemos que nossos usuários tivessem de lembrar se são 2, 3 ou 10 *a*’s. A linha a seguir faria isso:

```
app.get('/khaa+n', (req, res) => res.render('khaaan'))
```

No entanto, nem todos os metacaracteres de regex significam algo nos parthos de rota – só os listados anteriormente. Isso é importante, porque os pontos,

que normalmente são um metacaractere de regex que significam “qualquer caractere”, podem ser usados em rotas sem escape.

Para concluir, se você precisar realmente de todo o poder das expressões regulares para sua rota, há suporte para isso:

```
app.get(/crazy|mad(ness)?|lunacy/, (req,res) =>
  res.render('madness')
)
```

Ainda preciso encontrar uma boa razão para usar metacaracteres de regex em meus paths de rota, imagine então regex completas, mas é bom saber que a funcionalidade está disponível.

Parâmetros de rota

As rotas com regex podem não ter muita utilidade em sua caixa de ferramentas do Express, mas provavelmente você usará os parâmetros de rota com bastante frequência. Resumindo, é uma maneira de transformar parte de sua rota em um parâmetro de variável. Digamos que quiséssemos ter em nosso site uma página para cada membro da equipe. Temos um banco de dados de membros da equipe com biografias e fotos. À medida que a empresa cresce, fica cada vez mais complicado adicionar uma nova rota para cada membro. Vejamos como os parâmetros de rota podem nos ajudar (*ch14/05-staff.js* no repositório fornecido):

```
const staff = {
  mitch: { name: "Mitch",
    bio: 'Mitch is the man to have at your back in a bar fight.' },
  madeline: { name: "Madeline", bio: 'Madeline is our Oregon expert.' },
  walt: { name: "Walt", bio: 'Walt is our Oregon Coast expert.' },
}
app.get('/staff/:name', (req, res, next) => {
  const info = staff[req.params.name]
  if(!info) return next() // acabará resultando no erro 404
  res.render('05-staffer', info)
})
```

Observe como usamos *:name* em nossa rota. Ele encontrará todas as strings (que não incluam uma barra inclinada) e as inserirá no objeto req.params com a

chave `name`. Esse é um recurso que utilizaremos com frequência, principalmente ao criar uma API REST. Podemos ter vários parâmetros na rota. Por exemplo, se quiséssemos dividir a listagem da equipe por cidade, poderíamos usar o seguinte:

```
const staff = {
  portland: {
    mitch: { name: "Mitch", bio: 'Mitch is the man to have at your back.' },
    madeline: { name: "Madeline", bio: 'Madeline is our Oregon expert.' },
  },
  bend: {
    walt: { name: "Walt", bio: 'Walt is our Oregon Coast expert.' },
  },
}

app.get('/staff/:city/:name', (req, res, next) => {
  const cityStaff = staff[req.params.city]
  if(!cityStaff) return next() // cidade não reconhecida -> 404
  const info = cityStaff[req.params.name]
  if(!info) return next() // membro não reconhecido -> 404
  res.render('staffer', info)
})
```

Organizando as rotas

Deve ter ficado claro que seria complicado definir todas as nossas rotas no arquivo principal da aplicação. Além de o arquivo crescer com o tempo, não seria uma separação de funcionalidades adequada porque já há muita coisa ocorrendo nele. Um site simples talvez tenha apenas uma dúzia de rotas ou menos, mas um maior poderia ter centenas delas.

Então, como organizar as rotas? Bem, como você *deseja* organizar suas rotas? O Express não faz exigências para a organização das rotas, logo, podemos dar asas à imaginação.

Abordarei algumas maneiras populares de manipular rotas nas próximas seções, mas recomendo quatro princípios orientadores para a decisão de como organizá-las:

Use funções nomeadas para os manipuladores de rota

Criar manipuladores de rota inline definindo a função que manipula a rota em vários locais funciona para aplicações pequenas ou protótipos, mas sairá logo de controle à medida que o site crescer.

As rotas não devem ser misteriosas

Esse princípio é intencionalmente vago porque um site grande e complexo pode por necessidade requerer um esquema organizacional mais complicado que um site de 10 páginas. Em um lado do espectro teríamos simplesmente de inserir *todas* as rotas do site em um único arquivo para saber onde elas estão. Para sites grandes, isso não é desejável, então, devemos dividir as rotas por áreas funcionais. No entanto, mesmo assim, temos de saber para onde ir para encontrar uma rota específica. Quando você precisar corrigir algo, não vai querer ter de passar uma hora tentando descobrir onde a rota está sendo manipulada. Trabalhei em um projeto ASP.NET MVC que nesse aspecto era um pesadelo. As rotas eram manipuladas em pelo menos 10 locais diferentes, e não fui lógico ou consistente, sendo quase sempre contraditório. Ainda que estivesse familiarizado com esse (enorme) site, tive de perder muito tempo rastreando onde certas URLs eram manipuladas.

A organização das rotas deve ser extensível

Se você tiver 20 ou 30 rotas nesse exato momento, defini-las em um único arquivo pode funcionar. No entanto, e daqui a três anos quando houver 200 rotas? Isso pode ocorrer. Seja qual for o método que escolher, você precisa assegurar que haja espaço para crescer.

Não subestime os manipuladores de rota automáticos baseados em view

Se seu site for composto de muitas páginas estáticas e com URLs fixas, todas as rotas acabarão tendo esta aparência: `app.get('/static/thing', (req, res) => res.render('static/thing'))`. Para reduzir a repetição desnecessária de código, considere usar um manipulador de rota automático baseado em view. Essa abordagem será descrita posteriormente neste capítulo e pode ser usada junto com as rotas personalizadas.

Declarando rotas em um módulo

A primeira etapa para a organização de nossas rotas é colocarmos todas elas em seu próprio módulo. Há várias maneiras de fazer isso. Uma abordagem seria se fizéssemos o módulo retornar um array de objetos contendo propriedades de método e manipulador. Então, você poderia definir as rotas no arquivo da aplicação desta forma:

```
const routes = require('./routes.js')

routes.forEach(route => app[route.method](route.handler))
```

Esse método tem suas vantagens e pode ser adequado para o armazenamento dinâmico de rotas, como em um banco de dados ou em um arquivo JSON. Porém, se você não precisar dessa funcionalidade, recomendo passar a instância de app para o módulo e deixá-la adicionar as rotas. Essa é a abordagem que adotaremos em nosso exemplo. Crie um arquivo chamado *routes.js* e mova todas as rotas existentes para ele:

```
module.exports = app => {

  app.get('/', (req,res) => app.render('home'))

  //...

}
```

Se copiarmos e colarmos, provavelmente teremos problemas. Por exemplo, se tivermos manipuladores de rota inline que usem variáveis ou métodos não disponíveis no novo contexto, essas referências serão inválidas. Poderíamos adicionar as importações necessárias, mas não o faremos. Moveremos os manipuladores para seu próprio módulo em breve e então resolveremos o problema.

Mas o que fazer para usar nossas rotas? Fácil: em *meadowlark.js*, vamos apenas importá-las:

```
require('./routes')(app)
```

Ou poderíamos ser mais explícitos e adicionar uma importação nomeada (que chamaremos de *addRoutes* para refletir melhor sua natureza de função;

também poderíamos usar esse nome para o arquivo se quiséssemos):

```
const addRoutes = require('./routes')
```

```
addRoutes(app)
```

Agrupando os manipuladores logicamente

Para seguir o primeiro princípio orientador (usar funções nomeadas para manipuladores de rota), precisaremos de um local para inserir os manipuladores. Uma opção extrema seria termos um arquivo JavaScript separado para cada manipulador. É difícil imaginar uma situação em que essa abordagem traria benefícios. É melhor agrupar de alguma forma as funcionalidades relacionadas. Isso facilitaria não só usarmos funcionalidades compartilhadas como também fazer alterações em métodos relacionados.

Por enquanto, agruparemos nossas funcionalidades em arquivos separados: *handlers/main.js*, nos quais inseriremos o manipulador da home page, o manipulador de “about” e qualquer manipulador que não tenha outro destino lógico; *handlers/vacations.js*, onde entrarão os manipuladores relacionados às férias, e assim por diante.

Considere *handlers/main.js*:

```
const fortune = require('../lib/fortune')
```

```
exports.home = (req, res) => res.render('home')
```

```
exports.about = (req, res) => {  
  const fortune = fortune.getFortune()  
  res.render('about', { fortune })  
}
```

```
//...
```

Agora modificaremos *routes.js* para fazer uso dele:

```
const main = require('./handlers/main')
```

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

```
app.get('/', main.home)
app.get('/about', main.about)
//...

}
```

Isso atende a todos os princípios orientadores. */routes.js* é *bem* simples. É fácil ver que rotas existem no site e onde elas estão sendo manipuladas. Também deixamos bastante espaço para crescer. Podemos agrupar funcionalidades relacionadas em quantos arquivos diferentes quisermos. E, se *routes.js* ficar complexo, podemos usar a mesma técnica novamente e passar o objeto *app* para outro módulo que por sua vez registrará mais rotas (embora estejamos entrando no terreno do “excesso de complexidade” – é preciso ter boas razões para justificar uma abordagem tão complicada!).

Renderizando views automaticamente

Se você alguma vez já quis voltar para a época em que podíamos simplesmente inserir um arquivo HTML em um diretório e – abracadabra! – o site o servia, tenha certeza de que não está sozinho. Se o site tiver muito conteúdo e pouca funcionalidade, pode ser uma dor de cabeça desnecessária adicionar uma rota para cada view. Felizmente, podemos resolver esse problema.

Digamos que você quisesse adicionar o arquivo *views/foo.handlebars* e torná-lo magicamente disponível na rota */foo*. Vejamos como poderíamos fazê-lo. Em nossa aplicação, imediatamente antes do manipulador de erro 404, adicione o middleware a seguir (*ch14/06-auto-views.js* no repositório fornecido):

```
const autoViews = {}
const fs = require('fs')
const { promisify } = require('util')
const fileExists = promisify(fs.exists)

app.use(async (req, res, next) => {
  const path = req.path.toLowerCase()
  // verifica o cache; se a view estiver presente,
  // ela será renderizada
```

```
if(autoViews[path]) return res.render(autoViews[path])
// se ela não estiver no cache, verifica se há
// um arquivo .handlebars correspondente
if(await fileExists(__dirname + '/views' + path + '.handlebars')) {
  autoViews[path] = path.replace(/^\//, "")
  return res.render(autoViews[path])
}
// a view não foi encontrada; passa para o manipulador de erro 404
next()
})
```

Agora podemos simplesmente adicionar um arquivo *.handlebars* ao diretório *view* e fazê-lo ser renderizado magicamente no path apropriado. É bom ressaltar que rotas comuns ignorarão esse mecanismo (porque inserimos o manipulador automático de views após todas as rotas). Logo, se você tiver uma rota que renderize uma view diferente para a rota */foo*, ela terá prioridade.

Essa abordagem não funcionará se você *excluir* uma view que tiver sido visitada; ela será adicionada ao objeto *autoViews*, portanto, as views subsequentes tentarão renderizá-la mesmo que tenha sido excluída, resultando em erro. O problema pode ser resolvido pelo encapsulamento da renderização em um bloco *try/catch* e a remoção da view de *autoViews* quando um erro for descoberto; deixarei essa melhoria como exercício para o leitor.

Conclusão

O roteamento é uma parte importante do projeto e há outras abordagens para a organização dos manipuladores de rota além das descritas aqui, logo, fique à vontade para fazer testes e encontrar uma técnica que funcione para você e seu projeto. Recomendo que escolha técnicas que sejam claras e fáceis de rastrear. O roteamento é como um mapa que vai do ambiente externo (o cliente, geralmente um navegador) até o código do lado do servidor que fornecerá a resposta. Se esse mapa for confuso, será difícil rastrear o fluxo de informações na aplicação, o que atrapalhará tanto o desenvolvimento quanto a depuração.

CAPÍTULO 15

APIs REST e JSON

Embora tenhamos visto alguns exemplos de API REST no Capítulo 8, nosso paradigma até agora foi em grande parte “processar os dados no lado do servidor e enviar HTML formatado para o cliente”. Esse está deixando de ser o modo de operação padrão para aplicações web. Em vez disso, as aplicações web mais modernas são aplicações de página única (SPAs) que recebem todo o HTML e CSS em um único pacote estático e depois obtêm dados não estruturados no formato JSON e manipulam o HTML diretamente. Da mesma forma, a importância de postar formulários para informar alterações para o servidor está sendo substituída pela comunicação direta com o uso de requisições HTTP para uma API.

É hora de usarmos o Express para o fornecimento de endpoints de API em vez de HTML pré-formatado. Isso nos ajudará no Capítulo 16, quando demonstrarmos como a API pode ser usada para renderizar uma aplicação dinamicamente.

Neste capítulo, despojaremos nossa aplicação para que forneça uma “futura” interface HTML: ela será concluída no Capítulo 16. Enfocaremos então a API que fornecerá acesso ao banco de dados de pacotes de férias e daremos suporte de API para o registro de ouvintes de férias “fora da temporada”.

Web service é um termo genérico para qualquer interface de programação de aplicação (API, application programming interface) que possa ser acessada com o uso do HTTP. O conceito de web service já existe há algum tempo, mas até recentemente as tecnologias que possibilitavam seu uso eram confusas, ultrapassadas e muito complicadas. Ainda há sistemas que empregam essas tecnologias (como SOAP e WSDL), e existem pacotes Node que podem ajudá-lo a interagir com elas. No entanto, não vamos examiná-las. Em vez disso, daremos ênfase ao fornecimento dos chamados serviços

RESTful, que têm interação muito mais fácil.

O acrônimo *REST* é a abreviação de *representational state transfer*, e o termo gramaticalmente confuso *RESTful* é usado como adjetivo para descrever um web service que atenda aos princípios da arquitetura REST. A descrição formal do REST é complicada e cheia das convenções da ciência da computação, mas o que importa é que o REST é uma conexão stateless entre um cliente e um servidor. A definição formal também especifica que o serviço pode ser armazenado em cache e que vários serviços podem ser dispostos em camadas (isto é, quando você estiver usando uma API REST, podem existir outras APIs REST abaixo dela).

Do ponto de vista prático, na verdade as restrições do HTTP dificultam a criação de uma API que não seja RESTful; seria preciso nos desviarmos de nosso caminho para estabelecer o estado, por exemplo. Logo, nosso trabalho será em grande parte abreviado.

JSON e XML

Falar uma linguagem comum é vital para o fornecimento de uma API. Parte da comunicação é imposta: devemos usar métodos HTTP para a comunicação com o servidor. No entanto, fora isso, podemos usar a linguagem de dados que quisermos. Tradicionalmente, o XML tem sido uma opção popular, e permanece sendo uma linguagem de marcação importante. Embora o XML não seja particularmente complicado, Douglas Crockford percebeu que poderia existir algo mais leve, e o JavaScript Object Notation (JSON) nasceu. Além de ser compatível com JavaScript (mesmo sendo proprietário; é um formato fácil para qualquer linguagem analisar), também apresenta a vantagem de ser mais fácil de escrever manualmente que XML.

Prefiro JSON a XML para a maioria das aplicações: há um suporte melhor para JavaScript e é um formato mais simples e compacto. Recomendo usar JSON e fornecer XML somente se os sistemas existentes o demandarem para a comunicação com o aplicativo.

Nossa API

Planejaremos a API antes de começar a implementá-la. Além da listagem de pacotes de férias e do registro para o recebimento de notificações de “abertura de temporada”, adicionaremos um endpoint de “exclusão de pacote de férias”. Já que essa é uma API pública, não excluiremos realmente o pacote de férias. Vamos apenas marcá-lo como “exclusão solicitada” para que o administrador possa revisar. Por exemplo, você poderia usar esse endpoint não protegido para permitir que os vendedores solicitem a remoção do pacote de férias do site, o que poderá então ser analisado posteriormente pelo administrador. Aqui estão os endpoints da API:

GET /api/vacations

Recupera pacotes de férias

GET /api/vacation/:sku

Retorna um pacote de férias por sua SKU

POST /api/vacation/:sku/notify-when-in-season

Recebe email como parâmetro de querystring e adiciona um ouvinte de notificação para o pacote de férias especificado

DELETE /api/vacation/:sku

Solicita a exclusão de um pacote de férias; recebe email (a pessoa que está solicitando a exclusão) e notes como parâmetros de querystring

[NOTE]

Example 15-1.

Há muitos verbos HTTP disponíveis. GET e POST são os mais comuns, seguidos por DELETE e PUT. Tornou-se um padrão o uso de POST para a *criação* de algo e PUT para a *atualização* (ou modificação). O significado dessas palavras em inglês não sustenta essa distinção, logo, se quiser, use o path para diferenciar essas duas operações e evitar confusão. Para obter mais informações sobre os verbos HTTP, recomendo começar com este *artigo de Tamas Piros* (<http://bit.ly/32L4QWt>).

Poderíamos ter descrito nossa API de várias maneiras. Aqui, optamos por usar combinações de métodos HTTP e paths para diferenciar as chamadas de

API, e combinações de parâmetros de querystring e de corpo (body) para passar dados. Como alternativa, poderíamos ter diferentes paths (como `/api/vacations/delete`) com o mesmo método.¹ Também poderíamos ter passado dados de maneira mais consistente. Por exemplo, optar por passar todas as informações necessárias para a recuperação de parâmetros na URL em vez de usar uma querystring: `DEL /api/vacation/:id/:email/:notes`. Para evitar URLs excessivamente longas, recomendo usar o corpo da requisição para passar blocos de dados maiores (como no caso das notas de solicitação de exclusão).



Há uma convenção popular e respeitada para as APIs JSON, criativamente chamada de JSON:API. Na minha opinião, ela é um pouco verbosa e repetitiva, mas também acho que um padrão imperfeito é melhor do que nenhum padrão. Embora não estejamos usando a JSON:API neste livro, você aprenderá tudo que é necessário para adotar as convenções definidas por ela. Consulte a *home page da JSON:API* (<https://jsonapi.org>) para obter mais informações.

Relato de erros da API

Geralmente o relato de erros em APIs HTTP é feito por códigos de status HTTP. Se a requisição retornar 200 (OK), o cliente saberá que ela foi bem-sucedida. Se retornar 500 (Internal Server Error), ela terá falhado. No entanto, na maioria das aplicações, nem tudo pode (ou deve) ser categorizado simplesmente como “sucesso” ou “falha”. Por exemplo, e se você requisitasse algo pelo seu ID, mas esse ID não existisse? Isso não representa um erro do servidor. O cliente solicitou algo que não existe. Normalmente, os erros podem ser agrupados nas seguintes categorias:

Erros catastróficos

Erros que resultam em um estado instável ou desconhecido do servidor. Geralmente, são originados por uma exceção não manipulada. A única maneira segura de se recuperar de um erro catastrófico é reiniciar o servidor. Idealmente, qualquer requisição pendente receberia um código de resposta 500, mas, se a falha for suficientemente grave, o servidor pode não

conseguir responder, e o tempo da requisição expirará.

Erros recuperáveis do servidor

Os erros recuperáveis não demandam a reinicialização do servidor ou qualquer outra ação heroica. O erro é resultado de uma condição inesperada no servidor (por exemplo, uma conexão de banco de dados que está indisponível). O problema pode ser temporário ou permanente. Um código de resposta 500 é apropriado nessa situação.

Erros do cliente

Ocorrem quando o cliente comete o erro – geralmente por parâmetros ausentes ou inválidos. Não é apropriado usar um código de resposta 500. Afinal, o servidor não falhou. Tudo está funcionando normalmente; o cliente apenas não está usando a API corretamente. Temos algumas opções aqui: você poderia responder com um código de status 200 e descrever o erro no corpo da resposta ou tentar descrever o erro com um código de status HTTP apropriado. Recomendo a segunda abordagem. Os códigos de resposta mais úteis nesse caso são 404 (Not Found), 400 (Bad Request) e 401 (Unauthorized). O corpo da resposta também deve conter uma explicação das particularidades do erro. Se quiséssemos ir além, a mensagem de erro ainda conteria um link da documentação. Observe que, se o usuário requisitar uma lista de coisas e não houver nada a ser retornado, essa não é uma condição de erro. É apropriado simplesmente retornar uma lista vazia.

Em nossa aplicação, usaremos uma combinação de códigos de resposta HTTP e mensagens de erro no corpo.

Compartilhamento de recursos de origem cruzada

Se você estiver publicando uma API, vai querer torná-la disponível para outras pessoas. Isso resultará em uma *requisição HTTP entre sites* (cross-site HTTP request). As requisições HTTP entre sites foram alvo de muitos ataques e, portanto, foram restringidas pela *política de mesma origem*, que limita a partir de onde os scripts podem ser carregados. Especificamente, o

protocolo, o domínio e a porta devem coincidir. Isso impossibilita que a API seja usada por outro site, e é aí que o compartilhamento de recursos de origem cruzada (CORS, cross-origin resource sharing) entra em cena. O CORS permite suspender essa restrição conforme o caso, possibilitando até mesmo listar que domínios têm autorização para acessar o script. Ele é implementado pelo cabeçalho `Access-Control-Allow-Origin`. A maneira mais fácil de implementá-lo em uma aplicação Express é usando o pacote `cors` (`npm install cors`). Para ativar o CORS em sua aplicação, use o seguinte:

```
const cors = require('cors')
```

```
app.use(cors())
```

Já que a API de mesma origem existe por uma razão (prevenir ataques), recomendo aplicar o CORS somente onde necessário. Em nosso caso, queremos expor a API inteira (mas só a API), logo, restringiremos o CORS a paths que comecem com `/api`:

```
const cors = require('cors')
```

```
app.use('/api', cors())
```

Consulte a *documentação do pacote* para obter informações sobre o uso mais avançado do CORS.

Nossos testes

Se usarmos verbos HTTP diferentes de GET, pode ser difícil testar a API, já que os navegadores só sabem como emitir requisições GET (e requisições POST para formulários). Há maneiras de resolver isso, como com a excelente aplicação *Postman* (<https://www.getpostman.com>). No entanto, usando ou não esse utilitário, é bom termos testes automatizados. Antes de criar testes para a API, precisamos de um modo de *chamar* uma API REST. Para fazê-lo, usaremos um pacote Node chamado `node-fetch`, que replica a API *fetch* do navegador:

```
npm install --save-dev node-fetch@2.6.0
```

Inseriremos os testes das chamadas de API que implementaremos em `tests/api/api.test.js` (`ch15/test/api/api.test.js` no repositório fornecido):

```
const fetch = require('node-fetch')
```

```
const baseUrl = 'http://localhost:3000'
```

```
const _fetch = async (method, path, body) => {  
  body = typeof body === 'string' ? body : JSON.stringify(body)  
  const headers = { 'Content-Type': 'application/json' }  
  const res = await fetch(baseUrl + path, { method, body, headers })  
  if(res.status < 200 || res.status > 299)  
    throw new Error(`API returned status ${res.status}`)  
  return res.json()  
}
```

```
describe('API tests', () => {
```

```
  test('GET /api/vacations', async () => {  
    const vacations = await _fetch('get', '/api/vacations')  
    expect(vacations.length).not.toBe(0)  
    const vacation0 = vacations[0]  
    expect(vacation0.name).toMatch(/w/)  
    expect(typeof vacation0.price).toBe('number')  
  })
```

```
  test('GET /api/vacation/:sku', async() => {  
    const vacations = await _fetch('get', '/api/vacations')  
    expect(vacations.length).not.toBe(0)  
    const vacation0 = vacations[0]  
    const vacation = await _fetch('get', '/api/vacation/' + vacation0.sku)  
    expect(vacation.name).toBe(vacation0.name)  
  })
```

```
  test('POST /api/vacation/:sku/notify-when-in-season', async() => {  
    const vacations = await _fetch('get', '/api/vacations')  
    expect(vacations.length).not.toBe(0)  
    const vacation0 = vacations[0]  
    // nesse momento, só precisamos verificar  
    // se a requisição HTTP foi bem-sucedida  
    await _fetch('post', `/api/vacation/${vacation0.sku}/notify-when-in-season`,  
      { email: 'test@meadowlarktravel.com' })
```

```
}}
```

```
test('DELETE /api/vacation/:id', async() => {  
  const vacations = await _fetch('get', '/api/vacations')  
  expect(vacations.length).not.toBe(0)  
  const vacation0 = vacations[0]  
  // nesse momento, só precisamos verificar  
  // se a requisição HTTP foi bem-sucedida  
  await _fetch('delete', `/api/vacation/${vacation0.sku}`)  
})
```

```
}}
```

Nossa suíte de testes começa com uma função auxiliar `_fetch`, que manipula algumas tarefas de preparação (housekeeping) comuns. Ela codificará o corpo em JSON se ele já não estiver nesse formato, adicionará os cabeçalhos apropriados e lançará um erro adequado se o código de status da resposta não estiver no intervalo dos códigos 200.

Temos um único teste para cada endpoint da API. Não estou sugerindo que esses testes sejam robustos ou completos; mesmo com uma API simples, poderíamos (e deveríamos) ter vários testes para cada endpoint. O que temos aqui está mais para um ponto de partida que ilustra técnicas para o teste de uma API.

Há algumas características importantes desses testes que merecem menção. Uma é que estamos presumindo que a API já tenha sido iniciada e esteja sendo executada na porta 3000. Uma suíte de testes mais robusta encontraria uma porta aberta, iniciaria a API nessa porta como parte de sua configuração, e a interromperia quando todos os testes tivessem sido executados. Em segundo lugar, esse teste presume que os dados já estejam presentes na API. Por exemplo, o primeiro teste espera que haja pelo menos um pacote de férias e que ele tenha um nome e um preço. Em uma aplicação real, talvez não possamos fazer essas suposições (poderíamos começar sem dados e querer verificar se a quantidade de dados ausentes é a permitida). Novamente, um framework de teste mais robusto teria uma maneira de definirmos e redefinirmos os dados iniciais na API para podermos sempre começar com um estado conhecido. Por exemplo, poderíamos ter scripts que configurassem

e alimentassem um banco de dados de teste, anexassem a API a ele e a desconectassem a cada execução do teste. Como vimos no Capítulo 5, a execução de testes é um tópico extenso e complicado e aqui podemos examiná-la apenas superficialmente.

O primeiro teste aborda nosso endpoint `GET /api/vacations`. Ele busca todos os pacotes de férias, valida se há pelo menos um e verifica o primeiro para ver se tem nome e preço. Também poderíamos testar outras propriedades de dados. Deixarei como exercício para o leitor considerar a que propriedades devemos dar mais importância nos testes.

O segundo teste aborda o endpoint `GET /api/vacation/:sku`. Já que não temos dados de teste consistentes, começamos buscando todos os pacotes de férias e obtendo a SKU do primeiro para poder testar esse endpoint.

Nossos dois últimos testes abordam os endpoints `POST /api/vacation/:sku/notify-when-in-season` e `DELETE /api/vacation/:sku`. Infelizmente, com a API e o framework de teste atuais, podemos fazer muito pouco para verificar se esses endpoints estão agindo como deveriam, logo, adotamos como padrão chamá-los e supor que a API esteja fazendo a coisa certa quando ela não retorna um erro. Se quiséssemos tornar esses testes mais robustos, teríamos de adicionar endpoints que nos permitissem verificar as ações (por exemplo, um endpoint que determinasse se um email específico foi registrado para um pacote de férias em particular) ou dar de alguma forma acesso “backdoor” (porta dos fundos) a nosso banco de dados.

Se você executar os testes agora, seu tempo expirará e eles falharão... porque não implementamos nossa API e nem mesmo iniciamos o servidor. Então, é hora de começar!

Usando o Express para fornecer uma API

O Express tem recursos para fornecer uma API. Há vários módulos npm disponíveis que oferecem funcionalidades úteis (examine `connect-rest` e `json-api`, por exemplo), mas acredito que o Express seja perfeitamente capaz de fazê-lo por conta própria e definiremos uma implementação criada somente com ele.

Começaremos criando os manipuladores em *lib/handlers.js* (poderíamos criar um arquivo separado, como *lib/api.js*, mas por enquanto queremos manter a simplicidade):

```
exports.getVacationsApi = async (req, res) => {
  const vacations = await db.getVacations({ available: true })
  res.json(vacations)
}

exports.getVacationBySkuApi = async (req, res) => {
  const vacation = await db.getVacationBySku(req.params.sku)
  res.json(vacation)
}

exports.addVacationInSeasonListenerApi = async (req, res) => {
  await db.addVacationInSeasonListener(req.params.sku, req.body.email)
  res.json({ message: 'success' })
}

exports.requestDeleteVacationApi = async (req, res) => {
  const { email, notes } = req.body
  res.status(500).json({ message: 'not yet implemented' })
}
```

Em seguida, incluiremos a API em *meadowlark.js*:

```
app.get('/api/vacations', handlers.getVacationsApi)
app.get('/api/vacation/:sku', handlers.getVacationBySkuApi)
app.post('/api/vacation/:sku/notify-when-in-season',
  handlers.addVacationInSeasonListenerApi)
app.delete('/api/vacation/:sku', handlers.requestDeleteVacationApi)
```

A essa altura nada do que fizemos deve ser novidade. Observe que estamos usando nossa camada de abstração de banco de dados, logo, não importa se será utilizada a implementação do MongoDB ou do Postgresql (mas, dependendo da implementação, você terá menos campos irrelevantes adicionais, que poderemos remover se necessário).

Estou deixando `requestDeleteVacationsApi` como exercício para o leitor, principalmente porque essa funcionalidade pode ser implementada de várias maneiras. A abordagem mais simples seria apenas modificar nosso esquema

de pacotes de férias para termos campos de “exclusão solicitada” que fossem atualizados com o email e as notas quando a API fosse chamada. Uma abordagem mais sofisticada seria termos uma tabela própria, como uma fila de moderação, que registrasse as solicitações de exclusão separadamente, referenciando o pacote de férias em questão, o que seria mais apropriado para uso do administrador.

Supondo que você tenha configurado o Jest corretamente no Capítulo 5, deve conseguir executar `npm test`, e os testes da API serão selecionados (o Jest procurará qualquer arquivo que termine com `.test.js`). Você verá que temos três testes que serão bem-sucedidos e um que falhará: o do endpoint `DELETE /api/vacation/:sku` incompleto.

Conclusão

Espero que este capítulo tenha deixado você com a dúvida: “Isso é tudo?”. Já deve ter percebido que a principal função do Express é responder a requisições HTTP. Para que servirão as requisições – e como elas responderão – é uma decisão inteiramente sua. Precisarão responder com HTML? Com CSS? Com texto puro? JSON? Tudo isso é fácil de fazer com o Express. Você poderia responder até mesmo com tipos de arquivos binários. Por exemplo, não seria difícil construir e retornar imagens dinamicamente. Nesse aspecto, uma API é apenas uma das várias maneiras pelas quais o Express pode responder.

No próximo capítulo, faremos uso dessa API construindo uma aplicação de página única e replicaremos o que fizemos em capítulos anteriores de maneira diferente.

¹ Se seu cliente não puder usar diferentes métodos HTTP, consulte *esse módulo* (<http://bit.ly/2O7nr9E>), que permite “simular” métodos distintos.

Aplicações de página única

O termo *aplicação de página única* (SPA) não é muito adequado, ou pelo menos é dúbio e confunde dois significados da palavra “página”. Do ponto de vista do usuário, os SPAs podem parecer (e geralmente parecem) ter páginas diferentes: a home page, a página Vacations, a página About e assim por diante. No entanto, você poderia criar uma aplicação tradicional renderizada no lado do servidor e um SPA que fossem indistinguíveis para o usuário.

A expressão “página única” tem mais a ver com onde e como o HTML é construído do que com a experiência do usuário. Em um SPA, o servidor distribui um único pacote HTML quando o usuário carrega a aplicação pela primeira vez,¹ e qualquer alteração na UI (que pode aparecer na forma de diferentes páginas para o usuário) é resultado de código JavaScript manipulando o DOM em resposta à atividade do usuário ou a eventos de rede.

Os SPAs têm de se comunicar frequentemente com o servidor, mas em geral o HTML é enviado como parte da primeira requisição. Depois disso, só dados JSON e arquivos estáticos são transferidos entre o cliente e o servidor.

Para entendermos a razão dessa abordagem de desenvolvimento de aplicações web agora dominante é preciso um pequeno histórico...

Breve história do desenvolvimento das aplicações web

A maneira de conduzirmos o desenvolvimento web passou por uma grande mudança nos últimos 10 anos, mas uma coisa permaneceu relativamente igual: os componentes envolvidos em um site ou aplicação web. Eles são:

- HTML e o Document Object Model (DOM)
- JavaScript

- CSS
- Arquivos estáticos (geralmente multimídia: imagens e vídeos etc.).

Quando reunidos por um navegador, são esses componentes que fornecem a experiência do usuário.

No entanto, a *maneira* de essa experiência ser construída começou a mudar drasticamente por volta de 2012. Atualmente, o paradigma dominante para o desenvolvimento web são as *aplicações de página única*, ou SPAs.

Para entender os SPAs, temos de saber com o que poderíamos compará-los, logo, voltaremos mais ainda no tempo, para 1998, o ano antes de o termo “Web 2.0” ser pronunciado pela primeira vez, e oito anos antes do jQuery ser introduzido.

Em 1998, o método predominante para a distribuição de aplicações web consistia em servidores web enviando HTML, CSS, JavaScript e arquivos multimídia *em resposta a cada requisição*. Suponhamos que você estivesse assistindo à TV e quisesse mudar de canal. Como equivalente metafórico, teria de jogar a TV fora, comprar outra, trazê-la para casa e instalá-la – só para mudar de canal (navegar para uma página diferente, ainda que no mesmo site).

O problema dessa abordagem é que há muito overhead envolvido. Às vezes o HTML – ou grandes blocos dele – não apresentava nenhuma mudança. O CSS mudava menos ainda. Os navegadores reduziam um pouco o custo do overhead armazenando arquivos em cache, mas a velocidade das inovações trazidas para as aplicações web estava exaurindo esse modelo.

Em 1999, o termo “Web 2.0” foi cunhado para tentar descrever a riqueza na experiência que as pessoas estavam começando a esperar dos sites. Os anos entre 1999 e 2012 viram avanços tecnológicos que criaram a base dos SPAs.

Desenvolvedores web inteligentes começaram a perceber que, se quisessem manter seus usuários interessados, o overhead de carregar o site inteiro sempre que eles desejassem (metaforicamente) mudar de canal era inaceitável. Esses desenvolvedores notaram que nem toda mudança em uma aplicação requeria informações do servidor, e nem toda mudança que demandava informações do servidor precisava da aplicação inteira só para a

distribuição de uma pequena alteração.

Nesse período de 1999 a 2012, as páginas geralmente ainda eram páginas: quando acessávamos um site pela primeira vez, recebíamos o HTML, o CSS e os arquivos estáticos. Quando navegávamos para outra página, obtínhamos um HTML diferente, arquivos estáticos distintos e às vezes CSS diferente. No entanto, a cada alternância de página, a própria página podia mudar em resposta à interação do usuário, e, em vez de solicitar ao servidor uma aplicação totalmente nova, o JavaScript alterava o DOM diretamente. Quando informações precisavam ser buscadas no servidor, elas eram enviadas em XML ou JSON, sem todo o HTML correspondente. Mais uma vez, ficava a cargo do JavaScript interpretar os dados e alterar a interface de usuário de maneira apropriada. Em 2006, o jQuery foi introduzido, o que diminuiu muito o trabalho de manipulação do DOM e de tratamento de requisições de rede.

Muitas dessas mudanças estavam sendo conduzidas pelo crescente poder dos computadores e – por extensão – dos navegadores, e os desenvolvedores web descobriram que uma parte cada vez maior do trabalho de fazer um site ou uma aplicação web ter boa aparência poderia ser realizada diretamente no computador do usuário em vez de no servidor para então ser enviada para o usuário.

Essa mudança na abordagem aumentou de ritmo no fim dos anos 2000, quando os smartphones foram introduzidos. Agora, não só os navegadores podiam se envolver mais no trabalho, como as pessoas queriam acessar aplicações web *por redes wireless*. Repentinamente, o custo do overhead de enviar dados subiu, tornando ainda mais interessante fazer o menor número de transferências pela rede, e permitir que o navegador se encarregasse da maior parcela de trabalho possível.

Em 2012, era prática comum tentar enviar o menor número de informações possível pela rede, e fazer o máximo no navegador. Como a sopa primordial que fez surgir a primeira forma de vida, esse rico ambiente forneceu as condições para a evolução natural dessa técnica: a aplicação de página única.

A ideia é simples: seja qual for a aplicação web, o HTML, o JavaScript e o CSS (se houver) são enviados *apenas uma vez*. Quando o navegador está de

posse do HTML, fica a cargo do JavaScript realizar todas as alterações no DOM para fazer o usuário achar que está navegando para uma página diferente. O servidor não precisa mais enviar HTML diferente quando navegamos da home page para a página Vacations, por exemplo.

É claro que o servidor ainda está envolvido: ele é responsável pelo fornecimento de dados atualizados e é a “fonte única da verdade” em uma aplicação multiusuário. No entanto, em uma arquitetura SPA, a maneira como a aplicação aparece para o usuário não é mais responsabilidade do servidor: é tarefa do JavaScript e dos frameworks que criam essa ilusão inteligente.

Embora geralmente o Angular seja considerado o primeiro framework SPA, vários outros se juntaram a ele: o React, o Vue e o Ember são os de maior destaque entre os rivais do Angular.

Se você é iniciante na área de desenvolvimento, os SPAs talvez sejam seu único referencial, o que torna esse relato apenas uma história interessante. Porém, se é veterano, pode achar a mudança confusa e incômoda. Independentemente do grupo em que se enquadrar, este capítulo foi planejado para ajudá-lo a entender como as aplicações web são distribuídas como SPAs e que papel o Express desempenha nesse cenário.

Essa história é relevante para o Express porque o papel do servidor mudou durante a alteração nas técnicas de desenvolvimento web. Quando a primeira edição deste livro foi publicada, o Express ainda era usado para servir aplicações multipáginas (junto com as APIs que suportavam a funcionalidade de tipo Web 2.0). Agora ele é quase totalmente usado para servir SPAs, servidores de desenvolvimento, e APIs, refletindo a natureza mutante do desenvolvimento web.

O interessante é que ainda há razões válidas para uma aplicação web servir uma página específica (em vez da página “genérica”, que será reformatada pelo navegador). Embora possa parecer que estamos andando em círculos, ou jogando fora os benefícios dos SPAs, a técnica de como fazer isso melhor espelha a arquitetura destes. Chamada *renderização no lado do servidor* (SSR, server-side rendering), ela permite que os servidores usem o mesmo código que o navegador utiliza para criar páginas individuais a fim de

melhorar o carregamento da primeira página. A chave aqui é que o servidor não tem de se esforçar muito: ele apenas usa as mesmas técnicas do navegador para gerar uma página específica. Geralmente esse tipo de SSR é usado para aprimorar a experiência de carregamento da primeira página e dar suporte à otimização do mecanismo de busca. Trata-se de um tópico mais avançado que não abordaremos, mas é preciso que você saiba que essa prática existe.

Agora que conhecemos um pouco como e por que os SPAs surgiram, examinaremos os frameworks SPA que estão disponíveis atualmente.

Tecnologias SPA

Hoje em dia há muitas opções de tecnologias SPA:

React

Por enquanto, o React parece ser o mestre do universo SPA, embora de um lado haja grandes pioneiros (Angular) e do outro usurpadores ambiciosos (Vue). Em algum momento de 2018, ele superou o Angular nas estatísticas de utilização. O React é uma biblioteca open source, mas começou como um projeto do Facebook, que ainda é um ativo colaborador. Usaremos o React na refatoração da Meadowlark Travel.

Angular

Considerado por muitos o SPA “original”, o Angular do Google tornou-se muito popular, mas acabou sendo superado pelo React. No fim de 2014, o Angular anunciou a versão 2, que foi uma grande mudança na primeira versão, desagradando a muitos usuários antigos e assustando os novos. Acho que essa mudança (embora provavelmente necessária) contribuiu para o React acabar superando o Angular. Outra razão é que o Angular é um framework muito maior que o React. Isso tem vantagens e desvantagens: o Angular fornece uma arquitetura muito mais completa para a construção de aplicações inteiras, e há sempre uma “maneira fácil típica do Angular” de fazer as coisas, enquanto frameworks como o React e o Vue deixam mais detalhes a cargo da escolha pessoal e da criatividade. Independentemente de

qual é a abordagem melhor, os frameworks maiores são mais pesados e demoram mais para evoluir, o que deu ao React uma vantagem em termos de inovação.

Vue.js

Pretensioso desafiador do React, e idealizado por um único desenvolvedor, Evan You. Em tempo surpreendentemente curto, ganhou um número impressionante de seguidores, e é bastante respeitado por seus adeptos, mas ainda está muito atrás da popularidade desgovernada do React. Fiz algumas experiências com o Vue, e aprecio sua documentação clara e abordagem level, mas acabo preferindo a arquitetura e a filosofia do React.

Ember

Como o Angular, o Ember oferece um framework de aplicações abrangente. Há uma comunidade de desenvolvimento grande e ativa, e, embora ele não seja tão inovador quanto o React ou o Vue, fornece muitas funcionalidades e clareza. Descobri que com certeza prefiro frameworks mais leves, e é por isso que adotei o React.

Polymer

Não tenho experiência no uso do Polymer, mas ele é apoiado pelo Google, o que lhe dá credibilidade. Parece haver certa curiosidade com o que o Polymer está trazendo de novo, mas não tenho visto muitas pessoas correndo para adotá-lo.

Se você está procurando um framework robusto e pronto para uso, e não se importa em seguir convenções, deve considerar o Angular ou o Ember. Se quiser espaço para se expressar criativamente e inovar, recomendo o React ou o Vue. Ainda não sei onde o Polymer se enquadra, mas vale a pena prestar atenção nele.

Agora que vimos os concorrentes, daremos prosseguimento com o React e refatoraremos a Meadowlark Travel como um SPA!

Criando um aplicativo React

A melhor maneira de começar a trabalhar em um aplicativo React é usando o utilitário create-react-app (CRA), que cria todo o boilerplate e as ferramentas de desenvolvedor e fornece uma aplicação inicial mínima como base. Além disso, create-react-app manterá a configuração atualizada para que você possa se concentrar na construção da aplicação, e não nas ferramentas de desenvolvedor. Dito isso, se você chegar a um ponto em que precisar configurar suas ferramentas, poderá “ejetar” a aplicação: perderá a habilidade de se manter atualizado com as últimas ferramentas do CRA, mas terá controle total sobre a configuração da aplicação.

Ao contrário do que fizemos até agora, com todos os artefatos convivendo lado a lado com a aplicação Express, os SPAs devem ser considerados como uma aplicação independente e totalmente separada. Logo, teremos *duas* raízes de aplicação em vez de uma. Para deixar claro, quando me referir ao diretório em que a aplicação Express reside, usarei *raiz do servidor*, e para o diretório em que a aplicação React reside, usarei *raiz do cliente*. A *raiz da aplicação* é onde esses dois diretórios residem.

Vá até a raiz da aplicação e crie um diretório chamado *server*; é nele que o servidor Express residirá. Não crie um diretório para o aplicativo cliente; o CRA fará isso para nós.

Antes de executar o CRA, devemos instalar o Yarn (<https://yarnpkg.com>). O Yarn é um gerenciador de pacotes como o npm... na verdade, o Yarn está mais para um substituto casual do npm. Ele não é obrigatório para o desenvolvimento com o React, mas é o padrão usado, e não o utilizar seria como nadar contra a corrente. Há algumas pequenas diferenças de utilização entre o Yarn e o npm, porém a única que você deve notar é que terá de executar `yarn add` em vez de `npm install`. Para instalar o Yarn, apenas siga *suas instruções de instalação* (<http://bit.ly/2xHZ2Cx>).

Após instalar o Yarn, execute o seguinte a partir da raiz da aplicação:

```
yarn create react-app client
```

Agora vá até o diretório da aplicação e digite `yarn start`. Após alguns segundos, você verá uma nova janela de navegador abrir, com o aplicativo React sendo executado nela!

Deixe a janela do terminal sendo executada. O CRA dá um suporte muito

bom ao “hot reloading”, logo, quando você fizer alterações no código-fonte, ele será compilado com *muita* rapidez e o navegador será recarregado automaticamente. Após se acostumar, você não conseguirá viver sem isso.

Aspectos básicos do React

O React tem uma ótima documentação, que não vou recriar aqui. Então, se você for iniciante, comece com o tutorial *Intro to React* (<http://bit.ly/36VdKUq>) e, em seguida, examine o guia *Main Concepts* (<http://bit.ly/2KgT939>).

Você verá que o React foi organizado com base em *componentes*, que são seus principais blocos de construção. Geralmente tudo que o usuário vê ou com o qual interage é um componente no React. Examinaremos *client/src/App.js* (o conteúdo do seu arquivo pode diferir um pouco – o CRA muda com o tempo):

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
```

```
);  
}
```

```
export default App;
```

Um dos principais conceitos do React é o de que a UI é gerada por *funções*. E o componente mais simples do React é apenas uma função que retorna HTML, como vemos aqui. Você deve estar olhando para esse código e pensando que não é JavaScript válido; parece que há HTML envolvido! A verdade é um pouco mais complicada. Por padrão, o React habilita um superconjunto de JavaScript chamado JSX. O JSX permite criar algo parecido com HTML. Porém, não é *realmente* HTML; ele cria elementos do React, e a finalidade de um elemento do React é corresponder a um elemento do DOM.

No entanto, no fim das contas, podemos considerar como HTML. Aqui, App é uma função que renderizará o HTML correspondente ao JSX retornado.

Algumas coisas merecem menção: já que o JSX é parecido com – mas não é exatamente – HTML, há diferenças sutis. Você deve ter notado que usamos `className` em vez de `class`, e fizemos isso porque `class` é uma palavra reservada do JavaScript.

Tudo que você precisa fazer para especificar o HTML é iniciar um elemento HTML em qualquer local em que uma expressão for esperada. Também é possível “voltar ao” JavaScript com o uso de chaves dentro do HTML. Por exemplo:

```
const value = Math.floor(Math.random()*6) + 1  
const html = <div>You rolled a {value}!</div>
```

Nesse exemplo, o elemento `<div>` inicia o HTML, e as chaves ao redor de `value` nos retornam para o JavaScript para fornecer o número armazenado aí. Seria igualmente fácil inserir o cálculo inline:

```
const html = <div>You rolled a {Math.floor(Math.random()*6) + 1}!</div>
```

Qualquer expressão JavaScript válida pode ser inserida em chaves dentro do JSX – inclusive outros elementos HTML! Um caso de uso comum é a renderização de listas:

```
const colors = ['red', 'green', 'blue']
```

```
const html = (  
  <ul>  
    {colors.map(color =>  
      <li key={color}>{color}</li>  
    )}  
  </ul>  
)
```

Precisamos destacar alguns detalhes desse exemplo. Em primeiro lugar, observe que mapeamos nossas cores para poder retornar os elementos de ``. Isso é crucial: o JSX funciona apenas avaliando *expressões*. Logo, `` tem de conter uma expressão ou um array de expressões. Se você alterasse `map` para `forEach`, veria que os elementos de `` não seriam renderizados. Em segundo lugar, repare que os elementos de `` recebem uma propriedade `key`: essa é uma concessão relacionada ao desempenho. Para o React saber quando deve renderizar novamente os elementos de um array, ele precisa de uma chave exclusiva para cada elemento. Já que os elementos do array são exclusivos, usamos esse valor, mas normalmente usaríamos um ID ou – se não houvesse nada diferente disponível – o índice do item no array.

Sugiro que você faça testes com alguns dos exemplos do JSX em `client/src/App.js` antes de continuar. Se deixar `yarn start` sendo executado, sempre que você salvar suas alterações, elas serão refletidas automaticamente no navegador, o que deve acelerar seu ciclo de aprendizagem.

Temos mais um tópico a abordar antes de deixar os aspectos básicos do React para trás; trata-se do conceito de *estado*. Cada componente pode ter o próprio estado, o que significa basicamente que “os dados associados ao componente podem mudar”. Um carrinho de compras é um bom exemplo. O estado do componente de carrinho de compras contém uma lista de itens; à medida que adicionamos e removemos itens do carrinho, o estado do componente muda. Pode parecer um conceito muito simples ou óbvio, mas a maioria dos detalhes da criação de uma aplicação React se resume a projetar e gerenciar eficientemente o estado dos componentes. Veremos um exemplo de estado quando lidarmos com a página *Vacations*.

Daremos prosseguimento e criaremos a home page da Meadowlark Travel.

A Home Page

Você deve se lembrar de que nossas views do Handlebars tinham um arquivo de “layout” principal que estabelecia a aparência primária do site. Começaremos examinando o que entrou na tag `<body>` (exceto os scripts):

```
<div class="container">
  <header>
    <h1>Meadowlark Travel</h1>
    <a href="/"></a>
  </header>
  {{{body}}}
</div>
```

Vai ser muito fácil transformar esse código em um componente do React. Primeiro, copiaremos nosso logotipo no diretório *client/src*. Por que não no diretório *public*? Para itens gráficos pequenos ou normalmente usados, pode ser mais eficiente inseri-los inline no pacote JavaScript, e o empacotador fornecido pelo CRA deve tomar uma decisão inteligente sobre isso. O exemplo de aplicativo do CRA inseriu o logotipo diretamente no diretório *client/src*, mas prefiro reunir arquivos de imagens em um subdiretório, portanto, inseriremos o logotipo (*logo.png*) em *client/src/img/logo.png*.

Há apenas mais uma parte complicada: o que fazer com `{{{body}}}`. Em nossas views, é nesse local que outra view seria renderizada – o conteúdo da página específica em que estamos. Podemos replicar a mesma ideia básica no React. Já que todo o conteúdo é renderizado na forma de componentes, apenas renderizaremos outro componente aqui. Começaremos com um componente Home vazio e o construiremos em breve:

```
import React from 'react'
import logo from './img/logo.png'
import './App.css'

function Home() {
  return (<i>coming soon</i>)
}

function App() {
  return (
```

```

    <div className="container">
      <header>
        <h1>Meadowlark Travel</h1>
        <img src={logo} alt="Meadowlark Travel Logo" />
      </header>
      <Home />
    </div>
  )
}

```

```
export default App
```

Estamos usando a mesma abordagem que o exemplo de aplicativo para o CSS: podemos simplesmente criar um arquivo CSS e importá-lo. Assim, poderemos editar esse arquivo e aplicar os estilos necessários. Manteremos a simplicidade nesse exemplo, embora nada fundamental tenha mudado na maneira de estilizar o HTML com o CSS, logo, ainda temos todas as ferramentas com as quais nos acostumamos.



O CRA ativará o linting para nós, e, à medida que avançarmos neste capítulo, provavelmente veremos avisos (tanto na saída do terminal CRA quanto no console JavaScript do navegador). Isso acontecerá porque estamos adicionando coisas incrementalmente; quando chegarmos ao fim do capítulo, não deve haver mais avisos... se houver, verifique se não pulou alguma etapa! Você também pode verificar o repositório fornecido.

Roteamento

O conceito básico de roteamento que aprendemos no Capítulo 14 não mudou: ainda estamos usando o path de URL para determinar que parte da interface o usuário está vendo. A diferença é que é responsabilidade da aplicação cliente manipular isso. Alterar a UI com base na rota é responsabilidade do aplicativo cliente: se a navegação demandar dados novos ou atualizados do servidor, o aplicativo cliente deve solicitá-los.

Há várias opções para – e opiniões obstinadas sobre – o roteamento em aplicativos React. No entanto, uma biblioteca se sobressai: *React Router*

(<http://bit.ly/32GvAXK>). Há muitas coisas que não gosto no React Router, mas seu uso é tão comum que é impossível não deparar com ele. Além disso, é uma boa opção para se colocar algo básico em funcionamento, e, por essas duas razões, vamos usá-lo aqui.

Começaremos instalando a versão DOM do React Router (também há uma versão para React Native, para desenvolvimento móvel):

```
yarn add react-router-dom
```

Agora incluiremos o roteador, e adicionaremos uma página About e uma Not Found. Também vincularemos o logotipo do site novamente à home page:

```
import React from 'react'
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from 'react-router-dom'
import logo from './img/logo.png'
import './App.css'

function Home() {
  return (
    <div>
      <h2>Welcome to Meadowlark Travel</h2>
      <p>Check out our "<Link to="/about">About</Link>" page!</p>
    </div>
  )
}

function About() {
  return (<i>coming soon</i>)
}

function NotFound() {
  return (<i>Not Found</i>)
}

function App() {
```

```

return (
  <Router>
    <div className="container">
      <header>
        <h1>Meadowlark Travel</h1>
        <Link to="/"><img src={logo} alt="Meadowlark Travel Logo" /></Link>
      </header>
      <Switch>
        <Route path="/" exact component={Home} />
        <Route path="/about" exact component={About} />
        <Route component={NotFound} />
      </Switch>
    </div>
  </Router>
)
}

```

```
export default App
```

A primeira coisa a se mencionar é que estamos encapsulando nossa aplicação inteira em um componente `<Router>`. É isso que permite o roteamento, como era de se esperar. Dentro de `<Router>`, podemos usar `<Route>` para renderizar condicionalmente um componente de acordo com o path da URL. Inserimos as rotas de conteúdo dentro de um componente `<Switch>`, o que assegurará que só *um* dos componentes contidos aqui seja renderizado.

Há algumas diferenças sutis entre o roteamento que fizemos com o Express e o roteamento do React Router. No Express, renderizamos a página de acordo com a primeira ocorrência encontrada (ou exibimos a página 404 quando não encontramos nenhuma ocorrência). Com o React Router, o path é simplesmente uma “dica” que determina que combinação de componentes deve ser exibida. Nesse aspecto, ele é mais flexível que o roteamento com o Express. Portanto, por padrão as rotas do React Router se comportam como se tivessem um asterisco (*) no final. Isto é, a rota `/` encontraria *todas* as páginas (já que todas começam com uma barra inclinada). Usamos, então, a propriedade `exact` para fazer essa rota se comportar como uma rota do Express. Da mesma forma, sem a propriedade `exact`, a rota `/about` também encontraria `/about/contact`, que não é o que queremos. Para o roteamento do

conteúdo principal, é melhor que todas as rotas (exceto Not Found) tenham exact. Caso contrário, você terá de se certificar de organizá-las corretamente dentro de <Switch> para que sejam encontradas na ordem certa.

A segunda coisa digna de nota é o uso de <Link>. Você deve estar se perguntando por que não usamos simplesmente tags <a>. O problema das tags <a> é que – sem algum esforço adicional – o navegador as tratará obediamente como “indo para outro lugar”, ainda que seja no mesmo site; isso resultará em uma nova requisição HTTP sendo enviada para o servidor... e o HTML e o CSS serão baixados novamente, invalidando o que é preconizado pelo SPA. *Funcionará*, no sentido de que, quando a página for carregada, o React Router fará o que é certo, mas não será tão rápido ou eficiente, com chamadas a requisições de rede desnecessárias. Na verdade, ver a diferença é um exercício instrutivo que deve deixar clara a natureza dos SPAs. Como teste, crie dois elementos de navegação, um usando <Link> e o outro com <a>:

```
<Link to="/">Home (SPA)</Link>  
<a href="/">Home (reload)</Link>
```

Em seguida, abra suas ferramentas de desenvolvedor, abra a aba Network, remova o tráfego e clique em “Preserve log” (no Chrome). Agora clique no link “Home (SPA)” e repare que não há nenhum tráfego de rede. Clique no link “Home (reload)” e observe o tráfego de rede. Resumindo, essa é a natureza de um SPA.

Página Vacations – design visual

Até aqui construímos uma aplicação puramente front-end... mas onde o Express é necessário? Nosso servidor ainda é a fonte única da verdade. Especificamente, ele mantém o banco de dados de pacotes de férias que queremos exibir no site. Felizmente, já fizemos grande parte do trabalho no Capítulo 15: expusemos uma API que retornará os pacotes de férias em formato JSON, prontos para uso em uma aplicação React.

No entanto, antes de conectar essas duas coisas, construiremos nossa página Vacations. Não haverá pacotes de férias para renderizarmos, mas não deixaremos que isso nos detenha.

Na seção anterior, incluímos todas as páginas de conteúdo em *client/src/App.js*, o que geralmente é considerado uma prática inadequada: é mais convencional cada componente residir em seu próprio arquivo. Logo, aproveitaremos para inserir o componente *Vacations* em seu próprio arquivo. Crie o arquivo *client/src/Vacations.js*:

```
import React, { useState, useEffect } from 'react'
import { Link } from 'react-router-dom'

function Vacations() {
  const [vacations, setVacations] = useState([])
  return (
    <>
      <h2>Vacations</h2>
      <div className="vacations">
        {vacations.map(vacation =>
          <div key={vacation.sku}>
            <h3>{vacation.name}</h3>
            <p>{vacation.description}</p>
            <span className="price">{vacation.price}</span>
          </div>
        )}
      </div>
    </>
  )
}

export default Vacations
```

O que temos até agora é muito simples: estamos apenas retornando um `<div>` que contém elementos `<div>` adicionais, cada um representando um pacote de férias. Porém, de onde vem a variável `vacations`? Nesse exemplo, estamos usando um recurso mais recente do React, chamado *hooks*. Antes dos hooks, quando um componente precisava ter o próprio estado (nesse caso, uma lista de pacotes de férias), era necessário usar uma implementação de classe. Os hooks nos permitem usar componentes baseados em funções que tenham o próprio estado. Em nossa função `Vacations`, chamamos `useState` para definir o estado. Observe que passamos um array vazio para `useState`: esse será o valor

inicial de vacations para o estado (discutiremos como preenchê-lo em breve). O que setState retorna é um array contendo o valor do estado (vacations) e uma maneira de atualizá-lo.

Você deve estar se perguntando por que não podemos modificar vacations diretamente: já que estamos lidando com um array, não poderíamos chamar push para adicionar pacotes de férias a ele? Poderíamos, mas isso invalidaria o objetivo do sistema de gerenciamento de estado do React, que assegura consistência, desempenho e comunicação entre componentes.

Também deve estar querendo saber que componente é esse (`<>...</>`) no qual estão inseridos nossos pacotes de férias. Ele se chama *fragmento* (<http://bit.ly/2ryneVj>). O fragmento é necessário porque cada componente deve renderizar um único elemento. Em nosso caso, temos dois elementos, `<h2>` e `<div>`. O fragmento fornece um elemento raiz “transparente” para incluirmos esses dois elementos e podermos renderizar um único elemento.

Adicionaremos nosso componente Vacations à aplicação, mesmo ainda não havendo nenhum pacote de férias para ser exibido. Em *client/src/App.js*, primeiro importe a página de pacotes de férias:

```
import Vacations from './Vacations'
```

Agora só precisamos criar uma rota para ela no componente `<Switch>`:

```
<Switch>
  <Route path="/" exact component={Home} />
  <Route path="/about" exact component={About} />
  <Route path="/vacations" exact component={Vacations} />
  <Route component={NotFound} />
</Switch>
```

Salve esse código; sua aplicação deve ser carregada automaticamente e você poderá navegar para a página */vacations*, embora ainda não haja nada de interessante para ver. Como já estamos com grande parte da infraestrutura do cliente definida, voltaremos nossa atenção para a integração com o Express.

Página Vacations – integração com o servidor

Fizemos quase todo o trabalho necessário para a construção da página Vacations; temos um endpoint de API que acessa os pacotes de férias no

banco de dados e os retorna em formato JSON. Agora temos de descobrir como fazer o servidor e o cliente se comunicarem.

Podemos começar com o trabalho que realizamos no Capítulo 15; não é preciso adicionar nada a ele, mas podemos retirar algumas coisas que não são mais necessárias. Removeremos o seguinte:

- O suporte ao Handlebars e às views (deixaremos o middleware `static`, no entanto, por razões que veremos posteriormente).
- Os cookies e as sessões (nosso SPA ainda pode usar cookies, mas não precisa mais da ajuda do servidor... e nossa maneira de considerar as sessões mudou totalmente).
- Todas as rotas que renderizam uma view (mas é claro que manteremos as rotas da API).

Isso simplificará muito o servidor. Mas o que faremos com ele? A primeira coisa que temos de fazer é considerar o fato de que estivemos usando a porta 3000, e por padrão o servidor de desenvolvimento do CRA também usa a porta 3000. Poderíamos mudar, logo, sugiro arbitrariamente a mudança da porta do Express. Geralmente uso a porta 3033 – só porque gosto de como esse número soa. Você deve lembrar-se de que optamos pelo uso da porta padrão em *meadowlark.js*, portanto, basta mudá-la:

```
const port = process.env.PORT || 3033
```

É claro que poderíamos usar uma variável de ambiente para controlar a porta, mas, já que vamos usá-la com frequência junto com o servidor de desenvolvimento do SPA, também podemos mudar o código.

Agora que os dois servidores estão sendo executados, podemos fazer com que se comuniquem. Mas como? Em nosso aplicativo React, poderíamos fazer algo assim:

```
fetch('http://localhost:3033/api/vacations')
```

O problema dessa abordagem é que faremos requisições como essa em toda a aplicação... e agora estamos embutindo `http://localhost:3033` em todos os locais, o que não funcionará na produção, e pode não funcionar no computador do seu colega porque talvez ele precise usar portas diferentes, talvez a porta tenha de ser diferente para os servidores de teste e assim por diante. O uso

dessa abordagem pode trazer problemas de configuração. Sim, você poderia armazenar a URL base como uma variável para usar em qualquer lugar, mas existe uma maneira melhor.

Em um mundo perfeito, do ponto de vista da aplicação, tudo é obtido a partir do mesmo local: são os mesmos protocolo, host e porta acessando o HTML, os arquivos estáticos e a API. Simplifica muito as coisas e assegura a consistência no código-fonte. Se tudo estiver vindo a partir do mesmo local, você pode simplesmente omitir o protocolo, o host e a porta e chamar `fetch(/api/vacations)`. É uma bela abordagem, que felizmente é muito fácil de criar!

A configuração do CRA vêm com suporte a *proxy*, permitindo passar requisições web para a API. Edite o arquivo *client/package.json* e adicione o seguinte:

```
"proxy": "http://localhost:3033",
```

Essa linha pode ser adicionada onde você quiser. Geralmente a insiro entre "private" e "dependencies" só porque gosto de vê-la no início do arquivo. Agora – contanto que o servidor Express esteja sendo executado na porta 3033 – o servidor de desenvolvimento do CRA passará requisições de API por intermédio do servidor Express.

Já que a configuração está definida, usaremos um *efeito* (outro hook do React) para buscar e atualizar dados de férias. Aqui está o componente Vacations inteiro com o hook `useEffect`:

```
function Vacations() {  
  // define o estado  
  const [vacations, setVacations] = useState([])  
  
  // busca dados iniciais  
  useEffect(() => {  
    fetch('/api/vacations')  
      .then(res => res.json())  
      .then(setVacations)  
  }, [])  
  
  return (  

```

```

<>
  <h2>Vacations</h2>
  <div className="vacations">
    {vacations.map(vacation =>
      <div key={vacation.sku}>
        <h3>{vacation.name}</h3>
        <p>{vacation.description}</p>
        <span className="price">{vacation.price}</span>
      </div>
    )}
  </div>
</>
)
}

```

Como antes, `useState` está configurando o estado de nosso componente para que tenha um array `vacations`, com um setter o acompanhando. Adicionamos `useEffect`, que chama a API para recuperar pacotes de férias, e depois chama o setter assincronamente. Observe que passamos um array vazio como segundo argumento de `useEffect`; é um sinal para o React de que esse efeito só deve ser executado uma vez, quando o componente estiver montado. À primeira vista, pode parecer uma maneira estranha de sinalizar isso, mas, quando você aprender mais sobre os hooks, verá que na verdade é bastante consistente. Para aprender mais sobre os hooks, consulte a *documentação de hooks do React* (<http://bit.ly/34MGSeK>).

Os hooks são relativamente novos – foram adicionados na versão 16.8 em fevereiro de 2019 – logo, mesmo se você tiver experiência com o React, pode não estar familiarizado com eles. Acredito veementemente que os hooks são uma ótima inovação na arquitetura do React e, embora inicialmente pareçam estranhos, você verá que eles simplificam os componentes e reduzem alguns dos erros mais complicados relacionados ao estado que as pessoas costumam cometer.

Agora que aprendemos como recuperar dados no servidor, enviaremos informações na outra direção.

Enviando informações para o servidor

Já há um endpoint de API para fazermos alterações no servidor, portanto, temos um endpoint para o recebimento de um email quando uma temporada estiver ocorrendo novamente. Daremos prosseguimento e modificaremos nosso componente Vacations para exibir um formulário de inscrição em pacotes de férias que estejam fora da temporada. Conforme o padrão do React, criaremos dois componentes novos: dividiremos a view individual de pacotes de férias nos componentes Vacation e NotifyWhenInSeason. Poderíamos fazê-lo em um único componente, mas a abordagem recomendada para o desenvolvimento com o React é a de que haja muitos componentes de uso específico em vez de componentes multitarefa maiores (no entanto, para simplificar, vamos parar de inserir esses componentes em seus próprios arquivos: deixarei isso como exercício para o leitor):

```
import React, { useState, useEffect } from 'react'
```

```
function NotifyWhenInSeason({ sku }) {  
  return (  
    <>  
      <i>Notify me when this vacation is in season:</i>  
      <input type="email" placeholder="(your email)" />  
      <button>OK</button>  
    </>  
  )  
}
```

```
function Vacation({ vacation }) {  
  return (  
    <div key={vacation.sku}>  
      <h3>{vacation.name}</h3>  
      <p>{vacation.description}</p>  
      <span className="price">{vacation.price}</span>  
      {!vacation.inSeason &&  
        <div>  
          <p><i>This vacation is not currently in season.</i></p>  
          <NotifyWhenInSeason sku={vacation.sku} />  
        </div>  
      }  
    </div>  
  )  
}
```

```
)  
}
```

```
function Vacations() {  
  const [vacations, setVacations] = useState([])  
  useEffect(() => {  
    fetch('/api/vacations')  
      .then(res => res.json())  
      .then(setVacations)  
  }, [])  
  return (  
    <>  
    <h2>Vacations</h2>  
    <div className="vacations">  
      {vacations.map(vacation =>  
        <Vacation key={vacation.sku} vacation={vacation} />  
      )}  
    </div>  
  </>  
  )  
}
```

```
export default Vacations
```

Agora, se houver algum pacote de férias para o qual `inSeason` seja `false` (e haverá, se os scripts do banco de dados e de inicialização não forem alterados), você atualizará o formulário. Incluiremos o botão que fará a chamada de API. Modifique `NotifyWhenInSeason`:

```
function NotifyWhenInSeason({ sku }) {  
  const [registeredEmail, setRegisteredEmail] = useState(null)  
  const [email, setEmail] = useState("")  
  function onSubmit(event) {  
    fetch(`/api/vacation/${sku}/notify-when-in-season`, {  
      method: 'POST',  
      body: JSON.stringify({ email }),  
      headers: { 'Content-Type': 'application/json' },  
    })  
    .then(res => {  
      if(res.status < 200 || res.status > 299)
```



```

        return alert('We had a problem processing this...please try again.')
        setRegisteredEmail(email)
    })
    event.preventDefault()
}
if(registeredEmail) return (
  <i>You will be notified at {registeredEmail} when
  this vacation is back in season!</i>
)
return (
  <form onSubmit={onSubmit}>
    <i>Notify me when this vacation is in season:</i>
    <input
      type="email"
      placeholder="(your email)"
      value={email}
      onChange={({ target: { value } }) => setEmail(value)}
    />
    <button type="submit">OK</button>
  </form>
)
}

```

Optamos por fazer o componente rastrear dois valores diferentes: o endereço de email quando ele for digitado e o valor final após o usuário pressionar OK. No caso do primeiro valor, essa é uma técnica conhecida como *componentes controlados* e você pode ler mais sobre ela na *documentação de formulários do React* (<http://bit.ly/2X9P9qh>). Estamos rastreando o outro valor para saber quando o usuário executou a ação de pressionar OK e alterar a UI apropriadamente. Também poderíamos ter usado um booleano simples “registered” (registrado), mas nossa abordagem permite que a UI lembre ao usuário com que email ele foi registrado.

Também tivemos um pouco mais de trabalho na comunicação da API: tivemos de especificar o método (POST), codificar o corpo como JSON e estabelecer o tipo de conteúdo.

Observe que estamos tomando a decisão de qual UI será retornada. Se o usuário já tiver se registrado, retornaremos uma mensagem simples, caso

contrário, renderizaremos o formulário. Esse é um padrão muito comum no React.

Parece trabalho demais para essa pequena funcionalidade, que além disso é uma funcionalidade muito básica. Nossa manipulação de erros para o caso de haver algo inadequado com a chamada de API é funcional, mas pouco amigável para o usuário, e, embora o componente lembre os pacotes de férias nos quais nos inscrevemos, ele só fará isso enquanto estivermos nessa página. Se sairmos dela e voltarmos, veremos o formulário novamente.

Há medidas que poderíamos tomar para tornar esse código um pouco mais palatável. Para começar, poderíamos criar um encapsulador de API que manipulasse os confusos detalhes da codificação das entradas e da determinação de erros; isso certamente traria benefícios à medida que usássemos mais endpoints de API. Também há muitos frameworks populares de processamento de formulários para o React que ajudam a diminuir a carga desse processamento.

Resolver o problema da “lembrança” dos pacotes de férias nos quais o usuário se inscreveu é um pouco mais complicado. O que nos ajudaria seria uma maneira de nossos objetos de férias terem essa informação (se o usuário se registrou ou não) disponível. No entanto, o componente de uso especial não sabe nada sobre o pacote de férias; só conhece a SKU. Na próxima seção, falaremos sobre o *gerenciamento de estado*, que indica uma solução para esse problema.

Gerenciamento de estado

Grande parte do trabalho de arquitetura envolvido no planejamento e design de uma aplicação React se baseia no gerenciamento de estado – mas nem sempre no gerenciamento de estado de componentes individuais, e sim em como eles compartilham e coordenam o estado. Nosso exemplo de aplicação compartilha algum estado: o componente `Vacations` passa um objeto de férias para o componente `Vacation`, que por sua vez passa a SKU do pacote de férias para o ouvinte `NotifyWhenInSeason`. Porém, até agora, as informações estão apenas *descendo* a árvore; o que acontecerá quando elas precisarem voltar *para cima*?

A abordagem mais comum seria passarmos funções que fossem responsáveis pela atualização do estado. Por exemplo, o componente `Vacations` poderia ter uma função para a modificação de um pacote de férias, que ele passaria para `Vacation`, e que então poderia ser passada para baixo para `NotifyWhenInSeason`. Quando `NotifyWhenInSeason` a chamasse para modificar o pacote de férias, `Vacations`, que está no topo da árvore, reconheceria que as coisas mudaram, o que o faria ser renderizado novamente, e consequentemente todos os seus descendentes também seriam renderizados.

Parece exaustivo e complicado, e às vezes é, mas há técnicas que podem ajudar. Elas são tão variadas e ocasionalmente complexas que não podemos abordá-las totalmente aqui (e, além disso, este livro não é sobre o React), mas posso lhe indicar alguma leitura adicional:

Redux (<https://redux.js.org>)

Geralmente o Redux é a primeira coisa que vem à mente das pessoas quando elas pensam em um gerenciamento de estado abrangente para aplicações React. Foi uma das primeiras arquiteturas de gerenciamento de estado formalizadas, e ainda é muito popular. Seu conceito é extremamente simples e ele continua sendo meu framework de gerenciamento de estado preferido. Mesmo se você não optar por usar o Redux, recomendo que assista aos *vídeos de tutoriais gratuitos* (<https://egghead.io/courses/getting-started-with-redux>) de seu criador, Dan Abramov.

MobX (<https://mobx.js.org>)

O MobX surgiu depois do Redux. Ele atraiu vários seguidores em pouco tempo e provavelmente é o segundo contêiner de estado mais popular, atrás somente do Redux. O MobX pode com certeza resultar em código que parece mais fácil de escrever, mas ainda acho que o Redux apresenta a vantagem de ser um bom framework à medida que a aplicação aumenta de escala, mesmo com o crescimento de seu boilerplate.

Apollo (<https://www.apollographql.com>)

O Apollo não é *exatamente* uma biblioteca de gerenciamento de estado, mas a maneira como é usado com frequência o torna um bom substituto.

Ele é basicamente uma interface front-end para o *GraphQL* (<https://graphql.org>) – uma alternativa às APIs REST – que oferece uma ótima integração com o React. Se você estiver usando o GraphQL (ou achá-lo interessante), vale a pena testar o Apollo.

React Context (<https://reactjs.org/docs/context.html>)

O React foi introduzido ao fornecer a Context API, agora incorporada a ele. O React Context faz muitas das coisas que o Redux faz, porém com menos boilerplate. No entanto, acho-o menos robusto e considero o Redux uma opção melhor para as aplicações quando elas crescem.

Quando você começar a usar o React, provavelmente ignorará as complexidades de gerenciamento do estado em sua aplicação, mas muito rapidamente sentirá a necessidade de ter uma maneira mais organizada de gerenciá-lo. Quando chegar a esse ponto, vai querer examinar algumas dessas opções e selecionar a que melhor lhe atender.

Opções de implantação

Até agora, usamos o servidor de desenvolvimento interno do CRA – que na verdade é a melhor opção para o desenvolvimento e recomendo seu uso. No entanto, quando chega a hora da implantação, ele não é uma opção adequada. Felizmente, o CRA vem carregado com um script de build que cria um pacote otimizado para a produção e que oferece muitas opções. Quando estiver pronto para criar um pacote de implantação, simplesmente execute `yarn build`, e um diretório *build* será gerado. Todos os arquivos do diretório *build* são estáticos e podem ser implantados em qualquer local.

Atualmente meu método de implantação preferido é inserir o build do CRA em um bucket S3 da AWS com a *Hospedagem de Site Estático* (<https://amzn.to/3736fuT>) ativada. É claro que essa não é a única opção: todos os principais provedores de nuvem e CDNs oferecem algo semelhante.

Nessa configuração, temos de criar o roteamento para que as chamadas de API sejam enviadas para o servidor Express e o pacote estático seja servido a partir de uma CDN. Em minhas implantações da AWS, uso o *AWS CloudFront* (<https://amzn.to/2KglZRb>) para executar o roteamento; os

arquivos estáticos são servidos a partir do bucket S3 já mencionado, e as requisições de API são roteadas para um servidor Express em uma instância EC2, ou em um Lambda.

Outra opção é permitir que o Express faça tudo. Ela tem a vantagem de podermos consolidar a aplicação inteira em um único servidor, o que gera uma implantação muito mais simples e facilita o gerenciamento. Pode não ser ideal para a escalabilidade ou o desempenho, mas é uma opção válida para aplicações pequenas.

Para servir sua aplicação totalmente a partir do Express, simplesmente pegue o conteúdo do diretório *build* que foi criado quando você executou `yarn build` e copie-o no diretório *public* na aplicação Express. Se você tiver incluído o `middleware static`, ele servirá o arquivo *index.html* automaticamente, que é só o que precisamos.

Faça um teste: se seu servidor Express ainda estiver sendo executado na porta 3033, você deve poder visitar <http://localhost:3033> e ver a mesma aplicação que o servidor de desenvolvimento do CRA está fornecendo!



Caso queira saber como o servidor de desenvolvimento do CRA funciona, ele usa um pacote chamado `webpack-dev-server`, que utiliza o Express em segundo plano! Logo, no fim das contas, tudo se resume ao Express.

Conclusão

Este capítulo examinou o React, e as tecnologias que o cercam, apenas superficialmente. Se quiser conhecê-lo melhor, *Learning React* (<https://oreil.ly/ROqku>) de Alex Banks e Eve Porcello (O'Reilly) é um ótimo ponto de partida. Esse livro também aborda o gerenciamento de estado com o Redux (no entanto, atualmente ele não fala sobre hooks). A *documentação oficial do React* (<http://bit.ly/37377Qb>) também é abrangente e bem escrita.

Sem dúvida, os SPAs mudaram a maneira de pensarmos sobre e distribuímos aplicações web, e permitiram melhorias significativas no desempenho, principalmente no ambiente móvel. Ainda que o Express tenha sido criado em uma época em que grande parte do HTML ainda era

renderizada no servidor, isso não o tornou obsoleto. Pelo contrário, a necessidade de fornecermos APIs para aplicações de página única lhe deu nova vida!

Também deve ter ficado claro com a leitura deste capítulo que o jogo ainda é o mesmo: dados sendo enviados e retornados entre navegadores e servidores. Só a natureza dos dados mudou, e temos de nos acostumar a alterar o HTML por intermédio da manipulação dinâmica do DOM.

¹ Por razões de desempenho, o pacote pode ser dividido em “blocos” que são carregados quando necessário (o que é chamado de *carregamento preguiçoso*[lazy loading], mas o princípio é o mesmo.

CAPÍTULO 17

Conteúdo estático

Conteúdo estático são os recursos servidos pelo aplicativo que não mudam a cada requisição. Aqui estão os mais comuns:

Multimídia

Imagens, vídeos e arquivos de áudio. É claro que é possível gerar arquivos de imagens (assim como de vídeo e de áudio, embora esses casos sejam menos comuns) dinamicamente, mas a maioria dos recursos multimídia é estática.

HTML

Se nossa aplicação web estiver usando views para renderizar HTML dinâmico, isso não seria categorizado como HTML estático (mas, por razões de desempenho, podemos gerar HTML dinamicamente, armazená-lo em cache e servi-lo como um recurso estático). Como vimos, normalmente as aplicações SPA enviam um único arquivo HTML estático para o cliente, e essa é a razão mais comum para tratarmos HTML como um recurso estático. É bom ressaltar que demandar que o cliente use uma extensão *.html* não é uma abordagem muito moderna, logo, atualmente a maioria dos servidores permite que recursos HTML estáticos sejam servidos sem essa extensão (portanto, */foo* e */foo.html* retornariam o mesmo conteúdo).

CSS

Mesmo se você usar um pré-processador dinâmico da linguagem CSS como o LESS, Sass ou Stylus, seu navegador precisará do CSS puro, que é um recurso estático.¹

JavaScript

Só porque o servidor está executando JavaScript isso não significa que não haverá JavaScript no lado do cliente. O JavaScript do lado do cliente é considerado um recurso estático. É claro que atualmente a fronteira está ficando um pouco mais nebulosa: e se houvesse um código que quiséssemos usar no back-end e no lado do cliente? Há maneiras de resolver esse problema, mas geralmente o JavaScript que é enviado para o cliente é estático.

Downloads binários

Essa categoria é genérica: PDFs, arquivos ZIP, documentos do Word, instaladores e outros recursos semelhantes.



Se você só estiver construindo uma API, talvez não haja recursos estáticos. Se for esse o caso, pode pular este capítulo.

Considerações de desempenho

A maneira como manipulamos os recursos estáticos afeta significativamente o desempenho do site no mundo real, principalmente se ele fizer uso intenso de multimídia. As duas principais considerações de desempenho são *reduzir o número de requisições* e *diminuir o tamanho do conteúdo*.

Das duas, reduzir o número de requisições (HTTP) é a mais crítica, especialmente no ambiente móvel (o overhead de enviar uma requisição HTTP é substancialmente maior por uma rede de celular). A redução do número de requisições pode ser obtida de duas maneiras: pela combinação de recursos e pelo cache do navegador.

A combinação de recursos é uma consideração relacionada principalmente à arquitetura e ao front-end: sempre que possível, imagens pequenas devem ser combinadas em um único sprite. Use então o CSS para definir o deslocamento e o tamanho e exibir somente a parte da imagem que quiser. Para a criação de sprites, recomendo o serviço gratuito *SpritePad* (<http://bit.ly/33GYvwm>). Ele torna muito fácil gerar sprites e também gera o CSS. Não poderia ser mais fácil. Provavelmente você só vai precisar da funcionalidade gratuita do SpritePad, mas, se tiver de criar muitos sprites,

talvez valha a pena usar seus recursos premium.

O cache do navegador ajuda a reduzir as requisições HTTP armazenando os recursos estáticos mais usados no navegador do cliente. Embora os navegadores se esforcem para tornar o cache o mais automático possível, não há mágica: há muitas coisas que você pode e deve fazer para permitir o armazenamento de seus recursos estáticos no cache do navegador.

Para concluir, podemos melhorar o desempenho diminuindo o tamanho dos recursos estáticos. Algumas técnicas são *lossless* (a diminuição do tamanho pode ser obtida sem a perda de dados) e outras são *lossy* (a diminuição é obtida pela redução da qualidade dos recursos). As técnicas *lossless* incluem a minificação de JavaScript e CSS e a otimização de imagens PNG. Como técnicas *lossy* temos o aumento dos níveis de compactação de arquivos JPEG e de vídeo. Discutiremos a minificação e o empacotamento (que também reduz as requisições HTTP) neste capítulo.



A importância da redução das requisições HTTP diminuirá com o tempo quando o HTTP/2 for mais usado. Uma das principais melhorias do HTTP/2 é a *multiplexação de requisição e resposta*, que reduz o overhead da busca de vários recursos em paralelo. Consulte o texto “*Introdução a HTTP/2*” (<http://bit.ly/34TXhxR>) de Ilya Grigorik for para obter mais informações.

Redes de distribuição de conteúdo

Quando você passar seu site para a produção, os recursos estáticos devem ser hospedados *em algum lugar* na internet. Você pode estar acostumado a hospedá-los no mesmo servidor em que todo o HTML dinâmico é gerado. Até agora, nosso exemplo também usou essa abordagem: o servidor Node/Express que criamos quando digitamos `node meadowlark.js` serve todo o HTML assim como os recursos estáticos. No entanto, se quiser maximizar o desempenho do site (ou quiser fazer isso no futuro), deve tornar fácil hospedar os recursos estáticos em uma *rede de distribuição de conteúdo* (CDN, content delivery network). Uma CDN é um servidor que é otimizado para distribuir recursos estáticos. Ela usa cabeçalhos especiais (que

conheceremos em breve) que permitem o armazenamento no cache do navegador.

As CDNs também facilitam a *otimização geográfica* (com frequência chamada de *cache de borda* [edge caching]; isto é, elas podem distribuir o conteúdo estático a partir de um servidor que esteja geograficamente mais próximo do cliente. Embora a internet seja muito rápida (não opera na velocidade da luz, mas chega perto), continua sendo mais rápido distribuir dados por centenas em vez de milhares de quilômetros. As economias de tempo individuais podem ser pequenas, mas, se você multiplicar por todos os seus usuários, requisições e recursos, elas aumentam bastante.

A maioria de seus recursos estáticos será referenciada em views HTML (elementos `<link>` de arquivos CSS, referências `<script>` a arquivos JavaScript, tags `` referenciando imagens, e tags de inclusão de multimídia). Também é comum que haja referências estáticas em CSS, geralmente na propriedade `background-image`. Por fim, às vezes os recursos estáticos são referenciados em JavaScript, como em um código JavaScript que alterasse ou inserisse dinamicamente tags `` ou a propriedade `background-image`.



Geralmente não precisamos nos preocupar com o compartilhamento de recursos entre domínios (CORS) quando usamos uma CDN. Recursos externos carregados em HTML não estão sujeitos à política do CORS: você só precisa habilitar o CORS para recursos que sejam carregados por intermédio do Ajax (consulte o Capítulo 15).

Design para CDNs

A arquitetura de seu site influenciará como você usará uma CDN. A maioria das CDNs permite configurar regras de roteamento para a determinação de para onde serão enviadas as requisições recebidas. Embora essas regras de roteamento possam criar uma sofisticação arbitrária, tudo se resume a enviar requisições de arquivos estáticos para um local (geralmente fornecido pela CDN) e requisições de endpoints dinâmicos (como páginas dinâmicas ou endpoints de API) para outro.

A seleção e a configuração de uma CDN compõem um tópico extenso que

não detalharei aqui, mas mostrarei o que aprendi com minha experiência para ajudá-lo a configurar a CDN que escolher.

A abordagem mais cômoda para a estruturação de sua aplicação seria facilitar a diferenciação entre arquivos dinâmicos e estáticos para simplificar ao máximo as regras de roteamento da CDN. Embora seja possível fazer isso usando subdomínios (os arquivos dinâmicos seriam servidos a partir de meadowlark.com e os estáticos a partir de static.meadowlark.com, por exemplo), essa abordagem tem complicações adicionais e dificulta o desenvolvimento local. Um método melhor seria com o uso dos paths de requisição: tudo que começasse com `/public/` seria um arquivo estático e o resto seria dinâmico, por exemplo. A abordagem pode ser diferente se você estiver gerando seu conteúdo com o Express ou usando-o para fornecer uma API para uma aplicação de página única.

Site renderizado no servidor

Se você estiver usando o Express para renderizar o HTML dinâmico, seria mais fácil dizermos “Tudo que comece com `/static/` será um arquivo estático e o resto será dinâmico”. Com essa abordagem, todas as suas URLs (geradas dinamicamente) serão o que você quiser que elas sejam (contanto, claro, que não comecem com `/static/!`) e todos os arquivos estáticos serão prefixados com `/static/`:

```

Welcome to <a href="/about">Meadowlark Travel</a>.
```

Até agora neste livro, usamos o middleware `static` do Express como se ele estivesse hospedando todos os arquivos estáticos na raiz. Isto é, se inserirmos um arquivo estático `foo.png` no diretório `public`, vamos referenciá-lo com o path de URL `/foo.png`, e não com `/static/foo.png`. É claro que poderíamos criar um subdiretório `static` dentro do diretório `public` existente, e então `/public/static/foo.png` teria a URL `/static/foo.png`, mas isso parece um pouco despropositado. Felizmente, o middleware `static` nos poupa desse despropósito. Só precisamos especificar um path diferente quando chamarmos `app.use`:

```
app.use('/static', express.static('public'))
```

Agora podemos usar a mesma estrutura de URL tanto no ambiente de desenvolvimento quanto na produção. Se tomarmos o cuidado de manter o diretório *public* sincronizado com o que há na CDN, poderemos referenciar os mesmos arquivos estáticos nos dois locais e nos mover naturalmente entre o desenvolvimento e a produção.

Quando configurarmos o roteamento de nossa CDN (você terá de consultar a documentação de sua CDN para fazer isso), ele funcionará assim:

Path da URL	Destino/origem do roteamento
/static/*	Armazenamento CDN de arquivos estáticos
/* (todo o resto)	Seu servidor Node/Express, proxy ou balanceador de carga

Aplicações de página única

Normalmente as aplicações de página única são o oposto de um site renderizado no servidor: só a API é roteada para o servidor (por exemplo, qualquer requisição prefixada com */api*) e todo o resto é roteado novamente para o armazenamento de arquivos estáticos.

Como vimos no Capítulo 16, podemos criar um pacote de produção para a aplicação, que incluirá todos os recursos estáticos, e será enviado por upload para a CDN. Depois, basta assegurar que o roteamento seja configurado corretamente para a API. O roteamento, então, ficará assim:

Path da URL	Destino/origem do roteamento
/api/*	Seu servidor Node/Express, proxy ou balanceador de carga
/* (todo o resto)	Armazenamento CDN de arquivos estáticos

Agora que vimos como podemos estruturar uma aplicação para passarmos naturalmente do desenvolvimento para a produção, voltaremos nossa atenção para o que ocorre com o cache e como ele pode melhorar o desempenho.

Armazenando arquivos estáticos em cache

Esteja você usando o Express para servir arquivos estáticos ou uma CDN, é útil conhecer os cabeçalhos de resposta HTTP que o navegador usa para determinar quando e como armazenar arquivos estáticos em cache:

Expires/Cache-Control

Esses dois cabeçalhos informam ao navegador o período de tempo máximo que um recurso pode ser armazenado em cache. Eles são seguidos fielmente pelo navegador: se solicitarem que o navegador armazene algo em cache por um mês, ele não o baixará novamente por um mês, contanto que permaneça no cache. É importante saber que o navegador pode remover a imagem do cache prematuramente, por razões que não controlamos. Por exemplo, o usuário poderia limpar o cache manualmente, ou o navegador poderia remover um recurso para fazer espaço para outros recursos que o usuário estivesse visitando com mais frequência. Você só precisará de um desses cabeçalhos, e Expires tem suporte mais amplo, logo, é preferível que ele seja usado. Se o recurso estiver no cache, e ainda não tiver expirado, o navegador não emitirá uma requisição GET, o que melhora o desempenho, principalmente no ambiente móvel.

Last-Modified/ETag

Essas duas tags fornecem um versionamento básico: se o navegador precisar buscar o recurso, ele as examinará *antes* de baixar o conteúdo. Uma requisição GET ainda será emitida para o servidor, mas, se os valores retornados por esses cabeçalhos garantirem ao navegador que o recurso não mudou, ele não fará o download do arquivo. Como o nome indica, Last-Modified permite especificar a data em que o recurso foi modificado pela última vez. ETag permite usar uma string arbitrária, que geralmente é uma string de versão ou um hash de conteúdo (content hash).

Ao servir recursos estáticos, você deve usar o cabeçalho Expires e Last-Modified ou ETag. O middleware interno static do Express usa Cache-Control, mas não manipula Last-Modified ou ETag. Logo, embora ele seja adequado para o desenvolvimento, não é uma boa solução para a implantação.

Se você optar por hospedar seus recursos estáticos em uma CDN, como a Amazon CloudFront, Microsoft Azure, Fastly, Cloudflare, Akamai ou StackPath, a vantagem é que elas manipularão a maioria desses detalhes automaticamente. Você poderá ajustar os detalhes, mas geralmente os padrões fornecidos por qualquer um desses serviços já são apropriados.

Alterando seu conteúdo estático

O cache melhora significativamente o desempenho do site, mas traz consequências. Especificamente, se você alterar alguns de seus recursos estáticos, os clientes podem não os ver até as versões armazenadas em cache expirarem no navegador. O Google recomenda o armazenamento em cache por um mês, preferivelmente por um ano. Imagine um usuário que usasse seu site todo dia no mesmo navegador: essa pessoa pode não ver suas atualizações por um ano inteiro!

É claro que essa é uma situação indesejável, e você não pode simplesmente solicitar aos usuários que limpem o cache. A solução é o cache busting (bloqueio de cache). *Cache busting* é uma técnica que nos permite controlar quando o navegador do usuário será forçado a baixar novamente um arquivo. Geralmente isso é feito com o versionamento do arquivo (*main.2.css* ou *main.css?version=2*) ou a inclusão de algum tipo de hash (*main.e16b7e149dccfcc399e025e0c454bf77.css*). Independentemente da técnica utilizada, quando você atualizar o arquivo, o nome do recurso mudará, e o navegador saberá que precisa baixá-lo.

Podemos fazer a mesma coisa com nossos arquivos multimídia. Vejamos o logotipo, por exemplo (*/static/img/meadowlark_logo.png*). Se o hospedarmos em uma CDN para maximizar o desempenho, especificando a expiração em um ano, e depois o alterarmos, os usuários podem não ver o logotipo atualizado durante o período de um ano. No entanto, se renomearmos o logotipo para */static/img/meadowlark_logo-1.png* (e refletirmos essa alteração de nome no HTML), o navegador será forçado a baixá-lo, porque agora ele parece um novo recurso.

Se você estiver usando um framework de aplicação de página única, como o create-react-app ou semelhante, ele fornecerá uma etapa de build que criará pacotes de recursos prontos para a produção com hashes acrescentados.

Se estiver começando do zero, provavelmente vai querer usar um *empacotador* (que é o que os frameworks SPA usam em segundo plano). Os empacotadores (bundlers) combinam o JavaScript, o CSS e alguns outros tipos de arquivos estáticos no menor conjunto possível, e minificam o resultado (dando-lhe o menor tamanho possível). A configuração de um

empacotador é um tópico extenso, mas felizmente há documentações muito bem feitas disponíveis. Os empacotadores mais populares existentes atualmente são os seguintes:

Webpack (<https://webpack.js.org>)

O Webpack foi um dos primeiros empacotadores que se tornou realmente popular, e ainda mantém muitos seguidores. É bastante sofisticado, mas a sofisticação tem um preço: a curva de aprendizagem é íngreme. No entanto, é útil conhecer pelo menos os aspectos básicos.

Parcel (<https://parceljs.org>)

O Parcel é novo, mas fez um grande estardalhaço. É extremamente bem documentado, é muito rápido e, o melhor de tudo, tem a curva de aprendizagem mais curta. Se você deseja concluir o trabalho rapidamente, sem muita confusão, comece por aqui.

Rollup (<https://rollupjs.org>)

O Rollup fica em algum lugar entre o Webpack e o Parcel. Como o Webpack, é muito robusto e tem vários recursos. No entanto, é mais fácil de começar a usar do que o Webpack, e não é tão simples como o Parcel.

Conclusão

Para algo que parece simples, os recursos estáticos podem causar muitos problemas. Porém, provavelmente representam grande parte dos dados que são transferidos para os visitantes, logo, vale a pena gastar algum tempo otimizando-os.

Uma solução viável para os arquivos estáticos que ainda não mencionamos seria simplesmente os hospedar em uma CDN desde o início, e sempre usar a URL completa do recurso nas views e no CSS. Isso tem a vantagem de ser simples, mas, se você quiser passar o fim de semana em um hackathon² em uma cabana na floresta sem acesso à internet, terá problemas!

O uso do empacotamento e da minificação é outra área na qual você pode economizar tempo se não valer a pena para sua aplicação. Especificamente,

se seu site incluir apenas um ou dois arquivos JavaScript, e todo o CSS residir em um único arquivo, provavelmente você poderá deixar o empacotamento de lado, mas as aplicações do mundo real tendem a crescer com o tempo.

Seja qual for a técnica que você usar para servir seus recursos estáticos, recomendo hospedá-los separadamente, de preferência em uma CDN. Se achar complicado, asseguro que não é tão difícil quanto parece, principalmente se você dedicar algum tempo ao seu sistema de implantação, caso em que a implantação de recursos estáticos em um local e de sua aplicação em outro será automática.

Se você está preocupado com os custos das CDNs, sugiro que examine o que está pagando atualmente com a hospedagem. A maioria dos provedores de hospedagem cobra basicamente pela largura de banda, mesmo quando não a conhecemos. No entanto, se repentinamente seu site for mencionado no Slashdot, e as visitas aumentarem, você pode receber uma conta de hospedagem que não esperava. Geralmente a hospedagem da CDN é definida de modo a pagarmos pelo que usamos. Como exemplo, um site que gerencie para uma empresa regional de tamanho médio, que usava cerca de 20 GB de largura de banda por mês, pagava apenas alguns dólares mensais para hospedar recursos estáticos (e era um site que usava intensamente mídia).

Os ganhos no desempenho que você perceberá ao hospedar seus recursos estáticos em uma CDN serão significativos, e o custo e a inconveniência de fazer isso são mínimos, logo, recomendo que siga esse caminho.

1 É possível usar LESS não compilado em um navegador com alguns passes de mágica do JavaScript. Há consequências para o desempenho nessa abordagem, logo, não a recomendo.

2 Hackathon, termo eventualmente aportuguesado para “hackaton”, é uma maratona de programação na qual hackers se reúnem por horas, dias ou até semanas, a fim de explorar dados abertos, desvendar códigos e sistemas lógicos, discutir novas ideias e desenvolver projetos de software ou mesmo de hardware.

CAPÍTULO 18

Segurança

Atualmente a maioria dos sites e aplicações tem algum tipo de requisito de segurança. Se você está permitindo que as pessoas façam login, ou se está armazenando informações de identificação pessoal (PII, personally identifiable information), é melhor implementar segurança em seu site. Neste capítulo, discutiremos o *HTTP Secure* (HTTPS), que estabelece uma base sobre a qual é possível construir um site seguro, e os mecanismos de autenticação, com ênfase na autenticação de terceiros.

A segurança é um tópico extenso que poderia ter um livro próprio. Logo, nos concentraremos no uso dos módulos de autenticação existentes. É claro que você poderia criar o próprio sistema de autenticação, mas essa é uma tarefa vultosa e complicada. Além disso, há boas razões para preferirmos uma abordagem de login de terceiros, o que discutiremos posteriormente neste capítulo.

HTTPS

A primeira etapa para o fornecimento de serviços seguros é usar o HTTPS. A natureza da internet possibilita que terceiros interceptem os pacotes que são transmitidos entre clientes e servidores. O HTTPS criptografa esses pacotes, tornando extremamente difícil para um invasor obter acesso às informações que estão sendo transmitidas (digo “muito difícil”, e não “impossível”, porque não existe segurança perfeita. No entanto, o HTTPS é considerado suficientemente seguro para serviços bancários, proteção empresarial e dados de assistência médica).

Você pode considerar o HTTPS como uma base para a proteção de seu site. Ele não fornece autenticação, mas disponibiliza a fundação para a sua

criação. Por exemplo, provavelmente seu sistema de autenticação envolve a transmissão de uma senha; se essa senha for transmitida sem criptografia, seja qual for a sofisticação da autenticação, ela não protegerá o sistema. O nível da segurança é determinado pela resistência do seu elo mais fraco, e o primeiro elo dessa corrente é o protocolo de rede.

O protocolo HTTPS requer que o servidor tenha um *certificado de chave pública*, às vezes chamado de certificado SSL. O formato padrão atual dos certificados SSL chama-se X.509. A ideia existente por trás dos certificados é a seguinte: é preciso que existam *autoridades de certificação* (CAs, certificate authorities) para emití-los. A autoridade de certificação torna *certificados raiz confiáveis* disponíveis para os fornecedores de navegadores. Os navegadores incluem esses certificados raiz confiáveis quando os instalamos, e é isso que estabelece a cadeia de confiança entre a CA e o navegador. Para essa cadeia funcionar, o servidor deve usar um certificado emitido por uma CA.

A conclusão é que, para fornecer o HTTPS, você precisa do certificado de uma CA, mas como obtê-lo? Podemos gerar nosso próprio certificado, obter um certificado em uma CA gratuita ou comprá-lo de uma CA comercial.

Gerando o próprio certificado

É fácil gerar nosso próprio certificado, mas geralmente isso só é adequado para fins de desenvolvimento e teste (e possivelmente para implantação na intranet). Devido à natureza hierárquica estabelecida pelas autoridades de certificação, os navegadores só confiam em certificados gerados por uma CA conhecida (que provavelmente não é você). Se seu site usar um certificado de uma CA que o navegador não conheça, ele o avisará em um linguajar muito alarmante que você está estabelecendo uma conexão segura com uma entidade desconhecida (e, portanto, não confiável). No desenvolvimento e em testes, não há problemas: você e sua equipe sabem que geraram o próprio certificado, e esse comportamento é esperado dos navegadores. Porém, se você implantasse o site no ambiente de produção para uso público, os visitantes fugiriam sem hesitar.



Se você controlar a distribuição e a instalação de navegadores, poderá

instalar automaticamente seu próprio certificado raiz ao instalar o navegador. Esse método impedirá que as pessoas que estiverem usando esse navegador sejam avisadas quando se conectarem com seu site. No entanto, isso não é fácil de definir e só é aplicável a ambientes em que haja um controle sobre o(s) navegador(es) usado(s). A menos que você tenha uma razão muito sólida para adotar essa abordagem, geralmente ela traz mais problemas do que benefícios.

Para gerar o próprio certificado, você precisará de uma implementação do OpenSSL. A Tabela 18.1 mostra como adquirir uma implementação.

Tabela 18.1 – Adquirindo uma implementação para diferentes plataformas.

Plataforma	Instruções
macOS	<code>brew install openssl</code>
Ubuntu, Debian	<code>sudo apt-get install openssl</code>
Outras distribuições Linux	Baixe de http://www.openssl.org/source/ ; extraia o tarball e siga as instruções
Windows	Baixe de http://gnuwin32.sourceforge.net/packages/openssl.htm



Se você é usuário do Windows, pode ter de especificar a localização do arquivo de configuração do OpenSSL, o que talvez seja complicado devido aos pathnames do Windows. A maneira segura é localizar o arquivo *openssl.cnf* (geralmente no diretório *share* da instalação), e, antes de executar o comando `openssl`, definir a variável de ambiente `OPENSSL_CONF: SET OPENSSL_CONF=openssl.cnf`.

Uma vez que você instalar o OpenSSL, poderá gerar uma chave privada e um certificado público:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout meadowlark.pem  
-out meadowlark.crt
```

Serão solicitados alguns detalhes, como o código de seu país, a cidade e o estado, o nome de domínio totalmente qualificado (*FQDN* [fully qualified domain name] também chamado de *nome comum* [common name] ou *nome de host totalmente qualificado*) e o endereço de email. Já que esse certificado é para fins de desenvolvimento/teste, os valores fornecidos não são particularmente importantes (na verdade, são todos opcionais, mas não os

informar resultará em um certificado que será considerado ainda mais suspeito pelo navegador). O nome comum (FQDN) é o que o navegador usa para identificar o domínio. Logo, se você estiver usando *localhost*, poderá empregá-lo como seu FQDN, ou pode usar o endereço IP do servidor, ou até mesmo o nome do servidor, se disponível. A criptografia funcionará mesmo se o nome comum e o domínio utilizado na URL não coincidirem, mas o navegador emitirá um aviso adicional sobre a discrepância.

Se você estiver curioso com relação aos detalhes desse comando, pode ler sobre eles na *página de documentação do OpenSSL* (<http://bit.ly/2q64psm>). É bom ressaltar que a opção *-nodes* não tem nenhuma ligação com o Node, e nem mesmo com a palavra no plural “nodes”: na verdade significa “no DES,” indicando que a chave privada não está DES-criptografada.

O resultado desse comando são dois arquivos, *meadowlark.pem* e *meadowlark.crt*. O arquivo Privacy-Enhanced Electronic Mail (PEM) é a chave privada e não deve ser disponibilizado para o cliente. O arquivo CRT é o certificado autoassinado que será enviado para o navegador para o estabelecimento de uma conexão segura.

Alternativamente, há sites que fornecem certificados autoassinados gratuitos, como *esse* (<http://bit.ly/354CIEl>).

Usando uma autoridade de certificação gratuita

O HTTPS se baseia na confiança, e infelizmente uma das maneiras mais fáceis de ganhar confiança na internet é comprando-a. E o resto também não é fácil: é caro estabelecer a infraestrutura de segurança, comprar um seguro para os certificados e manter relacionamentos com os fornecedores de navegador.

Comprar um certificado não é a única opção legítima para a aquisição de certificados prontos para a produção: a *Let's Encrypt* (<https://letsencrypt.org>), uma CA gratuita e automatizada baseada na abordagem open source, tornou-se uma ótima opção. Na verdade, a menos que você já tenha investido em uma infraestrutura que ofereça certificados gratuitos ou baratos como parte de sua solução de hospedagem (a AWS, por exemplo), a Let's Encrypt é uma excelente opção. A única desvantagem é que o prazo de validade máximo dos

certificados é de 90 dias. Essa desvantagem é compensada pelo fato de que a Let's Encrypt facilita a renovação automática dos certificados e recomenda a definição de um processo automatizado para que isso seja feito a cada 60 dias de modo a assegurarmos que os certificados não expirem.

Todos os principais fornecedores (como a Comodo e a Symantec) oferecem certificados de teste gratuitos que duram de 30 a 90 dias. Essa é uma opção válida se você quiser testar um certificado comercial, mas é preciso comprar o certificado antes de o período de teste começar para assegurar a continuidade do serviço.

Comprando um certificado

Atualmente, 90% dos aproximadamente 50 certificados raiz distribuídos com cada grande navegador são de propriedade de quatro empresas: Symantec (que comprou a VeriSign), Comodo Group, Go Daddy e GlobalSign. Comprar diretamente de uma CA pode ser muito caro: em geral começa com cerca de 300 dólares por ano (embora algumas ofereçam certificados por menos de 100 dólares por ano). Uma opção menos cara é contatar um revendedor, de quem você poderá obter um certificado SSL por meros 10 dólares por ano ou menos.

É importante saber pelo que exatamente você está pagando, e por que pagaria 10, 150 ou 300 dólares (ou mais) por um certificado. O primeiro ponto que é preciso conhecer é se não há diferença no nível de criptografia oferecido entre um certificado de 10 dólares e um de 1.500 dólares. Isso é algo que as autoridades de certificação caras preferem que não saibamos: sua equipe de marketing fará de tudo para ocultar a resposta.

Se você optar por entrar em contato com um fornecedor de certificado comercial, recomendo as três considerações a seguir como fatores para essa tomada de decisão:

Suporte ao cliente

Se você tiver problemas com seu certificado, sejam eles relacionados com o suporte ao navegador (os clientes o avisarão se o certificado for marcado pelo seu navegador como não confiável), questões de instalação ou

dificuldades na renovação, haverá um bom suporte ao cliente. Essa é uma das razões para a compra de um certificado mais caro. Com frequência, o provedor de hospedagem revende certificados, e, pelo que vi em minha experiência, fornecem um nível mais alto de suporte, porque também querem nos manter como clientes da hospedagem.

Certificados de domínio único, multissubdomínio, curinga e multidomínio

Geralmente os certificados mais baratos são os de *domínio único*. Isso pode não parecer tão ruim, mas lembre-se de que, se você comprar um certificado para *meadowlarktravel.com*, ele não funcionará para *www.meadowlarktravel.com*, ou vice-versa. Logo, tendo a evitar certificados de domínio único, embora sejam uma boa opção para quem estiver preocupado com o orçamento (sempre podemos definir redirecionamentos para enviar as requisições para o domínio apropriado). Os *certificados multidomínio* são bons porque, por exemplo, poderíamos comprar um único certificado que abrangesse *meadowlarktravel.com*, *www.meadowlark.com*, *blog.meadowlarktravel.com*, *shop.meadowlarktravel.com* etc. A desvantagem é que é preciso saber antecipadamente que subdomínios queremos usar.

Se você perceber que adicionou ou usou diferentes subdomínios no decorrer de um ano (que precisem suportar o HTTPS), pode ser melhor utilizar um certificado *curinga*, que geralmente é mais caro. No entanto, ele funcionará para *qualquer* subdomínio, e você não precisará especificar quais são os subdomínios.

Para concluir, há os *certificados multidomínio*, que, como os certificados curinga, tendem a ser mais caros. Esses certificados suportam vários domínios, logo, você poderia ter, por exemplo, *meadowlarktravel.com*, *meadowlarktravel.us*, *meadowlarktravel.com* e as variantes *www*.

Certificados de domínio, de organização e de validação estendida

Há três tipos de certificados: de domínio, de organização e de validação estendida. Os *certificados de domínio*, como o nome sugere, apenas nos asseguram de que estamos trabalhando com o *domínio* que achamos ser o correto. Os *certificados de organização*, por outro lado, dão alguma

segurança sobre a organização com a qual estamos lidando. Eles são mais difíceis de obter: geralmente há burocracia e precisamos fornecer informações como o registro estadual e/ou federal do nome da empresa, endereços físicos etc. Diferentes fornecedores de certificados exigem diferentes tipos de documentação, portanto, certifique-se de perguntar ao fornecedor o que é necessário para obter um desses certificados. Por fim, temos os *certificados de validação estendida*, que são o Rolls Royce dos certificados SSL. São como os certificados de organização por verificarem a existência da empresa, mas requerem um padrão mais alto de prova e podem demandar até mesmo auditorias caras para estabelecer práticas de segurança de dados (embora isso seja cada vez mais raro). Podem ser obtidos por meros 150 dólares para um único domínio.

Recomendo os menos caros certificados de domínio ou os certificados de validação estendida. Embora verifiquem a existência da empresa, os certificados de organização não são exibidos diferentemente nos navegadores, logo, pela minha experiência, a menos que o usuário examine o certificado (o que é raro), não haverá nenhuma diferença aparente entre ele e o certificado de domínio. Já os certificados de validação estendida em geral exibem algumas pistas para os usuários de que eles estão lidando com uma empresa legítima (como a barra de URL ser exibida em verde e o nome da empresa aparecer perto do ícone do SSL).

Se você já lidou com certificados SSL, deve estar se perguntando por que não mencionei o seguro do certificado. Omiti esse fator de diferença de preço porque é um seguro contra algo quase impossível de provar. A ideia é a de que, se alguém sofrer perda financeira devido a transações no site, e puder *provar que isso ocorreu devido a uma criptografia inadequada*, o seguro cobrirá o prejuízo. Embora seja possível que, se a aplicação envolver transações financeiras, alguém tente entrar com uma ação legal contra você por perda financeira, a probabilidade de que isso tenha ocorrido devido a uma criptografia inadequada é basicamente zero. Se eu quisesse procurar danos causados por uma empresa que geraram perda financeira relacionada a serviços online, a última abordagem que adotaria seria tentar provar que a criptografia SSL não funcionou. Se você deparar com dois certificados que só

difiram no preço e na cobertura do seguro, compre o mais barato.

O processo de compra de um certificado começa com a criação de uma chave privada (como fizemos anteriormente para o certificado autoassinado). Em seguida, você gerará uma *solicitação de assinatura de certificado* (CSR, certificate signing request) cujo upload será feito durante o processo de compra (o emissor do certificado fornecerá instruções de como fazer isso). O emissor do certificado não terá acesso à chave privada e ela não será transmitida pela internet, o que a protegerá. O emissor lhe enviará então o certificado, que terá a extensão *.crt*, *.cer* ou *.der* (ele estará em um formato chamado Distinguished Encoding Rules ou DER, daí a extensão menos comum *.der*). Você também receberá qualquer certificado que esteja envolvido na cadeia de certificados. É seguro enviar esse certificado por email porque ele não funcionará sem a chave privada que você gerou.

Ativando o HTTPS para seu aplicativo Express

Você pode modificar seu aplicativo Express para servir o site pelo HTTPS. Na prática e na produção, isso é muito raro, e veremos o porquê na próxima seção. No entanto, para aplicações avançadas, para teste e para você conhecer melhor o HTTPS, é útil saber como servi-lo.

Uma vez que você tiver sua chave privada e o certificado, será fácil usá-los em seu aplicativo. Revisitaremos como criamos nosso servidor:

```
app.listen(app.get('port'), () => {  
  console.log(`Express started in ${app.get('env')} mode ` +  
    `on port + ${app.get('port')}.`)  
})
```

É fácil passar para o HTTPS. Recomendo que você insira sua chave privada e o certificado SSL em um subdiretório chamado *ssl* (embora seja comum mantê-los na raiz do projeto). Em seguida, simplesmente use o módulo *https* em vez de *http* e passe um objeto *options* para o método *createServer*:

```
const https = require('https')  
const fs = require('fs') // geralmente no início do arquivo  
  
// ...o resto da configuração de sua aplicação
```



```
const options = {
  key: fs.readFileSync(__dirname + '/ssl/meadowlark.pem'),
  cert: fs.readFileSync(__dirname + '/ssl/meadowlark.crt'),
}

const port = process.env.PORT || 3000
https.createServer(options, app).listen(port, () => {
  console.log(`Express started in ${app.get('env')} mode ` +
    `on port + ${port}.`)
})
```

Isso é tudo. Supondo que você ainda esteja executando seu servidor na porta 3000, agora poderá se conectar com `https://localhost:3000`. Se tentar se conectar com `http://localhost:3000`, o tempo do acesso simplesmente expirará.

Observação sobre as portas

Sabendo ou não disso, quando você visitar um site, estará *sempre* se conectando com uma porta específica, ainda que ela não apareça na URL. Se você não especificar uma porta, a porta 80 será usada como padrão para o HTTP. A maioria dos navegadores não exibe o número da porta quando especificamos explicitamente a porta 80. Por exemplo, navegue para `http://www.apple.com:80`; provavelmente, quando a página for carregada, o navegador removerá a parte `:80`. Ele estará se conectando com a porta 80, mas isso ficará implícito.

Da mesma forma, há uma porta padrão para o HTTPS, a 443. O comportamento do navegador é semelhante: se você se conectar com `https://www.google.com:443`, a maioria dos navegadores não exibirá a parte `:443`, mas essa é a porta com a qual estarão se conectando.

Se você não estiver usando a porta 80 para o HTTP ou a porta 443 para o HTTPS, terá de especificar explicitamente a porta e o protocolo para se conectar corretamente. Não há uma maneira de executar o HTTP e o HTTPS na mesma porta (tecnicamente, é possível, mas não há uma boa razão para fazê-lo, e a implementação seria muito complicada).

Se você quiser executar seu aplicativo HTTP na porta 80 ou o aplicativo

HTTPS na porta 443 para não ter de especificar a porta explicitamente, precisa considerar duas coisas. A primeira é que muitos sistemas já têm um servidor web padrão sendo executado na porta 80.

A outra coisa que é preciso saber é que na maioria dos sistemas operacionais, as portas 1–1023 requerem privilégios elevados para serem abertas. Por exemplo, em uma máquina Linux ou macOS, se você tentar iniciar seu aplicativo na porta 80, provavelmente ele falhará com um erro EACCES. Para executá-lo na porta 80 ou 443 (ou qualquer porta abaixo de 1024), será necessário elevar seus privilégios usando o comando `sudo`. Se você não tiver direitos de administrador, não poderá iniciar o servidor diretamente na porta 80 ou 443.

A menos que esteja gerenciando os próprios servidores, é provável que não tenha acesso root à conta hospedada: o que acontecerá então quando quiser realizar a execução na porta 80 ou 443? Geralmente, os provedores de hospedagem têm algum tipo de serviço proxy executado com privilégios elevados que passa requisições para o aplicativo, que é executado em uma porta não privilegiada. Aprenderemos mais sobre isso na próxima seção.

HTTPS e Proxies

Como vimos, é muito fácil usar o HTTPS com o Express, e, para o desenvolvimento, ele funcionará bem. No entanto, se você quiser aumentar a escala de seu site para manipular mais tráfego, é melhor usar um servidor proxy como o NGINX (consulte o Capítulo 12). Se o site estiver sendo executado em um ambiente de hospedagem compartilhado, é quase certo que haja um servidor proxy que roteará as requisições para a aplicação.

Se você estiver usando um servidor proxy, o cliente (o navegador do usuário) se comunicará com *ele* e não com o seu servidor. Por sua vez, provavelmente o servidor proxy se comunicará com o aplicativo usando o HTTP comum (já que o aplicativo e o servidor proxy estarão sendo executados juntos em uma rede confiável). Você ouvirá com frequência falar que o HTTPS *termina* no servidor proxy, ou que o proxy está executando a “terminação SSL”.

Uma vez que você ou seu provedor de hospedagem tiver configurado corretamente o servidor proxy para manipular requisições HTTPS, não será

preciso fazer nenhum trabalho adicional. A exceção a essa regra ocorrerá se sua aplicação tiver de manipular requisições tanto seguras quanto não seguras.

Há três soluções para esse problema. A primeira é simplesmente configurar seu proxy para redirecionar todo o tráfego HTTP para o HTTPS, forçando a comunicação com a aplicação a ocorrer pelo HTTPS. Essa abordagem está se tornando muito mais comum, e é certamente uma solução fácil para o problema.

A segunda abordagem é informar de alguma forma para o servidor qual foi o protocolo usado na comunicação cliente-proxy. A maneira usual de informar isso é com o cabeçalho X-Forwarded-Proto. Por exemplo, para definir esse cabeçalho no NGINX, faça o seguinte:

```
proxy_set_header X-Forwarded-Proto $scheme;
```

Em seguida, em seu aplicativo, você poderia verificar se protocolo foi o HTTPS:

```
app.get('/', (req, res) => {  
  // a parte a seguir é  
  // equivalente a: if(req.secure)  
  if(req.headers['x-forwarded-proto'] === 'https') {  
    res.send('line is secure')  
  } else {  
    res.send('you are insecure!')  
  }  
})
```



No NGINX, há um bloco de configuração server separado para o HTTP e o HTTPS. Se você não definir X-Forwarded-Protocol no bloco de configuração correspondente ao HTTP, dará uma chance para o cliente fazer o spoofing do cabeçalho e levar a aplicação a achar que a conexão é segura ainda que não seja. Se você adotar essa abordagem, certifique-se de *sempre* definir o cabeçalho X-Forwarded-Protocol.

Quando você estiver usando um proxy, o Express fornecerá algumas propriedades de conveniência que tornarão o proxy mais “transparente” (como se ele não estivesse sendo usado, mas sem a perda das vantagens).

Para beneficiar-se disso, diga ao Express que confie no proxy usando `app.enable('trust proxy')`. Depois que o fizer, `req.protocol`, `req.secure` e `req.ip` referenciarão a conexão do cliente com o proxy e não com o seu aplicativo.

Falsificação de requisição entre sites

Os ataques de *falsificação de requisição entre sites* (CSRF, cross-site request forgery) exploram o fato de geralmente os usuários confiarem em seu navegador e visitarem vários sites na mesma sessão. Em um ataque CSRF, o script de um site malicioso faz requisições de outro site: se você tiver feito login no outro site, o site malicioso poderá acessar com sucesso dados de um site diferente.

Para evitar ataques CSRF, você precisa de uma maneira de verificar se uma requisição veio realmente de seu site. O modo de fazer isso é passando um token exclusivo para o navegador. Assim, quando o navegador enviar um formulário, o servidor verificará se o token coincide. O middleware `csrf` manipulará a criação e a verificação de tokens automaticamente; você só tem de se certificar de que o token seja incluído nas requisições enviadas para o servidor. Instale o middleware `csrf` (`npm install csrf`); em seguida, conecte-o e adicione um token a `res.locals`. Certifique-se de conectar o middleware `csrf` após conectar `body-parser`, `cookie-parser` e `express-session`:

```
// essa parte deve vir após conectarmos body-parser,  
// cookie-parser e express-session  
const csrf = require('csrf')
```

```
app.use(csrf({ cookie: true })))  
app.use((req, res, next) => {  
  res.locals._csrfToken = req.csrfToken()  
  next()  
})
```

O middleware `csrf` adiciona o método `csrfToken` ao objeto de requisição. Não precisamos atribuí-lo a `res.locals`; poderíamos passar `req.csrfToken()` explicitamente para todas as views que precisassem dele, porém daria mais trabalho.



Observe que o pacote se chama `csrf`, mas a maioria das variáveis e métodos usa `csrf`, sem o “u”. É fácil se enganar aqui, logo, confira suas vogais!

Agora, em todos os seus formulários (e chamadas AJAX), você terá de fornecer um campo chamado `_csrf`, que deve coincidir com o token gerado. Vejamos como o adicionaríamos a um de nossos formulários:

```
<form action="/newsletter" method="POST">
  <input type="hidden" name="_csrf" value="{ {_csrfToken} }">
  Name: <input type="text" name="name"><br>
  Email: <input type="email" name="email"><br>
  <button type="submit">Submit</button>
</form>
```

O middleware `csrf` manipulará o resto: se o corpo tiver campos, mas não houver um campo `_csrf` válido, ele lançará um erro (certifique-se de ter uma rota de erro em seu middleware!). Remova o campo oculto e veja o que acontece.



Se você tiver uma API, provavelmente não vai querer que o middleware `csrf` a afete. Se quiser restringir o acesso à sua API a partir de outros sites, deve examinar a funcionalidade de “chave de API” de uma biblioteca como o `connect-rest`. Para impedir que o `csrf` interfira no middleware, conecte-o antes de conectar o `csrf`.

Autenticação

A autenticação é um tópico extenso e complicado. Infelizmente, também é uma parte vital da maioria das aplicações web não triviais. O melhor conselho que posso dar é *não tente construí-la por conta própria*. Se seu cartão de visita não exibir “Especialista em Segurança”, você não deve estar preparado para as complexas considerações envolvidas no design de um sistema de autenticação seguro.

Não estou dizendo que você não deva tentar entender os sistemas de segurança de sua aplicação. Só estou recomendando que não tente construí-los. Fique à vontade para estudar o código open source das técnicas de autenticação que vou sugerir. Certamente isso lhe dará pistas de por que não

deve assumir essa tarefa sem ajuda!

Autenticação versus autorização

Embora com frequência os dois termos sejam usados de maneira intercambiada, há uma diferença. *Autenticação* é verificar as identidades dos usuários. Isto é, ver se são quem dizem ser. *Autorização* é determinar o que um usuário está autorizado a acessar, modificar ou visualizar. Por exemplo, poderíamos autorizar os clientes a acessar informações de sua conta, enquanto um funcionário da Meadowlark Travel seria autorizado a acessar as informações de conta ou as observações de vendas de outra pessoa.



A autenticação costuma ser abreviada com *authN* (authentication) e a “autorização” com *authZ* (authorization).

Geralmente (mas nem sempre), a autenticação vem primeiro, e depois a autorização é determinada. A autorização pode ser muito simples (autorizado/não autorizado), ampla (usuário/administrador) ou muito específica, estabelecendo privilégios de leitura, gravação, exclusão e atualização para diferentes tipos de conta. A complexidade do sistema de autorização depende do tipo de aplicação que estiver sendo criada.

Já que a autorização depende tanto dos detalhes da aplicação, fornecerei apenas uma descrição básica neste livro, usando um esquema de autenticação muito amplo (cliente/funcionário). Quase sempre usarei a abreviação “auth”, mas só quando ficar claro pelo contexto se significa “autenticação” ou “autorização”, ou quando não importar.

O problema das senhas

O problema das senhas é que todo sistema de segurança é tão forte quanto seu elo mais fraco. E as senhas requerem que o usuário as invente – e esse é o elo mais fraco. Os humanos são notoriamente desajeitados para inventar senhas seguras. Em uma análise de vulnerabilidades de segurança em 2018, a senha mais popular foi “123456”. “password” foi a segunda. Mesmo no consciente ano de 2018, as pessoas ainda escolhiam senhas muito ruins. A existência de políticas de senha que demandem, por exemplo, uma letra

maiúscula, um número e um sinal de pontuação resultará apenas em uma senha como “Senha1!”.

Até mesmo analisar as senhas comparando-as com as de uma lista de senhas comuns não ajuda muito a resolver o problema. As pessoas começam a anotar suas senhas de melhor qualidade em blocos de notas, a deixá-las em arquivos não criptografados nos computadores ou a enviá-las por email para elas próprias.

No fim das contas, é um problema que você, o designer do aplicativo, não tem muito o que fazer para resolver. No entanto, há coisas que você pode fazer para promover senhas mais seguras. Uma é passar o problema adiante e deixar que terceiros façam a autenticação. A outra é tornar seu sistema de login amigável para serviços de gerenciamento de senha, como o 1Password, Bitwarden e LastPass.

Autenticação de terceiros

A autenticação de terceiros se beneficia do fato de que quase todo mundo na internet tem uma conta em pelo menos um serviço importante, como Google, Facebook, Twitter ou LinkedIn. Todos esses serviços fornecem um mecanismo que autentica e identifica seus usuários.



A autenticação de terceiros é com frequência chamada de *autenticação federada* ou *autenticação delegada*. Os termos são intercambiáveis, embora geralmente a autenticação federada seja associada ao Security Assertion Markup Language (SAML) e ao OpenID, e a associação delegada seja associada ao OAuth.

Há três grandes vantagens na autenticação de terceiros. A primeira é que nosso trabalho para autenticar diminui. Não temos de nos preocupar com a autenticação de usuários individuais, apenas interagir com um terceiro confiável. A segunda vantagem é que reduz a *fadiga de senhas*: o estresse associado a possuir muitas contas. Uso o *LastPass* (<http://lastpass.com>) e acabei de verificar meu cofre de senhas: tenho quase 400 senhas. Como profissional de tecnologia, devo ter mais senhas do que o usuário médio da internet, mas não é raro até mesmo para um usuário casual ter dúzias ou talvez centenas de contas. Para concluir, a autenticação de terceiros *não*

causa atritos: ela permite que os usuários comecem a usar um site mais rapidamente, com credenciais que eles já têm. Com frequência, quando os usuários veem que tem de criar *outro* nome de usuário e senha, eles simplesmente mudam de local.

Se você não usa um gerenciador de senhas, as chances são de que esteja usando a mesma senha para a maioria dos sites (quase todas as pessoas têm uma senha “segura” que elas usam para serviços bancários e tarefas semelhantes, e uma senha insegura que usam para o resto). O problema dessa abordagem é que se *um* dos sites para os quais você usa essa senha for atacado, e sua senha tornar-se conhecida, os hackers tentarão usar essa mesma senha em outros serviços. É como colocar todos os ovos na mesma cesta.

A autenticação de terceiros tem suas desvantagens. Pode ser difícil de acreditar, mas *há* pessoas que não têm uma conta no Google, Facebook, Twitter ou LinkedIn. Além disso, entre as pessoas que *têm* essas contas, a desconfiança (ou o desejo de privacidade) pode levá-las a não querer usar essas credenciais para fazer login em nosso site. Muitos sites resolvem esse problema específico encorajando os usuários a usar uma conta existente, mas quem não as tiver (ou não quiser usá-las) pode criar um novo login para acessar o serviço.

Armazenando usuários em seu banco de dados

Usando ou não o serviço de terceiros para fazer a autenticação, provavelmente você vai querer armazenar um registro dos usuários em seu próprio banco de dados. Por exemplo, se estiver usando o Facebook para a autenticação, ela só verificará a identidade do usuário. Se você precisar ter configurações específicas para esse usuário, não poderá usar o Facebook para isso: terá de armazenar informações sobre o usuário em seu banco de dados. Você também pode querer associar um endereço de email aos usuários, e talvez eles não queiram usar o mesmo endereço que usam no Facebook (ou qualquer que seja o serviço de autenticação de terceiros que você estiver utilizando). Por fim, armazenar informações de usuários em seu banco de dados permitirá que você mesmo execute a autenticação, se quiser fornecer

essa opção.

Criaremos então um modelo para nossos usuários, *models/user.js*:

```
const mongoose = require('mongoose')

const userSchema = mongoose.Schema({
  authId: String,
  name: String,
  email: String,
  role: String,
  created: Date,
})

const User = mongoose.model('User', userSchema)
module.exports = User
```

E modificaremos *db.js* com as abstrações apropriadas (se você estiver usando o PostgreSQL, deixarei como exercício incluir essa abstração):

```
const User = require('./models/user')

module.exports = {
  //...
  getUserById: async id => User.findById(id),
  getUserByAuthId: async authId => User.findOne({ authId }),
  addUser: async data => new User(data).save(),
}
```

Lembre-se de que cada objeto de um banco de dados MongoDB tem o próprio ID exclusivo, armazenado na propriedade `_id`. No entanto, esse ID é controlado pelo MongoDB, e precisamos de alguma maneira de mapear um registro de usuário para um ID de terceiros, logo, temos nossa própria propriedade de ID, chamada `authId`. Já que usaremos várias estratégias de autenticação, esse ID será uma combinação de um tipo de estratégia e um ID de terceiros, para evitarmos colisões. Por exemplo, um usuário do Facebook poderia ter um `authId` igual a `facebook:525764102`, enquanto um do Twitter teria um `authId` como `twitter:376841763`.

Usaremos dois roles em nosso exemplo: “cliente” e “funcionário”.

Autenticação versus registro e a experiência do usuário

Autenticar é verificar a identidade do usuário, com a autenticação de terceiros ou com as credenciais fornecidas (como nome de usuário e senha). Registrar-se é o processo pelo qual um usuário obtém uma conta no site (no nosso caso, o ato de se registrar ocorre quando criamos um registro de usuário no banco de dados).

Quando os usuários ingressarem em seu site pela primeira vez, deve ficar claro para eles que estão se registrando. Usando um sistema de autenticação de terceiros, poderíamos registrá-los sem seu conhecimento se eles conseguirem ser autenticados com sucesso. Geralmente isso não é considerado boa prática, logo, devemos deixar claro para os usuários que eles estão se registrando no site (sendo ou não autenticados por terceiros) e fornecer um mecanismo fácil para o cancelamento de sua associação.

Uma situação de experiência de usuário que devemos considerar é a “confusão na autenticação de terceiros”. Se um usuário se registrar em janeiro em um serviço usando o Facebook, retornar em julho, e deparar com uma tela oferecendo as opções de login com o Facebook, Twitter, Google ou LinkedIn, ele pode não se lembrar do serviço de registro que foi originalmente usado. Essa é uma das armadilhas da autenticação de terceiros, e não há muita coisa que possamos fazer sobre isso. É outra boa razão para solicitarmos ao usuário que forneça um endereço de email: dessa forma, podemos lhe dar a opção de procurar sua conta pelo email, e enviar um email para esse endereço especificando que serviço foi usado para a autenticação.

Se você sabe que redes sociais seus usuários utilizam, pode amenizar esse problema tendo um serviço de autenticação primário. Por exemplo, se tem certeza de que a maioria dos usuários tem uma conta no Facebook, poderia ter um botão grande exibindo “Login com o Facebook” e depois usar botões menores ou até mesmo apenas links de texto com “ou login com o Google, Twitter ou LinkedIn”. Essa abordagem pode ajudar no caso da confusão na autenticação de terceiros.

Passport

O *Passport* é um módulo de autenticação muito popular e robusto para o

Node/Express. Ele não está associado a nenhum outro mecanismo de autenticação; em vez disso, se baseia na ideia de *estratégias* de autenticação plugáveis (inclusive uma estratégia local se você não quiser usar a autenticação de terceiros). Pode ser difícil entender o fluxo de informações de autenticação, logo, começaremos usando apenas um mecanismo de autenticação e adicionaremos outros depois.

O detalhe que é importante entender é que, com a autenticação de terceiros, o aplicativo *nunca recebe uma senha*. Esse detalhe é totalmente manipulado pelo grupo de fora. Isso é algo bom: passa a responsabilidade da manipulação e do armazenamento seguros de senhas para terceiros.¹

O processo inteiro se baseia em redirecionamentos (é preciso, já que a aplicação nunca recebe a senha de terceiros do usuário). Inicialmente, talvez você fique confuso pelo fato de poder passar URLs *localhost* para terceiros e mesmo assim se autenticar com sucesso (afinal, o servidor de terceiros que manipula a requisição não conhece *seu localhost*). Funciona porque a autenticação externa instrui o *navegador* a fazer o redirecionamento, e por esse estar dentro de sua rede, pode fazer o redirecionamento para endereços locais.

O fluxo básico é mostrado na Figura 18.1. Esse diagrama mostra o importante fluxo de funcionalidades, deixando claro que a autenticação ocorre realmente no site de terceiros. Aprecie a simplicidade do diagrama – as coisas vão ficar muito mais complicadas.

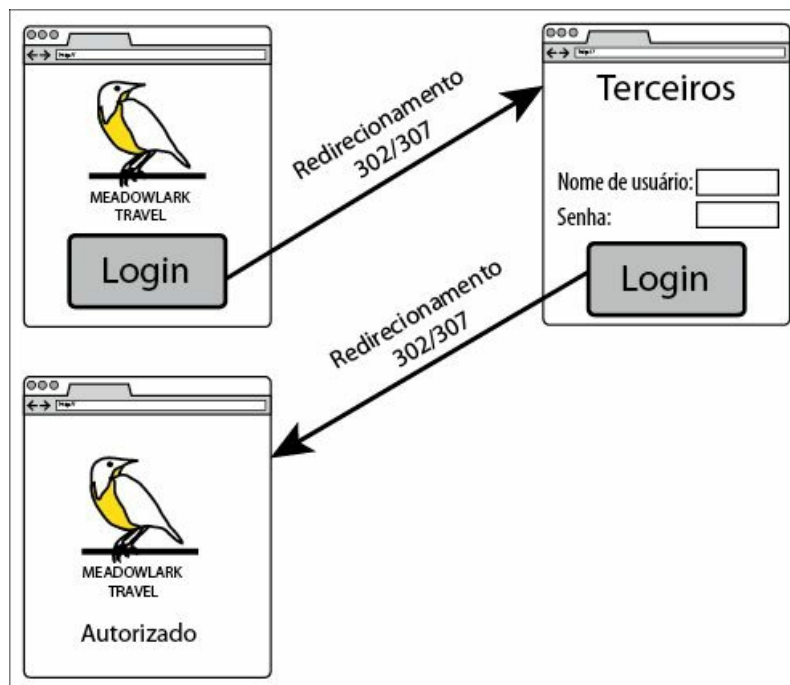


Figura 18.1 – Fluxo da autenticação de terceiros.

Quando você usar o Passport, seu aplicativo será responsável por quatro etapas. Considere uma visão mais detalhada do fluxo da autenticação de terceiros, como mostrado na Figura 18.2.

Para simplificar, estamos usando a Meadowlark Travel na representação do aplicativo, e o Facebook para o mecanismo de autenticação de terceiros. A Figura 18.2 ilustra como o usuário vai da página de login para a página segura Account Info (estamos usando a página Account Info somente a título de ilustração: poderia ser qualquer página de seu site que demandasse autenticação).

Esse diagrama mostra detalhes nos quais normalmente não pensamos, mas é importante entendê-los nesse contexto. Especificamente, ao visitar uma URL, *você* não estará fazendo a requisição do servidor: o navegador é que a fará. Dito isso, o navegador pode fazer três coisas: fazer uma requisição HTTP, exibir a resposta e executar um redirecionamento (que é basicamente fazer outra requisição e exibir outra resposta... que por sua vez poderia ser outro redirecionamento).

Na coluna Meadowlark, você pode ver as quatro etapas pelas quais sua aplicação é responsável. Felizmente, usaremos o Passport (e estratégias

plugáveis) para executar os detalhes dessas etapas; caso contrário, este livro seria muito mais longo.

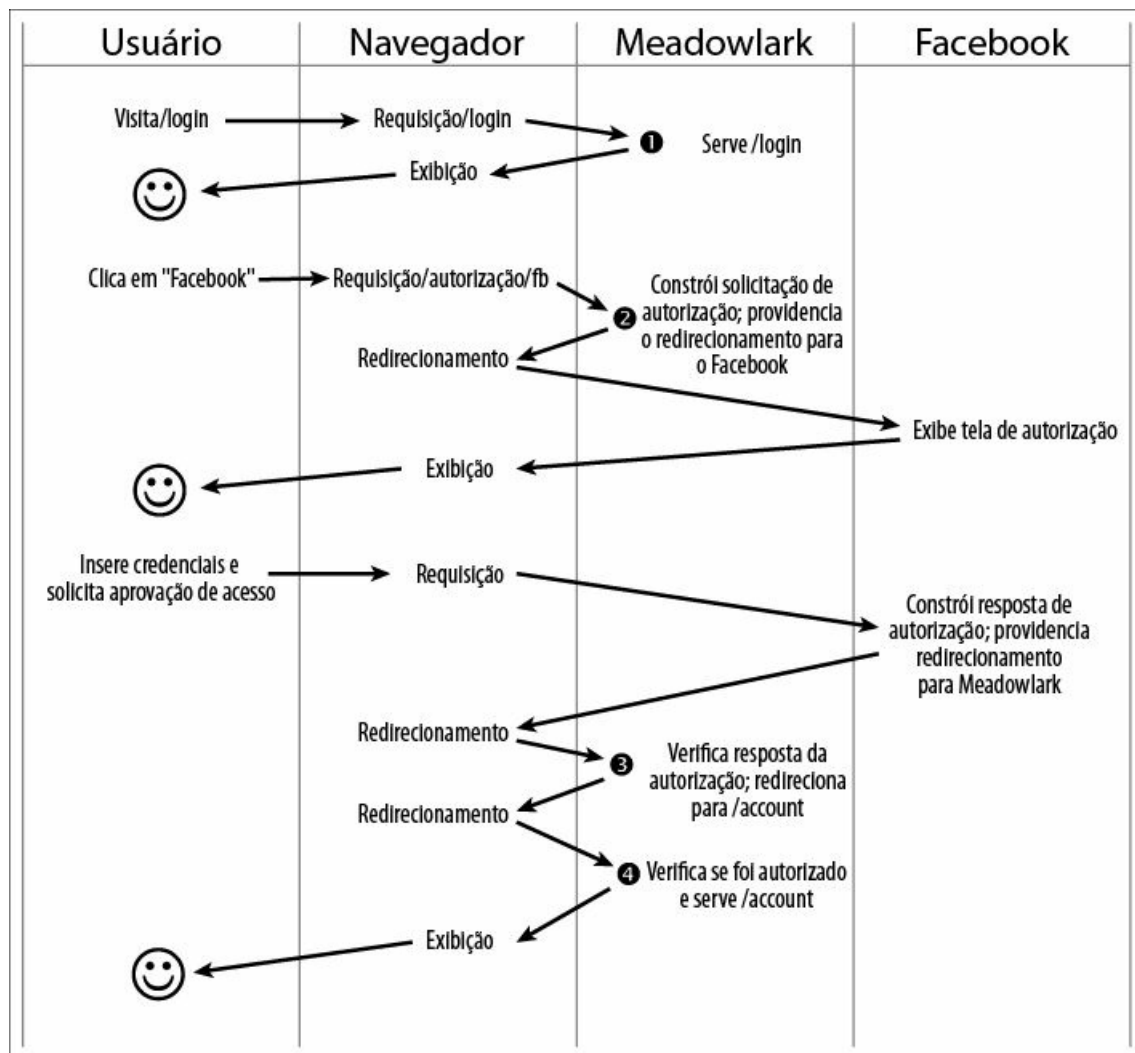


Figura 18.2 – Visão detalhada do fluxo da autenticação de terceiros.

Antes de vermos as particularidades da implementação, consideraremos cada uma das etapas com mais detalhes:

Página de login

A página de login é onde o usuário poderá selecionar o método de login. Quando é usada uma autenticação de terceiros, geralmente se trata apenas de um botão ou um link. Se usarmos a autenticação local, ela incluirá os campos de nome de usuário e senha. Se o usuário tentar acessar uma URL que demande autenticação (como /account em nosso exemplo) sem ter feito

login, provavelmente essa será a página para qual você vai querer redirecioná-lo (alternativamente, poderíamos redirecioná-lo para uma página Not Authorized com um link para a página de login).

Constrói requisição de autorização

Nessa etapa, você construirá uma requisição para ser enviada para terceiros (por meio de um redirecionamento). Os detalhes dessa requisição são complicados e específicos da estratégia de autenticação. O Passport (e o plugin da estratégia) se encarregará da parte difícil. A requisição de autorização inclui a proteção contra ataques de intermediário (man-in-the-middle attacks), assim como contra outros vetores que um invasor poderia explorar. Geralmente a requisição tem tempo de vida curto, logo, não podemos armazená-la para tentar usar posteriormente: isso ajuda a evitar ataques limitando o período durante o qual um invasor teria tempo para agir. É nesse momento que você pode solicitar informações adicionais a partir do mecanismo de autorização de terceiros. Por exemplo, é comum solicitar o nome do usuário, e possivelmente o endereço de email. Lembre-se de que, quanto maior o número de informações que você solicitar aos usuários, menores as chances de haver autorização para sua aplicação.

Verifica resposta da autorização

Supondo que o usuário tenha recebido autorização para sua aplicação, você receberá uma resposta de autorização válida, que servirá como prova da identidade do usuário. Novamente, os detalhes dessa validação são complicados e serão manipulados pelo Passport (e pelo plugin da estratégia). Se a resposta de autorização indicar que o usuário não está autorizado (se credenciais inválidas forem inseridas ou se a aplicação não conceder autorização ao usuário), faremos o redirecionamento para uma página apropriada (voltando à página de login ou indo para uma página Not Authorized ou Unable to Authorize). Na resposta da autorização estará incluída uma ID para o usuário que será exclusiva *de terceiros*, assim como qualquer detalhe adicional que você tiver solicitado na etapa 2. Para executar a etapa 4, precisamos de um modo de “nos lembrar” de que o usuário está autorizado. A maneira usual de fazer isso é definindo uma

variável de sessão contendo o ID do usuário, o que indicará que essa sessão foi autorizada (cookies também podem ser usados, mas recomendo usar sessões).

Verifica autorização

Na etapa 3, armazenamos um ID de usuário na sessão. A presença desse ID permite recuperar um objeto de usuário no banco de dados contendo informações sobre o que o usuário está autorizado a fazer. Dessa forma, não precisaremos executar a autenticação de terceiros a cada requisição (o que resultaria em uma experiência de usuário lenta e incômoda). Essa tarefa é simples e não precisamos mais do Passport: temos nosso próprio objeto de usuário que contém nossas regras de autenticação. (Se esse objeto não estiver disponível, isso indicará que a requisição não foi autorizada e poderemos fazer o redirecionamento para a página de login ou para Not Authorized).



Dá trabalho usar o Passport na autenticação, como você verá neste capítulo. No entanto, a autenticação é uma parte importante da aplicação e acho apropriado investir algum tempo para fazê-la funcionar corretamente. Há projetos como o *LockIt* (http://bit.ly/lock_it) que tentam fornecer uma solução mais “pronta para uso”. Outra opção cada vez mais popular é o *Auth0* (<https://auth0.com>), que é muito robusto, mas não é tão fácil de configurar como o LockIt. Porém, para usar o LockIt ou o Auth0 (ou soluções semelhantes) mais eficientemente, você precisa entender os detalhes da autenticação e da autorização, e é para isso que este capítulo foi criado. Se tiver de personalizar uma solução de autenticação, o Passport é um ótimo ponto de partida.

Configurando o Passport

Para simplificar, começaremos com um único provedor de autenticação. Selecionaremos arbitrariamente o Facebook. Antes de preparar o Passport e a estratégia do Facebook, teremos de lidar com algumas configurações no Facebook. Para usar a autenticação do Facebook, você precisará de um *aplicativo Facebook*. Se já tiver um aplicativo Facebook adequado, use-o, ou crie um novo especificamente para a autenticação. Se possível, você deve

usar a conta oficial do Facebook de sua empresa para criar o aplicativo. Isto é, se você trabalhasse para a Meadowlark Travel, usaria a conta do Facebook dessa empresa para criar o aplicativo (também é possível adicionar nossa própria conta do Facebook como a de administração para facilitar o gerenciamento). Para fins de teste, não há problema em usarmos nossa conta pessoal do Facebook, mas utilizar uma conta pessoal para a produção parecerá amador e suspeito para os usuários.

Os detalhes de administração do aplicativo Facebook parecem mudar com muita frequência, logo, não vou explicá-los aqui. Consulte a *documentação de desenvolvedor do Facebook* (<http://bit.ly/372bc7c>) se precisar de detalhes sobre a criação e a administração de seu aplicativo.

Para fins de desenvolvimento e teste, você terá de associar o nome de domínio de desenvolvimento/teste ao aplicativo. O Facebook permite usar *localhost* (e números de porta), o que é ótimo para fins de teste. Alternativamente, é possível especificar um endereço IP local, o que pode ser útil se você estiver usando um servidor virtualizado, ou outro servidor em sua rede para teste. O importante é que a URL que você inserir em seu navegador para o teste (por exemplo, <http://localhost:3000>) esteja associada ao aplicativo Facebook. Atualmente, só podemos associar um domínio ao aplicativo; se você precisar usar vários domínios, terá de criar múltiplos aplicativos (poderíamos ter Meadowlark Dev, Meadowlark Test e Meadowlark Staging; seu aplicativo de produção poderia se chamar simplesmente Meadowlark Travel).

Uma vez que você tiver configurado seu aplicativo, precisará de seu ID exclusivo e de seu segredo (*secret*), e os dois podem ser encontrados na página de gerenciamento de aplicativos do Facebook referente a esse aplicativo.



Provavelmente uma das maiores decepções que você terá será receber uma mensagem do Facebook como “Given URL is not allowed by the Application configuration” (A URL fornecida não é permitida pela configuração da aplicação). Isso indica que o nome de host e a porta da URL de callback não coincidem com o que foi configurado em seu aplicativo. Se você examinar o navegador, verá a URL codificada, o que

deve lhe dar uma pista. Por exemplo, se eu estiver usando 192.168.0.103:3443, e receber essa mensagem, examinarei a URL. Se deparar com `redirect_uri=https%3A%2F%2F192.68.0.103%3A3443%2Fauth%2Ffacebook%2Fcall` na querystring, será fácil detectar o erro: usei 68 em vez de 168 no nome de host.

Agora instalaremos o Passport e a estratégia de autenticação do Facebook:

```
npm install passport passport-facebook
```

Antes de terminarmos, haverá um grande volume de código de autenticação (principalmente se estivermos dando suporte a várias estratégias) e não queremos desorganizar *meadowlark.js* com todo esse código. Em vez disso, criaremos um módulo chamado *lib/auth.js*. Será um arquivo grande, logo, vamos manipulá-lo pedaço a pedaço (consulte *ch18* no repositório fornecido para ver o exemplo final). Começaremos com as importações e dois métodos que o Passport demanda, `serializeUser` e `deserializeUser`:

```
const passport = require('passport')
const FacebookStrategy = require('passport-facebook').Strategy

const db = require('../db')

passport.serializeUser((user, done) => done(null, user._id))

passport.deserializeUser((id, done) => {
  db.getUserById(id)
    .then(user => done(null, user))
    .catch(err => done(err, null))
})
```

O Passport usa `serializeUser` e `deserializeUser` para mapear requisições para o usuário autenticado, o que nos permite empregar o método de armazenamento que quisermos. Em nosso caso, só armazenaremos o ID do banco de dados (a propriedade `_id`) na sessão. A maneira como estamos usando o ID aqui torna “serialize” e “deserialize” designações erradas: na verdade estamos apenas armazenando um ID de usuário na sessão. Assim, quando precisarmos, poderemos obter um objeto de usuário procurando esse ID no banco de dados.

Uma vez que esses dois métodos forem implementados, contanto que haja uma sessão ativa, e o usuário tenha se autenticado com sucesso, `req.session.passport.user` será o objeto de usuário correspondente como recuperado no banco de dados.

Em seguida, selecionaremos o que vamos exportar. Para ativar a funcionalidade do Passport, teremos de executar duas atividades distintas: inicializar o Passport e registrar as rotas que manipularão a autenticação e os callbacks de redirecionamento a partir dos serviços de autenticação de terceiros. Não queremos combinar essas duas atividades na mesma função porque, no arquivo principal da aplicação, podemos querer selecionar quando o Passport será incluído na cadeia de middleware (lembre-se de que a ordem é relevante na inclusão de middleware). Logo, em vez de fazermos nosso módulo exportar uma função que execute uma dessas duas atividades, faremos com que ele retorne uma função contendo os métodos necessários. Por que não retornar apenas um objeto? Porque precisamos incorporar alguns valores de configuração. Além disso, já que temos de incluir o middleware Passport em nossa aplicação, uma função é uma maneira fácil de passar o objeto de aplicação do Express:

```
module.exports = (app, options) => {  
  // se redirecionamentos de sucesso e falha não  
  // forem especificados, defina alguns padrões  
  if(!options.successRedirect) options.successRedirect = '/account'  
  if(!options.failureRedirect) options.failureRedirect = '/login'  
  return {  
    init: function() { /* A CONCLUIR */ },  
    registerRoutes: function() { /* A CONCLUIR */ },  
  }  
}
```

Antes de entrar nos detalhes dos métodos `init` e `registerRoutes`, examinaremos como esse módulo será usado (espero que isso torne essa abordagem de retornar uma função que retorna um objeto um pouco mais clara):

```
const createAuth = require('./lib/auth')  
  
// ...outras configurações do aplicativo
```

```
const auth = createAuth(app, {  
  // baseUrl é opcional; ela usará como padrão localhost se você  
  // a omitir; pode ser útil defini-la se você não estiver trabalhando  
  // em sua máquina local. Por exemplo, se você estivesse em um servidor  
  // de teste, poderia configurar a variável de ambiente BASE_URL com  
  // https://staging.meadowlark.com  
  baseUrl: process.env.BASE_URL,  
  providers: credentials.authProviders,  
  successRedirect: '/account',  
  failureRedirect: '/unauthorized',  
})
```

```
// auth.init() conecta o middleware Passport:  
auth.init()
```

```
// agora podemos especificar nossas rotas de autenticação:  
auth.registerRoutes()
```

Observe que, além de especificar os paths de redirecionamento de sucesso e falha, também especificamos uma propriedade chamada `providers`, que exteriorizamos no arquivo de credenciais (consulte o Capítulo 13). Teremos de adicionar a propriedade `authProviders` a `.credentials.development.json`:

```
"authProviders": {  
  "facebook": {  
    "appId": "your_app_id",  
    "appSecret": "your_app_secret"  
  }  
}
```



Outra razão para empacotarmos nosso código de autenticação em um módulo como esse seria para reutilizá-lo em outros projetos; na verdade, já existem alguns pacotes de autenticação que fazem basicamente o que estamos fazendo aqui. No entanto, é importante entender os detalhes do que está ocorrendo, logo, mesmo se você acabar usando um módulo que outra pessoa criou, isso o ajudará a entender tudo que está ocorrendo em seu fluxo de autenticação.

Agora cuidaremos de nosso método `init` (anteriormente definido como “A CONCLUIR” em `auth.js`):

```

init: function() {
  var config = options.providers

  // configura a estratégia do Facebook
  passport.use(new FacebookStrategy({
    clientID: config.facebook.appId,
    clientSecret: config.facebook.appSecret,
    callbackURL: (options.baseUrl || "") + '/auth/facebook/callback',
  }, (accessToken, refreshToken, profile, done) => {
    const authId = 'facebook:' + profile.id
    db.getUserByAuthId(authId)
      .then(user => {
        if(user) return done(null, user)
        db.addUser({
          authId: authId,
          name: profile.displayName,
          created: new Date(),
          role: 'customer',
        })
        .then(user => done(null, user))
        .catch(err => done(err, null))
      })
      .catch(err => {
        if(err) return done(err, null);
      })
  })))

  app.use(passport.initialize())
  app.use(passport.session())
},

```

Esse é um fragmento de código um pouco complexo, mas grande parte dele é na verdade apenas boilerplate do Passport. A parte importante está dentro da função que é passada para a instância de FacebookStrategy. Quando essa função é chamada (após o usuário ter se autenticado com sucesso), o parâmetro profile contém informações sobre o usuário do Facebook. E o que é mais importante, ela inclui um ID do Facebook: que é o que usaremos para associar uma conta do Facebook a nosso objeto de usuário. Observe que incluímos a propriedade authId em um namespace usando como prefixo

facebook:. Embora as chances sejam pequenas, isso evitará que um ID do Facebook colida com um ID do Twitter ou do Google (também nos permite examinar modelos de usuário para ver que método de autenticação um usuário está usando, o que pode ser útil). Se o banco de dados já contiver uma entrada para o ID que foi incluído no namespace, vamos apenas retorná-la (é aí que `serializeUser` será chamada, inserindo nosso ID de usuário na sessão). Se nenhum registro de usuário for retornado, criaremos um novo objeto de usuário e o salvaremos no banco de dados.

A última coisa que temos de fazer é criar nosso método `registerRoutes` (não se preocupe, esse é muito mais curto):

```
registerRoutes: () => {
  app.get('/auth/facebook', (req, res, next) => {
    if(req.query.redirect) req.session.authRedirect = req.query.redirect
    passport.authenticate('facebook')(req, res, next)
  })
  app.get('/auth/facebook/callback', passport.authenticate('facebook',
    { failureRedirect: options.failureRedirect })),
    (req, res) => {
      // só chegaremos aqui se a autenticação for bem-sucedida
      const redirect = req.session.authRedirect
      if(redirect) delete req.session.authRedirect
      res.redirect(303, redirect || options.successRedirect)
    }
  )
},
```

Agora temos o path `/auth/facebook`; uma visita a esse path redirecionará o visitante automaticamente para a tela de autenticação do Facebook (isso é feito por `passport.authenticate('facebook')`), a etapa 2 da Figura 18.1. Observe que verificamos se há um parâmetro de querystring `redirect`; se houver, vamos salvá-lo na sessão. Faremos isso a fim de executar o redirecionamento automaticamente para o destino pretendido após a conclusão da autenticação. Quando o usuário for autorizado no Facebook, o navegador será redirecionado de volta ao seu site – especificamente para o path `/auth/facebook/callback` (com a querystring opcional `redirect` indicando onde o usuário estava originalmente).

Na querystring também há tokens de autenticação que o Passport verificará. Se a verificação falhar, o Passport redirecionará o navegador para `options.failureRedirect`. Se a verificação for bem-sucedida, o Passport chamará `next`, e a aplicação será trazida de volta. Observe como o middleware é encadeado para o manipulador de `/auth/facebook/callback`: `passport.authenticate` é chamado primeiro. Se ele chamar `next`, o controle passará para nossa função, que fará o redirecionamento para o local original ou para `options.successRedirect`, se o parâmetro de querystring `redirect` não tiver sido especificado.



Omitir o parâmetro de querystring `redirect` pode simplificar nossas rotas de autenticação, o que pode ser tentador se você tiver uma única URL que demande autenticação. No entanto, é útil ter essa funcionalidade disponível e ela fornece uma melhor experiência para o usuário. É claro que você já deve ter passado por isso: encontrou a página que queria e foi instruído a fazer login. Fez login, foi redirecionado para uma página padrão e teve de navegar de volta à página original. Não é uma experiência de usuário muito satisfatória.

A “mágica” que o Passport está fazendo durante esse processo é salvar o usuário (em nosso caso, apenas um ID de usuário do banco de dados) na sessão. Isso é algo bom, porque o navegador está fazendo o *redirecionamento*, que é uma requisição HTTP diferente: se não tivéssemos essa informação na sessão, não teríamos como saber que o usuário foi autenticado! Quando um usuário se autenticar com sucesso, `req.session.passport.user` será definido, e é assim que futuras requisições saberão que o usuário foi autenticado.

Examinaremos o manipulador de `/account` para ver como ele verifica se o usuário foi autenticado (esse manipulador de rota estará no arquivo principal da aplicação, ou em um módulo de roteamento separado, e não em `/lib/auth.js`):

```
app.get('/account', (req, res) => {  
  if(!req.user)  
    return res.redirect(303, '/unauthorized')  
  res.render('account', { username: req.user.name })  
})
```

```
// também precisamos de uma página 'unauthorized'
app.get('/unauthorized', (req, res) => {
  res.status(403).render('unauthorized')
})
// e de uma maneira de fazer logout
app.get('/logout', (req, res) => {
  req.logout()
  res.redirect('/')
})
```

Agora só usuários autenticados verão a página da conta; todos os outros serão redirecionados para uma página Not Authorized.

Autorização baseada em roles

Até aqui, tecnicamente não estamos executando nenhuma autorização (só estamos distinguindo usuários autorizados dos não autorizados). Porém, digamos que quiséssemos que somente os clientes vissem as suas views de conta (os funcionários poderiam ter uma view totalmente diferente onde veriam as informações de conta dos usuários).

Lembre-se de que a mesma rota pode ter várias funções, que serão chamadas em ordem. Criaremos uma função chamada `customerOnly` que só permitirá clientes:

```
const customerOnly = (req, res, next) => {
  if(req.user && req.user.role === 'customer') return next()
  // queremos páginas somente de clientes para que eles
  // saibam que precisam fazer login
  res.redirect(303, '/unauthorized')
}
```

Também criaremos uma função `employeeOnly` que operará de maneira um pouco diferente. Suponhamos que tivéssemos um path `/sales` que só quiséssemos disponibilizar para funcionários. Além disso, não queremos que não funcionários saibam de sua existência, mesmo se por acaso o virem. Se um possível invasor acessasse o path `/sales`, e visse uma página Not Authorized, essa seria uma informação menor que poderia facilitar um ataque (pelo simples conhecimento de sua existência). Logo, para termos alguma segurança adicional, queremos que não funcionários vejam uma página 404

comum quando visitarem a página `/sales`, e assim não daremos nada para possíveis invasores explorarem:

```
const employeeOnly = (req, res, next) => {  
  if(req.user && req.user.role === 'employee') return next()  
  // queremos que falhas de autorização de funcionários sejam  
  // "ocultadas", para que possíveis invasores não saibam nem  
  // mesmo que essa página existe  
  next('route')  
}
```

Uma chamada a `next('route')` não executará simplesmente o próximo manipulador na rota: ela pulará essa rota. Supondo que não haja outra rota abaixo que manipule `/account`, esse código passará para o manipulador de erro 404, fornecendo o resultado desejado.

Veja como é fácil colocar essas funções em ação:

```
// rotas de clientes
```

```
app.get('/account', customerOnly, (req, res) => {  
  res.render('account', { username: req.user.name })  
})  
app.get('/account/order-history', customerOnly, (req, res) => {  
  res.render('account/order-history')  
})  
app.get('/account/email-prefs', customerOnly, (req, res) => {  
  res.render('account/email-prefs')  
})
```

```
// rotas de funcionários
```

```
app.get('/sales', employeeOnly, (req, res) => {  
  res.render('sales')  
})
```

Deve ter ficado claro que, se quisermos, podemos ter uma autorização baseada em roles mais simples ou mais complicada. Por exemplo, e se você quisesse dar permissão a vários roles? Poderia usar a função e a rota a seguir:

```
const allow = roles => (req, res, next) => {  
  if(req.user && roles.split(',').includes(req.user.role)) return next()
```



```
    res.redirect(303, '/unauthorized')
  }
```

Espero que esse exemplo tenha lhe dado uma ideia de como você pode ser criativo na autorização baseada em roles. Você poderia executar a autorização com base em outras propriedades, como por quanto tempo um usuário é membro ou quantos pacotes de férias o usuário reservou.

Adicionando provedores de autenticação

Agora que nosso framework está definido, será fácil adicionar mais provedores de autenticação. Suponhamos que quiséssemos fazer a autenticação com o Google. Antes de começar a adicionar código, você terá de definir um projeto em sua conta do Google.

Acesse seu *Google Developers Console* (<http://bit.ly/2KcY1X0>) e selecione um projeto na barra de navegação (se ainda não tiver um projeto, clique em New Project e siga as instruções). Após selecionar um projeto, clique em “Enable APIs and Services” e ative Cloud Identity API. Clique em Credentials e em Create Credentials e, em seguida, selecione “OAuth client ID” e “Web application”. Insira as URLs apropriadas para seu aplicativo: no caso de teste você pode usar *http://localhost:3000* para as origens autorizadas, e *http://localhost:3000/auth/google/callback* para as URIs de redirecionamento autorizadas.

Quando você estiver com tudo definido no lado do Google, execute `npm install passport-google-oauth20` e adicione o código a seguir a *lib/auth.js*:

```
// configura a estratégia do Google
passport.use(new GoogleStrategy({
  clientID: config.google.clientID,
  clientSecret: config.google.clientSecret,
  callbackURL: (options.baseUrl || '') + '/auth/google/callback',
}, (token, tokenSecret, profile, done) => {
  const authId = 'google:' + profile.id
  db.getUserByAuthId(authId)
    .then(user => {
      if(user) return done(null, user)
      db.addUser({
        authId: authId,
```

```

    name: profile.displayName,
    created: new Date(),
    role: 'customer',
  })
  .then(user => done(null, user))
  .catch(err => done(err, null))
})
.catch(err => {
  console.log('whoops, there was an error: ', err.message)
  if(err) return done(err, null);
})
}))

```

E o código a seguir ao método `registerRoutes`:

```

app.get('/auth/google', (req, res, next) => {
  if(req.query.redirect) req.session.authRedirect = req.query.redirect
  passport.authenticate('google', { scope: ['profile'] })(req, res, next)
})
app.get('/auth/google/callback', passport.authenticate('google',
  { failureRedirect: options.failureRedirect })),
(req, res) => {
  // só chegaremos aqui se a autenticação for bem-sucedida
  const redirect = req.session.authRedirect
  if(redirect) delete req.session.authRedirect
  res.redirect(303, req.query.redirect || options.successRedirect)
}
)

```

Conclusão

Parabéns por ter chegado ao fim do capítulo mais complicado do livro! É uma pena que um recurso tão importante (autenticação e autorização) seja tão complicado, mas em um mundo cheio de ameaças à segurança, essa é uma complexidade inevitável. Felizmente, projetos como o Passport (e os excelentes esquemas de autenticação baseados nele) facilitam um pouco o trabalho. Mesmo assim, recomendo que você não seja negligente com essa área de sua aplicação: ser zeloso na área de segurança o tornará um bom cidadão na internet. Seus usuários podem não lhe agradecer por isso, mas

proprietários de aplicações que permitam que dados de usuários fiquem comprometidos devido a falhas na segurança terão um preço a pagar.

1 Também é improvável que terceiros armazenem senhas. Uma senha pode ser verificada pelo armazenamento de algo chamado *hash salgado* (salted hash), que é uma transformação unidirecional da senha. Isto é, uma vez que você gerar um hash a partir de uma senha, não poderá recuperá-la. *Salgar* o hash fornece proteção adicional contra certos tipos de ataques.

Fazendo a integração com APIs de terceiros

Cada vez mais os sites de sucesso não são totalmente autônomos. Na fidelização dos usuários existentes e na busca de novos usuários, a integração com redes sociais é obrigatória. Para o fornecimento de localizadores de lojas ou outros serviços de localização, o uso de geolocalização e mapeamento é essencial. E não para aí: um número crescente de empresas está percebendo que o fornecimento de uma API ajuda a expandir seus serviços e os torna mais úteis.

Neste capítulo, discutiremos as duas necessidades de integração mais comuns: mídias sociais e geolocalização.

Mídias sociais

As mídias sociais são uma ótima maneira de promover um produto ou serviço: se é esse o seu objetivo, a possibilidade de seus usuários compartilharem facilmente seu conteúdo nas mídias sociais é essencial. Quando escrevi este texto, os serviços de rede social predominantes eram o Facebook, o Twitter, o Instagram e o YouTube. Sites como o Pinterest e o Flickr também são importantes, mas geralmente têm um público um pouco mais específico (por exemplo, se seu site for sobre artesanato, você deve dar suporte ao Pinterest). Pode rir se quiser, mas prevejo a volta do MySpace. Seu novo design é inspirado, e vale a pena mencionar que o MySpace se baseou no Node.

Plugins de mídias sociais e desempenho do site

Grande parte da integração com mídias sociais está ligada ao front-end. Referenciamos os arquivos JavaScript apropriados na página, e isso permite tanto a chegada (os três principais stories de sua página do Facebook, por exemplo) quanto a saída (a possibilidade de tuitar sobre a página em que você está) de conteúdo. Embora com frequência esse seja o caminho mais fácil de integração com mídias sociais, ele tem um custo: vi os tempos de carregamento de páginas dobrarem ou até mesmo triplicarem devido ao aumento de requisições HTTP. Se o desempenho da página for importante (e ele deve ser, principalmente para usuários móveis), você precisa considerar cuidadosamente como integrará as mídias sociais.

Dito isso, o código que ativa um botão Like do Facebook ou um botão do Twitter se beneficia de cookies do navegador para fazer postagens em nome do usuário. Seria difícil (e, em alguns casos, impossível) passar essa funcionalidade para o back-end. Logo, se você precisa dela, usar uma biblioteca de terceiros apropriada é a melhor opção, ainda que possa afetar o desempenho da página.

Procurando tuítes

Digamos que quiséssemos mencionar os 10 tuítes mais recentes que tivessem as hashtags #Oregon #travel. Poderíamos usar um componente de front-end para fazê-lo, mas isso envolveria requisições HTTP adicionais. No entanto, se o fizéssemos no back-end, teríamos a opção de armazenar os tuítes em cache para não afetar o desempenho. Se fizéssemos a busca no back-end, também poderíamos incluir tuítes rudes em uma “lista negra”, o que seria mais difícil no front-end.

O Twitter, como o Facebook, permite criar *aplicativos*. Porém, essa não é uma designação correta: um aplicativo do Twitter não *faz* nada (no sentido tradicional). Ele é mais como um conjunto de credenciais que podemos usar para criar o aplicativo real no site. A maneira mais fácil e portátil de acessar a API do Twitter é criando um aplicativo e usando-o para obter tokens de acesso.

Crie um aplicativo Twitter acessando <http://dev.twitter.com>. Certifique-se de fazer login e clique em seu nome de usuário na barra de navegação e em

Apps. Clique em “Create an app” e siga as instruções. Quando você tiver um aplicativo, verá que agora possui uma *chave de API de consumidor* e uma *chave secreta de API*. A chave secreta de API, como o nome sugere, deve ser mantida secreta: nunca a inclua em respostas enviadas para o cliente. Se terceiros conseguirem acessá-la, poderão fazer requisições em nome da aplicação, o que pode produzir consequências inadequadas para você se o uso for malicioso.

Agora que temos uma chave de API de consumidor e a chave secreta, podemos nos comunicar com a API REST do Twitter.

Para manter nosso código limpo, inseriremos o código do Twitter em um módulo chamado *lib/twitter.js*:

```
const https = require('https')

module.exports = twitterOptions => {

  return {

    search: async (query, count) => {
      // A CONCLUIR
    }
  }

}
```

Esse padrão deve estar começando a ser familiar para você. Nosso módulo exporta uma função na qual o chamador passa um objeto de configuração. O que é retornado é um objeto contendo métodos. Dessa forma, podemos adicionar funcionalidade ao módulo. Atualmente, estamos fornecendo apenas um método `search`. Veja a seguir como usaremos a biblioteca:

```
const twitter = require('./lib/twitter')({
  consumerApiKey: credentials.twitter.consumerApiKey,
  apiSecretKey: credentials.twitter.apiSecretKey,
})

const tweets = await twitter.search('#Oregon #travel', 10)
// os tuítes estarão em result.statuses
```

(Não se esqueça de inserir uma propriedade twitter com consumerApiKey e apiSecretKey em seu arquivo *.credentials.development.json*).

Antes de implementar o método `search`, temos de fornecer a funcionalidade que nos autenticará no Twitter. O processo é simples: usaremos o HTTPS para requisitar um token de acesso com base na chave e no segredo de consumidor. Só precisamos fazer isso uma vez: atualmente o Twitter não encerra o prazo dos tokens de acesso (mas podemos invalidá-los manualmente). Já que não queremos ter de continuar requisitando um token de acesso, vamos armazená-lo em cache para poder reutilizá-lo.

A maneira como construímos nosso módulo nos permite criar funcionalidades privadas não disponíveis para o chamador. Especificamente, a única coisa que estará disponível será `module.exports`. Já que estamos retornando uma função, só ela estará disponível para o chamador. A chamada a essa função resulta em um objeto, e só as propriedades desse objeto estarão disponíveis para ele. Logo, criaremos uma variável `accessToken`, que usaremos para armazenar o token de acesso em cache, e uma função `getAccessToken` que obterá o token. Na primeira vez que for chamada, ela fará uma requisição de API do Twitter para obter o token de acesso. As chamadas subsequentes apenas retornarão o valor de `accessToken`:

```
const https = require('https')

module.exports = function(twitterOptions) {

  // essa variável será invisível fora desse módulo
  let accessToken = null

  // essa função será invisível fora desse módulo
  const getAccessToken = async () => {
    if(accessToken) return accessToken
    // A CONCLUIR: obtém o token de acesso
  }

  return {
    search: async (query, count) => {
      // A CONCLUIR
```

```

    }
  }

}

```

Marcamos `getAccessToken` como assíncrona porque podemos ter de fazer uma requisição para a API do Twitter (se não houver um token em cache). Agora que estabelecemos a estrutura básica, implementaremos `getAccessToken`:

```

const getAccessToken = async () => {
  if(accessToken) return accessToken

  const bearerToken = Buffer(
    encodeURIComponent(twitterOptions.consumerApiKey) + ':' +
    encodeURIComponent(twitterOptions.apiSecretKey)
  ).toString('base64')

  const options = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'POST',
    path: '/oauth2/token?grant_type=client_credentials',
    headers: {
      'Authorization': 'Basic ' + bearerToken,
    },
  }

  return new Promise((resolve, reject) => {
    https.request(options, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => {
        const auth = JSON.parse(data)
        if(auth.token_type !== 'bearer')
          return reject(new Error('Twitter auth failed.'))
        accessToken = auth.access_token
        return resolve(accessToken)
      })
    }).end()
  })
}

```



```
}
```

Os detalhes da construção dessa chamada estão disponíveis na *página da documentação de desenvolvedor para a autenticação somente de aplicação* (<http://bit.ly/2KcJ4EA>). Basicamente, temos de construir um token de portador (bearer token) que seja uma combinação da chave e do segredo de consumidor codificados na base64. Após construir esse token, poderemos chamar a API `/oauth2/token` com o cabeçalho `Authorization` contendo o token de portador para requisitar um token de acesso. É bom ressaltar que devemos usar o HTTPS: se você tentar fazer essa chamada pelo HTTP, transmitirá sua chave secreta não criptografada e a API simplesmente não responderá.

Quando recebermos a resposta completa da API (estamos escutando o evento `end` do stream da resposta), poderemos fazer o parsing do JSON e verificar se o token é de tipo `bearer`. Armazenaremos o token em cache e chamaremos o `callback`.

Como já temos um mecanismo para a obtenção de um token de acesso, podemos fazer chamadas de API. Implementaremos então nosso método `search`:

```
search: async (query, count) => {
  const accessToken = await getAccessToken()
  const options = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'GET',
    path: '/1.1/search/tweets.json?q=' +
      encodeURIComponent(query) +
      '&count=' + (count || 10),
    headers: {
      'Authorization': 'Bearer ' + accessToken,
    },
  }
  return new Promise((resolve, reject) => {
    https.request(options, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => resolve(JSON.parse(data)))
    }).end()
  })
}
```

```
)  
},
```

Renderizando tuítes

Estamos prontos para procurar tuítes... mas como os exibiremos no site? Basicamente, isso é uma decisão sua, mas há algumas coisas a serem consideradas. O Twitter tem a preocupação de assegurar que seus dados sejam usados de uma forma que seja consistente com a marca. Para fazê-lo, ele tem *requisitos de exibição* (<http://bit.ly/32ET4N2>), que empregam elementos funcionais que precisamos incluir para exibir um tuíte.

Os requisitos abrem algum espaço para exceções (por exemplo, se você estiver fazendo a exibição em um dispositivo que não dê suporte a imagens, não precisa incluir a imagem do avatar), mas quase sempre acabamos obtendo algo semelhante a um tuíte incorporado (embedded tweet). Dá muito trabalho, mas há uma maneira de resolver isso... porém envolve a conexão com a biblioteca de widgets do Twitter, que é a requisição HTTP que estamos tentando evitar.

Se você precisar exibir tuítes, a melhor opção é usar a biblioteca de widgets do Twitter, ainda que isso gere uma requisição HTTP adicional. Para um uso mais complicado da API, você também terá de acessar a API REST a partir do back-end, logo, provavelmente acabará usando a API REST junto com scripts de front-end.

Continuaremos com nosso exemplo: queremos exibir os 10 principais tuítes que mencionam as hashtags #Oregon #travel. Usaremos a API REST para procurar os tuítes e a biblioteca de widgets do Twitter para exibi-los. Já que não queremos exceder os limites de utilização (ou retardar nosso servidor), armazenaremos os tuítes e o HTML em cache para exibi-los por 15 minutos.

Começaremos modificando a biblioteca do Twitter para que inclua o método `embed`, que faz o HTML exibir um tuíte. Observe que estamos usando a biblioteca npm `querystringify` para construir uma `querystring` a partir de um objeto, portanto, não se esqueça de executar `npm install querystringify` e de importá-la (`const qs = require('querystringify ')`), e, em seguida, adicione a função a seguir à exportação de `lib/twitter.js`:

```

embed: async (url, options = {}) => {
  options.url = url
  const accessToken = await getAccessToken()
  const requestOptions = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'GET',
    path: '/1.1/statuses/oembed.json?' + qs.stringify(options),
    headers: {
      'Authorization': 'Bearer ' + accessToken,
    },
  }
  return new Promise((resolve, reject) => {
    https.request(requestOptions, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => resolve(JSON.parse(data)))
    }).end()
  })
},

```

Agora estamos prontos para procurar os tuítes e armazená-los em cache. No arquivo principal da aplicação, crie a função `getTopTweets` mostrada a seguir:

```

const twitterClient = createTwitterClient(credentials.twitter)

const getTopTweets = ((twitterClient, search) => {
  const topTweets = {
    count: 10,
    lastRefreshed: 0,
    refreshInterval: 15 * 60 * 1000,
    tweets: [],
  }
  return async () => {
    if(Date.now() > topTweets.lastRefreshed + topTweets.refreshInterval) {
      const tweets =
        await twitterClient.search('#Oregon #travel', topTweets.count)
      const formattedTweets = await Promise.all(
        tweets.statuses.map(async ({ id_str, user }) => {
          const url = `https://twitter.com/${user.id_str}/statuses/${id_str}`

```

```

    const embeddedTweet =
      await twitterClient.embed(url, { omit_script: 1 })
    return embeddedTweet.html
  })
)
topTweets.lastRefreshed = Date.now()
topTweets.tweets = formattedTweets
}
return topTweets.tweets
}
})(twitterClient, '#Oregon #travel')

```

O objetivo básico da função `getTopTweets` é não só procurar tuítes que tenham a hashtag especificada, mas também armazená-los em cache por algum período de tempo razoável. Repare que criamos uma expressão de função chamada imediatamente, ou IIFE (immediately invoked function expression): fizemos isso porque queremos que os `topTweets` sejam armazenados seguramente dentro de um closure no cache para que não possam ser manipulados. A função assíncrona que é retornada pela IIFE atualizará o cache se necessário e retornará seu conteúdo.

Para concluir, criaremos uma view, `views/social.handlebars`, como base para nossa presença na mídia social (que atualmente só inclui os tuítes selecionados):

```

<h2>Oregon Travel in Social Media</h2>

<script id="twitter-wjs" type="text/javascript"
  async defer src="//platform.twitter.com/widgets.js"></script>

{{{tweets}}}

```

E uma rota para manipulá-la:

```

app.get('/social', async (req, res) => {
  res.render('social', { tweets: await getTopTweets() })
})

```

Observe que referenciamos o script externo `widgets.js` do Twitter. Esse é o script que formatará e dará funcionalidade aos tuítes incorporados de sua página. Por padrão, a API `oembed` incluirá uma referência ao script no HTML,

mas, já que estamos exibindo 10 tuítes, ele será referenciado nove vezes além do necessário! Lembre-se de que, quando chamamos a API oembed, passamos a opção `{ omit_script: 1 }`. Já que o fizemos, tínhamos de fornecer o script em algum local, o que ocorreu na view. Vá em frente e tente remover o script da view. Você ainda verá os tuítes, mas eles não terão nenhuma formatação ou funcionalidade.

Agora temos um bom feed de mídia social! Voltaremos nossa atenção para outro recurso importante: exibir mapas na aplicação.

Geocodificação

Geocodificação é o processo de pegar um endereço ou o nome de um local (Bletchley Park, Sherwood Drive, Bletchley, Milton Keynes MK3 6EB, UK) e convertê-lo em coordenadas geográficas (latitude 51,9976597, longitude – 0,7406863). Se sua aplicação for executar algum tipo de cálculo geográfico – distâncias ou direções – ou exibir um mapa, você precisará de coordenadas geográficas.



Você deve estar acostumado a ver coordenadas geográficas especificadas em graus, minutos e segundos (DMS, degrees, minutes and seconds). As APIs de geocodificação e os serviços de mapeamento usam um único número de ponto flutuante para a latitude e a longitude. Se precisar exibir coordenadas DMS, consulte *esse arquivo da wikipedia* (<http://bit.ly/2Xc5IIM>).

Geocodificação com o Google

Tanto o Google quanto o Bing oferecem excelentes serviços REST para geocodificação. Usaremos o Google em nosso exemplo, mas o serviço do Bing é muito parecido.

Se você não anexar uma conta de cobrança à sua conta do Google, suas requisições de geocodificação ficarão limitadas a uma por dia, o que criará um ciclo de teste muito lento! Sempre que possível neste livro, tentaremos evitar recomendar serviços que você não possa usar pelo menos com capacidade de desenvolvimento gratuitamente, logo, testei alguns serviços de

geocodificação gratuitos, mas encontrei uma vantagem de usabilidade tão significativa na geocodificação do Google que continuo a recomendá-la. No entanto, quando este texto foi escrito, o custo de desenvolvimento-volume da geocodificação do Google era gratuito: recebíamos um crédito mensal de 200 dólares com nossa conta e tínhamos de fazer 40.000 requisições para exauri-lo! Se quiser acompanhar este capítulo, acesse *seu console do Google* (<http://bit.ly/2KcY1X0>), selecione Billing no menu principal e insira suas informações de cobrança.

Após definir a cobrança, você precisará de uma chave para a API de geocodificação do Google. Vá ao *console* (<http://bit.ly/2KcY1X0>), selecione seu projeto na barra de navegação e clique em APIs. Se a API de geocodificação não estiver em sua lista de APIs ativadas, localize-a na lista de APIs adicionais e inclua-a. A maioria das APIs do Google compartilha as mesmas credenciais, logo, clique no menu de navegação no canto superior esquerdo e volte ao dashboard. Clique em Credentials e crie uma nova chave de API se ainda não tiver uma apropriada. É bom ressaltar que as chaves de API podem apresentar restrições para impedir abusos, portanto, certifique-se de que sua chave possa ser usada a partir de sua aplicação. Se precisar de uma para desenvolvimento, você pode restringi-la por endereço IP e selecionar o seu endereço (se não souber qual é seu endereço, pergunte ao Google “Qual é meu endereço IP?”).

Quando tiver uma chave de API, adicione-a a *.credentials.development.json*:

```
"google": {  
  "apiKey": "<SUA CHAVE DE API>"  
}
```

Em seguida, crie um módulo *lib/geocode.js*:

```
const https = require('https')  
const { credentials } = require('../config')  
  
module.exports = async query => {  
  
  const options = {  
    hostname: 'maps.googleapis.com',  
    path: '/maps/api/geocode/json?address=' +
```

```

    encodeURIComponent(query) + '&key=' +
    credentials.google.apiKey,
  }

  return new Promise((resolve, reject) =>
    https.request(options, res => {
      let data = ""
      res.on('data', chunk => data += chunk)
      res.on('end', () => {
        data = JSON.parse(data)
        if(!data.results.length)
          return reject(new Error(`no results for "${query}"`))
        resolve(data.results[0].geometry.location)
      })
    }).end()
  )
}

```

Agora temos uma função que entrará em contato com a API do Google para executarmos a geocodificação de um endereço. Se ela não conseguir encontrar um endereço (ou falhar por alguma outra razão), um erro será retornado. A API pode retornar vários endereços. Por exemplo, se você procurar “10 Main Street” sem especificar uma cidade, estado ou código postal, ela retornará vários resultados. Nossa implementação selecionará o primeiro. A API retorna muitas informações, mas só estamos interessados nas coordenadas. Você pode modificar facilmente essa interface para que ela retorne mais informações. Consulte a *documentação da API de geocodificação do Google* (<http://bit.ly/2O4EE3t>) para obter mais detalhes sobre os dados que a API retorna.

Restrições de uso

Atualmente a API de geocodificação do Google tem um limite de uso mensal, sendo preciso pagar \$0.005 por requisição de geocodificação. Logo, se você fizer um milhão de requisições em um mês específico, receberá uma conta de 5.000 dólares do Google... então, provavelmente haverá um limite prático para o seu uso!



Se você está preocupado com cobranças fora de controle – que podem ocorrer se deixarmos acidentalmente um serviço sendo executado ou se terceiros maliciosos obtiverem acesso às credenciais –, pode adicionar um orçamento e configurar alertas para notificá-lo quando estiver se aproximando do limite. Vá ao console de desenvolvedor do Google e selecione “Budgets & alerts” no menu Billing.

Quando este texto foi escrito, o Google estava impondo um limite de 5.000 requisições por 100 segundos para evitar abusos, o que pode ser difícil de exceder. A API do Google também exige que, se usarmos um mapa em nosso site, utilizemos o Google Maps. Isto é, se você estiver usando o serviço do Google para geocodificar seus dados, não poderá exibir essas informações em um mapa do Bing sem violar os termos de serviço. Geralmente, essa não é uma restrição onerosa, já que você não executará a geocodificação a menos que queira exibir os locais em um mapa. No entanto, se prefere os mapas do Bing aos do Google, ou vice-versa, deve considerar os termos de serviço e usar a API apropriada.

Geocodificando seus dados

Temos um bom banco de dados de pacotes de férias na região do Oregon, e, se decidirmos exibir um mapa com pinos mostrando o destino dos diversos pacotes, é nesse momento que colocaremos em prática a geocodificação.

Já estamos com os dados dos pacotes de férias no banco de dados, e cada pacote tem uma string de busca de local que funcionará com a geocodificação, mas ainda não temos coordenadas.

A questão agora é quando e como executar a geocodificação. De um modo geral, temos três opções:

- Fazer a geocodificação quando adicionarmos novos pacotes de férias ao banco de dados. Essa é uma ótima opção para quando adicionarmos uma interface administrativa ao sistema que permita que os vendedores incluam pacotes de férias dinamicamente ao banco de dados. No entanto, já que não usaremos essa funcionalidade, descartaremos a opção.
- Fazer a geocodificação quando necessário ao recuperar pacotes de férias no banco de dados. Essa abordagem consiste em fazer uma verificação

sempre que acessarmos pacotes de férias no banco de dados: se algum não tiver coordenadas, executaremos a geocodificação. É uma opção que parece interessante, e talvez seja a mais fácil das três, mas apresenta algumas desvantagens que a tornam inviável. A primeira está relacionada ao desempenho: se você adicionar mil pacotes de férias novos ao banco de dados, a primeira pessoa que examinar a lista de pacotes terá de esperar que todas essas requisições de geocodificação sejam bem-sucedidas e gravadas. Além disso, imagine uma situação em que uma suíte de teste de carga adicionasse mil pacotes de férias ao banco de dados e executasse mil requisições. Já que todas são executadas concorrentemente, cada uma das mil requisições resultará em mil requisições de geocodificação porque os dados ainda não foram gravados no banco de dados... gerando um milhão de requisições de geocodificação e uma cobrança de 5.000 dólares do Google! Portanto, removeremos essa opção da lista.

- Usar um script que encontre pacotes de férias com dados de coordenadas ausentes e faça sua geocodificação. Essa abordagem oferece a melhor solução para nossa situação atual. Para fins de desenvolvimento, estamos preenchendo o banco de dados de pacotes de férias uma única vez, e ainda não temos uma interface administrativa para adicionar novos pacotes. Além disso, se decidirmos adicionar uma interface administrativa posteriormente, a abordagem não será incompatível: na verdade, poderíamos executar esse processo após adicionar um novo pacote, e ele funcionaria.

Primeiro, temos de adicionar uma maneira de atualizar um pacote de férias existente em *db.js* (também adicionaremos um método para fechar a conexão do banco de dados, o que será útil nos scripts):

```
module.exports = {  
  //...  
  updateVacationBySku: async (sku, data) => Vacation.updateOne({ sku }, data),  
  close: () => mongoose.connection.close(),  
}
```

Em seguida, podemos criar um script *db-geocode.js*:

```
const db = require('./db')
```

```

const geocode = require('./lib/geocode')

const geocodeVacations = async () => {
  const vacations = await db.getVacations()
  const vacationsWithoutCoordinates = vacations.filter(({ location }) =>
    !location.coordinates || typeof location.coordinates.lat !== 'number')
  console.log(`geocoding ${vacationsWithoutCoordinates.length} ` +
    `of ${vacations.length} vacations:`)
  return Promise.all(vacationsWithoutCoordinates.map(async ({ sku, location }) => {
    const { search } = location
    if(typeof search !== 'string' || !/^w/.test(search))
      return console.log(`SKU ${sku} FAILED: does not have location.search`)
    try {
      const coordinates = await geocode(search)
      await db.updateVacationBySku(sku, { location: { search, coordinates } })
      console.log(`SKU ${sku} SUCCEEDED: ${coordinates.lat}, ${coordinates.lng}`)
    } catch(err) {
      return console.log(`SKU {sku} FAILED: ${err.message}`)
    }
  })))
}

geocodeVacations()
  .then(() => {
    console.log('DONE')
    db.close()
  })
  .catch(err => {
    console.error('ERROR: ' + err.message)
    db.close() })

```

Quando você executar o script (node db-geocode.js), deve ver que todos os seus pacotes de férias foram geocodificados com sucesso! Agora que temos essa informação, aprenderemos como exibi-la em um mapa.

Exibindo um mapa

Embora exibir pacotes de férias em um mapa na verdade seja uma tarefa de “front-end”, seria muito frustrante chegarmos até aqui e não vermos os frutos

de nosso trabalho. Logo, vamos nos desviar um pouco do enfoque de back-end deste livro para ver como exibiremos nossos produtos recém-geocodificados em um mapa.

Já criamos uma chave de API do Google para fazer a geocodificação, mas ainda precisamos ativar a API de mapas. Vá até *seu console do Google* (<http://bit.ly/2KcY1X0>), clique em APIs, encontre Maps JavaScript API e ative a API se ela ainda não tiver sido ativada.

Agora podemos criar uma view para exibir nosso mapa de pacotes de férias, *views/vacations-map.handlebars*. Começaremos apenas exibindo o mapa e, em seguida, trabalharemos na inclusão de pacotes:

```
<div id="map" style="width: 100%; height: 60vh;"></div>
<script>
  let map = undefined
  async function initMap() {
    map = new google.maps.Map(document.getElementById('map'), {
      // centro geográfico aproximado do oregon
      center: { lat: 44.0978126, lng: -120.0963654 },
      // esse nível de zoom abrange grande parte do estado
      zoom: 7,
    })
  }
</script>
<script src="https://maps.googleapis.com/maps/api/js?key=
  {{googleApiKey}}&callback=initMap"
  async defer></script>
```

É hora de inserir alguns pinos no mapa correspondentes aos nossos pacotes de férias. No Capítulo 15, criamos um endpoint de API */api/vacations*, que agora incluirá dados geocodificados. Usaremos esse endpoint para acessar os pacotes e inseriremos pinos no mapa. Modificaremos a função *initMap* em *views/vacations-map.handlebars.js*:

```
async function initMap() {
  map = new google.maps.Map(document.getElementById('map'), {
    // centro geográfico aproximado do oregon
    center: { lat: 44.0978126, lng: -120.0963654 },
    // esse nível de zoom abrange grande parte do estado
    zoom: 7,
```

```

    })
    const vacations = await fetch('/api/vacations').then(res => res.json())
    vacations.forEach(({ name, location }) => {
      const marker = new google.maps.Marker({
        position: location.coordinates,
        map,
        title: name,
      })
    })
  })
}

```

Temos um mapa exibindo onde ficam todos os nossos pacotes de férias! Há muitas maneiras de melhorarmos essa página: provavelmente o melhor ponto de partida seja vincular os marcadores à página de detalhes dos pacotes, assim poderíamos clicar em um marcador e ele nos levaria à página de informações do pacote de férias. Também poderíamos implementar marcadores personalizados ou dicas de ferramenta: a API do Google Maps tem vários recursos e você pode conhecê-los na *documentação oficial do Google*

(<https://developers.google.com/maps/documentation/javascript/tutorial>).

Dados climáticos

Lembra-se de nosso widget de “condições meteorológicas atuais” do Capítulo 7? Vamos adicionar a ele alguns dados dinâmicos! Usaremos a API do Serviço Nacional de Meteorologia dos Estados Unidos (NWS, US National Weather Service) para obter informações de previsão do tempo. Como na integração com o Twitter, e o uso da geocodificação, armazenaremos a previsão em cache para evitar passar cada acesso ao nosso site para o NWS (o que pode nos fazer ser incluídos na lista negra se o site se popularizar). Crie um arquivo chamado *lib/weather.js*:

```

const https = require('https')
const { URL } = require('url')

const _fetch = url => new Promise((resolve, reject) => {
  const { hostname, pathname, search } = new URL(url)
  const options = {

```

```

hostname,
path: pathname + search,
headers: {
  'User-Agent': 'Meadowlark Travel'
},
}
https.get(options, res => {
  let data = ''
  res.on('data', chunk => data += chunk)
  res.on('end', () => resolve(JSON.parse(data)))
}).end()
})

```

```

module.exports = locations => {

```

```

  const cache = {
    refreshFrequency: 15 * 60 * 1000,
    lastRefreshed: 0,
    refreshing: false,
    forecasts: locations.map(location => ({ location })),
  }

```

```

  const updateForecast = async forecast => {
    if(!forecast.url) {
      const { lat, lng } = forecast.location.coordinates
      const path = `/points/${lat.toFixed(4)}/${lng.toFixed(4)}`
      const points = await _fetch('https://api.weather.gov' + path)
      forecast.url = points.properties.forecast
    }
    const { properties: { periods } } = await _fetch(forecast.url)
    const currentPeriod = periods[0]
    Object.assign(forecast, {
      imageUrl: currentPeriod.icon,
      weather: currentPeriod.shortForecast,
      temp: currentPeriod.temperature + ' ' + currentPeriod.temperatureUnit,
    })
    return forecast
  }

```

```

const getForecasts = async () => {
  if(Date.now() > cache.lastRefreshed + cache.refreshFrequency) {
    console.log('updating cache')
    cache.refreshing = true
    cache.forecasts = await Promise.all(cache.forecasts.map(updateForecast))
    cache.refreshing = false
  }
  return cache.forecasts
}

return getForecasts
}

```

Você notará que nos cansamos de usar a biblioteca interna `https` do Node diretamente e em vez disso criamos uma função utilitária `_fetch` para tornar nossa funcionalidade meteorológica um pouco mais legível. Uma coisa que pode chamar sua atenção é que estamos configurando o cabeçalho `User-Agent` com `Meadowlark Travel`. Essa é uma peculiaridade da API meteorológica do NWS: ela requer uma string para `User-Agent`. Sua equipe disse que essa exigência será substituída por uma chave de API, mas por enquanto precisamos fornecer um valor aqui.

Nesse caso, obter dados meteorológicos na API do NWS é uma tarefa de duas etapas. Há um endpoint de API chamado `points` que recebe a latitude e a longitude (com exatamente quatro dígitos decimais) e retorna informações sobre esse local, inclusive a URL apropriada a partir da qual podemos obter uma previsão. Quando tivermos essa URL para algum conjunto de coordenadas especificado, não precisaremos buscá-lo novamente. Só temos de chamar a URL para obter a previsão atualizada.

Observe que são retornados mais dados pela previsão do que os que estamos usando; poderíamos ser mais sofisticados ao usar esse recurso. Especificamente, a URL da previsão retorna um array de períodos, com o primeiro elemento sendo o período atual (por exemplo, “tarde” ou “noite”). Depois vêm períodos que se estendem até a semana seguinte. Fique à vontade para examinar os dados do array `periods` e ver os tipos de informações que estão disponíveis.

Um detalhe que devemos observar é que temos uma propriedade booleana em nosso cache chamada `refreshing`. Isso é necessário porque a atualização do cache demora um período de tempo finito e é feita assincronamente. Se várias requisições chegarem antes de a primeira atualização do cache ser concluída, todas elas iniciarão o trabalho de atualização do cache. Não causará danos, mas você fará mais chamadas de API do que seria necessário. Essa variável booleana é apenas um aviso para qualquer requisição adicional que diz “Estamos trabalhando nisso”.

Projetamos essa função como substituta, se necessário, para a função fictícia que criamos no Capítulo 7. Só precisamos abrir *lib/middleware/weather.js* e substituir a função `getWeatherData`:

```
const weatherData = require('../weather')

const getWeatherData = weatherData([
  {
    name: 'Portland',
    coordinates: { lat: 45.5154586, lng: -122.6793461 },
  },
  {
    name: 'Bend',
    coordinates: { lat: 44.0581728, lng: -121.3153096 },
  },
  {
    name: 'Manzanita',
    coordinates: { lat: 45.7184398, lng: -123.9351354 },
  },
])
```

Agora temos dados meteorológicos dinâmicos em nosso widget!

Conclusão

Só examinamos superficialmente o que é possível fazer quando executamos a integração com APIs de terceiros. Para onde quer que olhemos, vemos novas APIs surgindo, oferecendo todos os tipos de dados imagináveis (até mesmo a cidade de Portland já está tornando vários dados públicos disponíveis por meio de APIs REST). Seria impossível abranger mesmo que fosse uma

pequena porcentagem das APIs que estão disponíveis, mas este capítulo abordou os elementos básicos que precisamos conhecer para usá-las: `http.request`, `https.request` e o parsing de JSON.

Já adquirimos muito conhecimento. Cobrimos bastante terreno! No entanto, o que acontece quando as coisas dão errado? No próximo capítulo, discutiremos técnicas de depuração para nos ajudar quando as coisas não saírem como esperado.

CAPÍTULO 20

Depuração

Talvez “depuração” seja um termo inadequado, porque é associado apenas a falhas. Na verdade, o que chamamos de “depuração” é uma atividade de execução contínua, independentemente de estarmos implementando um novo recurso, aprendendo como algo funciona ou realmente corrigindo um bug. Um termo melhor seria “exploração”, mas ficaremos com “depuração”, já que a atividade a qual ele se refere é clara, não importando a motivação.

A depuração é uma habilidade frequentemente negligenciada: parece que as pessoas acham que os programadores já nasceram sabendo como executá-la. Talvez os professores de ciência da computação e os autores de livros vejam a depuração como uma habilidade óbvia que não precise de atenção.

A verdade é que a depuração é uma habilidade que pode ser ensinada, e é uma maneira importante de os programadores entenderem não só o framework em que estão trabalhando, mas também seu próprio código e o de sua equipe. Neste capítulo, discutiremos algumas das ferramentas e técnicas que você pode usar para depurar eficientemente aplicações Node e Express.

Primeiro princípio da depuração

Como o nome sugere, geralmente “depuração” é o processo de encontrar e eliminar defeitos. Antes de falar sobre ferramentas, consideraremos alguns princípios gerais da depuração.

Quantas vezes já lhe falei que, quando você elimina o impossível, o que quer que permaneça, embora improvável, deve ser a verdade?

SIR ARTHUR CONAN DOYLE

O primeiro e mais importante princípio da depuração é o processo de *eliminação*. Os sistemas de computador modernos são incrivelmente complicados, e, se você tivesse de guardar o *sistema inteiro* em sua mente, e remover a fonte de um único problema desse vasto espaço, provavelmente não saberia nem mesmo onde começar. Sempre que deparar com um problema que não seja imediatamente óbvio, seu *primeiro pensamento* deve ser “O que posso *eliminar* como fonte do problema?”. Quanto maior o número de coisas que puder eliminar, menos locais precisará examinar.

A eliminação pode ser feita de várias formas. Aqui estão alguns exemplos comuns:

- Desabilitar como comentários ou desativar sistematicamente blocos de código.
- Criar código que seja alvo de testes unitários; os testes unitários fornecem um framework para a eliminação.
- Analisar o tráfego de rede para determinar se o problema é no lado do cliente ou do servidor.
- Testar uma parte diferente do sistema que seja parecida com a primeira.
- Usar entradas que tenham funcionado antes e alterá-las uma de cada vez até o problema surgir.
- Usar o controle de versões para voltar e avançar no tempo até o problema desaparecer e você poder associá-lo a uma alteração específica (consulte *git bisect* (<http://bit.ly/34TOufp>) para obter mais informações sobre isso).
- “Simular” funcionalidades para eliminar subsistemas complexos.

No entanto, a eliminação não é uma bala de prata. Com frequência, os problemas ocorrem devido a interações complexas entre dois ou mais componentes: elimine (ou simule) um deles e o problema pode desaparecer, mas não podemos associá-lo a um único componente. Porém, mesmo nessa situação, a eliminação pode ajudar a restringir o problema, ainda que não indique o local exato.

A eliminação é mais bem-sucedida quando é cuidadosa e metódica. É muito fácil deixar passar algo quando eliminamos componentes arbitrariamente sem considerar como eles afetam o todo. Pense como se fosse um jogo: quando considerar um componente para eliminar, pondere como sua remoção afetará o sistema. Isso lhe mostrará o que esperar e se a remoção ou não dos componentes lhe revelará algo útil.

Beneficie-se do REPL e do console

Tanto o Node quanto os navegadores oferecem um *read-eval-print loop* (REPL), que é apenas uma maneira de escrever JavaScript interativamente. Você digita o JavaScript, pressiona Enter e imediatamente vê a saída. É uma ótima maneira de fazer testes, e com frequência é o modo mais fácil e intuitivo de localizar um erro em pequenos fragmentos de código.

Em um navegador, só precisamos acessar o console JavaScript e teremos um REPL. No Node, basta digitar `node` sem nenhum argumento e entramos no modo REPL; você pode demandar pacotes, criar variáveis e funções ou fazer qualquer coisa que faria normalmente em seu código (exceto criar pacotes: não há uma maneira relevante de fazer isso no REPL).

O logging do console também é útil. Pode ser uma técnica de depuração bruta, mas também é fácil (fácil de entender e de implementar). Chamar `console.log` no Node exibe o conteúdo de um objeto em um formato fácil de ler, logo, podemos detectar problemas rapidamente. Lembre-se de que alguns objetos são tão grandes que os registrar no console produzirá tanta saída que você terá dificuldades para encontrar qualquer informação útil. Como exemplo, execute `console.log(req)` em um de seus manipuladores de path.

Usando o depurador interno do Node

O Node tem um depurador interno que permite percorrer a aplicação, como se estivéssemos em uma exploração com o interpretador JavaScript. Tudo que precisamos fazer para começar a depuração do aplicativo é usar o argumento `inspect`:

```
node inspect meadowlark.js
```

Quando você fizer isso, notará imediatamente algumas coisas. Em primeiro lugar, verá uma URL no console; isso ocorre porque o depurador do Node funciona criando o próprio servidor web, o que nos permite controlar a execução da aplicação que está sendo depurada. No momento, talvez isso não pareça grande coisa, mas a utilidade dessa abordagem ficará clara quando discutirmos os clientes do inspetor.

Quando você estiver no depurador do console, poderá digitar `help` para obter uma lista de comandos. Os comandos que usará com mais frequência são `n` (`next`), `s` (`step in`) e `o` (`step out`). `n` “pula” a linha atual: ele a executará, mas, se essa instrução chamar outras funções, elas serão executadas antes de o controle ser retornado para você. `s`, ao contrário, *entra* na linha atual: se essa linha chamar outras funções, você poderá percorrê-las. `o` permite sair da função que está sendo executada. (Observe que as ações de “entrar” e “sair” só são aplicáveis a *funções*; elas não entram ou saem de blocos `if` ou `for` ou outras instruções de controle de fluxo).

O depurador de linha de comando tem mais funcionalidades, mas é provável que você não as use com muita frequência. A linha de comando é ótima para várias coisas, mas a depuração não é uma delas. É bom que esteja disponível em uma emergência (por exemplo, se você tiver apenas acesso SSH ao servidor ou se seu servidor não tiver nem mesmo uma GUI instalada). No entanto, é preferível usar um cliente de inspetor gráfico.

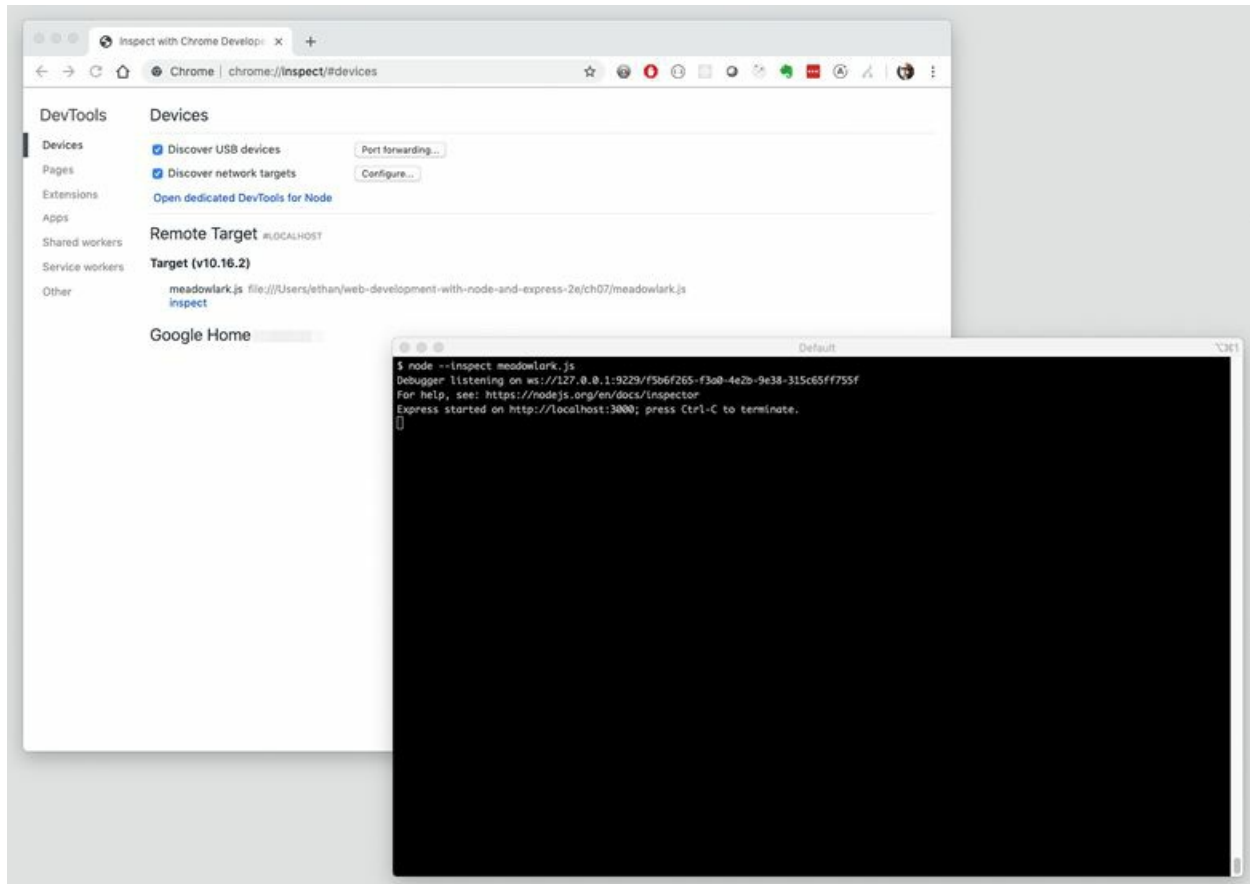
Clientes do inspetor do Node

Como você provavelmente não vai querer usar o depurador de linha de comando exceto em uma emergência, o fato de o Node expor seus controles de depuração por um web service fornece outras opções.

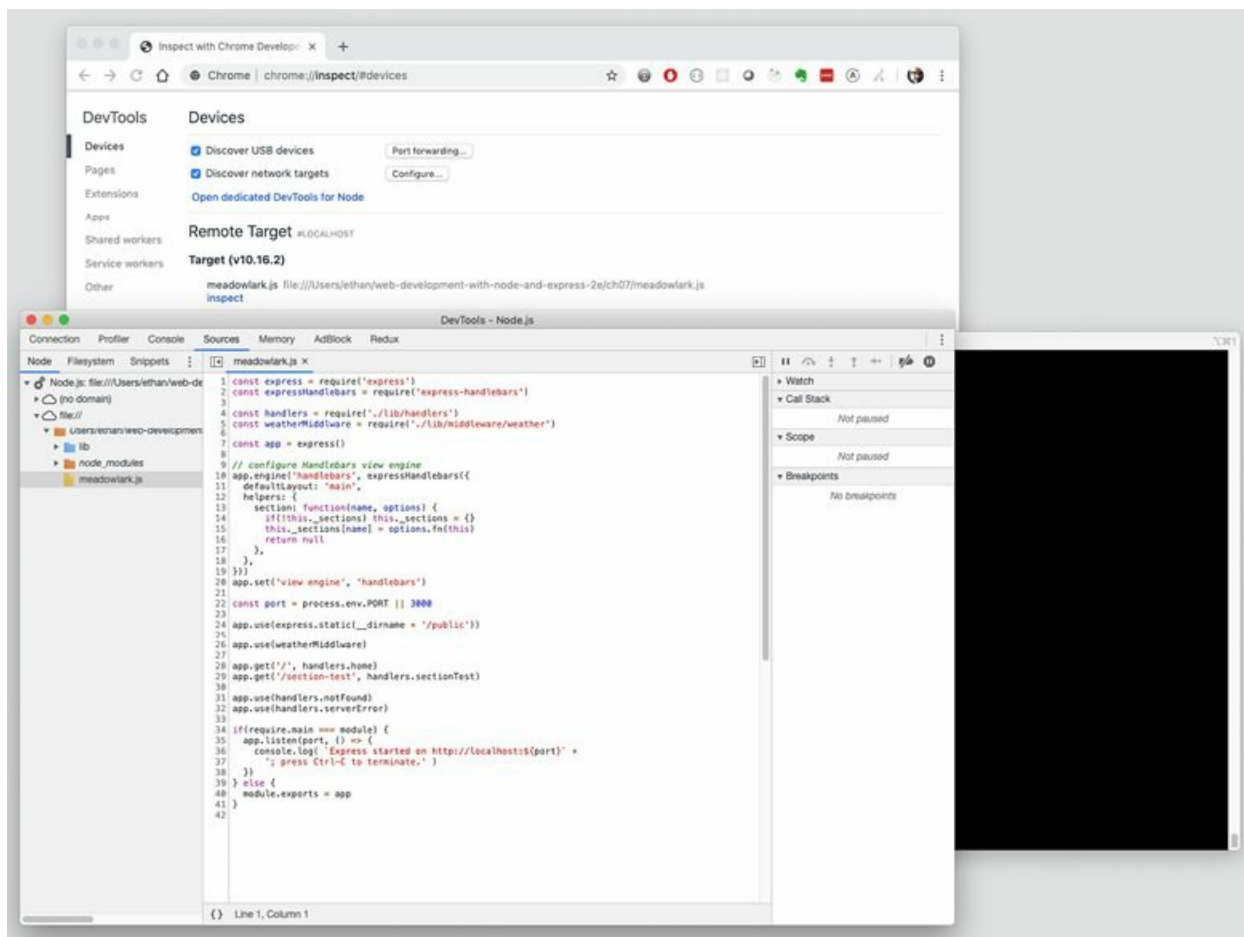
A maneira mais simples de depurar é usando o Chrome, que emprega a mesma interface de depuração utilizada para a depuração de código de front-end. Logo, se você já usou essa interface, vai se sentir em casa. É fácil começar. Inicie sua aplicação com a opção `--inspect` (que é diferente do argumento `inspect` mencionado anteriormente):

```
node --inspect meadowlark.js
```

Agora começa a diversão: na barra de URL de seu navegador, insira `chrome://inspect`. Você verá uma página DevTools; na seção Devices, clique em “Open dedicated DevTools for Node”. Isso abrirá uma nova janela, que é o depurador:



Clique na aba Sources; no painel da esquerda, clique na aba Node.js para expandi-la e em “file:///”. Você verá a pasta na qual está sua aplicação; expanda-a e verá todo o código-fonte JavaScript (são exibidos apenas o JavaScript e às vezes os arquivos JSON quando eles são requeridos em outro local). Aqui, você pode clicar em qualquer arquivo para ver o código-fonte e definir pontos de interrupção (breakpoints):



Ao contrário do que vimos em nossa experiência anterior com o depurador de linha de comando, a aplicação já estará sendo executada: todo o middleware terá sido incluído e o aplicativo estará escutando. Porém, como percorrer o código? A maneira mais fácil (e o método que provavelmente você usará com mais frequência) é definir um *ponto de interrupção*. Isso informará ao depurador que pare a execução em uma linha específica para você poder percorrer o código.

Para definir um ponto de interrupção, basta abrir um arquivo de código-fonte no navegador de “file://” no depurador e clicar no número da linha (na coluna esquerda); uma pequena seta azul aparecerá, indicando que há um ponto de interrupção nessa linha (clique novamente para desativá-lo). Vá em frente e defina um ponto de interrupção dentro de um de seus manipuladores de rota. Em seguida, em outra janela do navegador, visite essa rota. Se estiver usando o Chrome, o navegador mudará automaticamente para a janela do depurador, enquanto o navegador original apenas aguardará (porque o servidor foi

pausado e não está respondendo à requisição).

Na janela do depurador, você poderá percorrer o programa de maneira muito mais visual do que fizemos com o depurador de linha de comando. Você verá que a linha na qual definiu um ponto de interrupção está realçada em azul. Isso significa que ela é a linha atual da execução (que na verdade é a próxima linha que será executada). Aqui, você terá acesso aos mesmos comandos que acessamos no depurador de linha de comando. Como no depurador de linha de comando, temos as seguintes ações disponíveis:

Retomar a execução do script (F8)

Significa simplesmente “deixar prosseguir”; você não estará mais percorrendo o código, a menos que pare em outro ponto de interrupção. Geralmente esse recurso é usado quando já vimos o que precisávamos ou se quisermos passar para outro ponto de interrupção.

Passar para a próxima chamada de função (F10)

Se a linha chamar uma função, o depurador não entrará nela. Isto é, a função será executada e o depurador avançará para a próxima linha posterior à chamada da função. Você usará esse recurso quando estiver em uma chamada de função cujos detalhes não lhe interessarem.

Entrar na próxima chamada de função (F11)

Essa ação entra na chamada de função, não ocultando nada. Se essa fosse a única ação que você pudesse usar, mesmo assim veria tudo que está sendo executado – o que inicialmente pode parecer divertido, mas, após passar uma hora, desenvolvemos um respeito renovado pelo que o Node e o Express fazem para nos ajudar!

Sair da função atual (Shift-F11)

Executará o resto da função na qual você está atualmente e retomar a depuração na próxima linha do seu *chamador*. Geralmente, usamos esse recurso quando entramos acidentalmente em uma função ou já vimos o suficiente.

Além de todas as ações de controle, você terá acesso a um console: esse console estará sendo executado no *contexto atual de sua aplicação*. Logo, você poderá inspecionar variáveis, e até mesmo alterá-las, ou chamar funções. Isso pode ser bastante útil para o teste de coisas muito simples, mas pode tornar-se confuso rapidamente, portanto, não recomendo que você faça muitas modificações dinamicamente na aplicação em execução; é fácil se perder.

À direita temos alguns dados úteis. A partir do topo vemos as *watch expressions* (expressões de observação); são expressões JavaScript que podemos definir e que serão atualizadas em tempo real à medida que percorrermos a aplicação. Por exemplo, se houvesse uma variável específica que quiséssemos rastrear, poderíamos inseri-la aqui.

Abaixo das watch expressions está a *pilha de chamadas* (call stack); ela exhibe como chegamos onde estamos. Isto é, a função na qual você está foi chamada por alguma outra função, e essa função também foi chamada por uma função; a pilha de funções lista todas essas funções. No universo altamente assíncrono do Node, pode ser muito difícil decifrar e entender a pilha de chamadas, principalmente quando funções anônimas estiverem envolvidas. A primeira entrada dessa lista é onde você está agora. A que vem logo abaixo é a função que chamou a função na qual você está e assim por diante. Se clicar em uma entrada da lista, será transportado para esse contexto: todas as suas observações e o contexto do console passarão para o contexto em questão.

Abaixo da pilha de chamadas temos as variáveis de escopo. Como o nome sugere, são as variáveis que fazem parte do escopo atual (o que inclui as variáveis do escopo pai que estiverem visíveis para nós). Essa seção pode fornecer imediatamente muitas informações sobre as principais variáveis que nos interessarem. Se você tiver muitas variáveis, essa lista será complexa e pode ser melhor definir as variáveis de interesse como watch expressions.

Em seguida, há uma lista de todos os pontos de interrupção, que na verdade é apenas uma relação: será útil usá-la se você tiver um problema complicado e houver muitos pontos de interrupção. Clicar em um o levará diretamente para esse local (mas não mudará o contexto, como quando clicamos em algo na pilha de chamadas; isso faz sentido porque nem todo ponto de interrupção

representa um contexto ativo, ao contrário do que ocorre na pilha de chamadas).

Às vezes, o que precisa de depuração é a configuração da aplicação (quando incluímos middleware no Express, por exemplo). Se executarmos o depurador como temos feito, a depuração acontecerá rapidamente antes mesmo de podermos definir um ponto de interrupção. Felizmente, há uma maneira de resolver esse problema. Basta especificar `--inspect-brk` em vez de simplesmente `--inspect`:

```
node --inspect-brk meadowlark.js
```

O depurador parará na primeira linha de sua aplicação e você poderá percorrer ou definir os pontos de Interrupção como quiser.

O Chrome não é a única opção de cliente do inspetor. Especificamente, o depurador interno do Visual Studio Code funciona muito bem. Em vez de iniciar a aplicação com as opções `--inspect` ou `--inspect-brk`, clique no ícone Debug no menu lateral do Visual Studio Code (um inseto com uma linha cortando-o). No topo da barra lateral, você verá um pequeno ícone de engrenagem; clique nele e isso abrirá algumas definições de configuração de depuração. A única configuração com a qual você precisa se preocupar é “program”; certifique-se de que ela esteja direcionada para seu ponto de entrada (*meadowlark.js*, por exemplo).



Você também pode ter de definir que está usando o diretório de trabalho atual, ou “`cwd`”. Por exemplo, se tiver aberto o Visual Studio Code em um diretório pai de onde *meadowlark.js* reside, pode ter de passar para “`cwd`” (que é o mesmo que se tivéssemos de mudar com `cd` para o diretório certo antes de executar `node meadowlark.js`).

Quando estiver com tudo preparado, simplesmente clique na seta verde de Play na barra de depuração e seu depurador será executado. A interface é um pouco diferente da do Chrome, mas, se você usar o Visual Studio Code, provavelmente se sentirá em casa. Para obter mais informações, consulte *Debugging in Visual Studio Code* (<http://bit.ly/2pb7JBV>).

Depurando funções assíncronas

Uma das frustrações mais comuns para as pessoas quando elas são expostas à programação assíncrona pela primeira vez é a depuração. Considere o código a seguir, por exemplo:

```
1 console.log('Baa, baa, black sheep,');
2 fs.readFile('yes_sir_yes_sir.txt', (err, data) => {
3   console.log('Have you any wool?');
4   console.log(data);
5 })
6 console.log('Three bags full.')
```

Se você for iniciante na programação assíncrona, vai esperar ver algo assim:

```
Baa, baa, black sheep,
Have you any wool?
Yes, sir, yes, sir,
Three bags full.
```

No entanto, não é isso que verá, e sim o descrito a seguir:

```
Baa, baa, black sheep,
Three bags full.
Have you any wool?
Yes, sir, yes, sir,
```

Se ficou confuso, a depuração não deve ajudá-lo. Você começará na linha 1, e a pulará, o que o levará para a linha 2. Você então entrará nessa linha, esperando entrar na função e passar para a linha 3, mas na verdade acabará passando para a linha 5! Isso ocorre porque `fs.readFile` só executa a função *quando termina de ler o arquivo*, o que não acontecerá até a aplicação estar ociosa. Logo, você pulará a linha 5 e passará para a linha 6... e depois tentará continuar percorrendo o código, mas nunca chegará à linha 3 (acabará chegando, mas pode demorar um pouco).

Se você quiser depurar as linhas 3 ou 4, basta definir um ponto de interrupção na linha 3, e deixar o depurador ser executado. Quando o arquivo for lido e a função for chamada, você parará nessa linha, e tudo deve dar certo.

Depurando o Express

Se, como eu, você viu muitos frameworks com excesso de recursos em sua carreira, a ideia de percorrer o código-fonte de um framework pode parecer

loucura (ou tortura). E examinar o código-fonte do Express não é fácil, mas *está* ao alcance de qualquer pessoa que conheça bem JavaScript e o Node. Às vezes, quando temos problemas com o código, a melhor maneira de resolvê-los pode ser percorrer o código-fonte do próprio Express (ou de middleware de terceiros).

Esta seção será um breve passeio pelo código-fonte do Express para que você seja mais eficaz na depuração de suas aplicações Express. Em cada parte da jornada, fornecerei o nome do arquivo em relação à raiz do Express (que você encontrará em seu diretório *node_modules/express*), e o nome da função. Não estou usando números de linha, porque é claro que eles podem diferir dependendo da versão do Express que você estiver usando:

Criação do aplicativo Express (lib/express.js, function createApplication)

É aqui que seu aplicativo Express passará a existir. Essa é a função invocada quando chamamos `const app = express()` em nosso código.

Inicialização do aplicativo Express (lib/application.js, app.defaultConfiguration)

É aqui que o Express é inicializado: é um bom local para vermos todos os padrões que o Express usa inicialmente. Raramente é necessário definir um ponto de interrupção aqui, mas é útil percorrer essa parte pelo menos uma vez para conhecer as configurações padrão do Express.

Inclusão de middleware (lib/application.js, app.use)

Sempre que o Express faz a inclusão de middleware (independentemente de ter sido você que a fez explicitamente ou de ter sido feita implicitamente pelo Express ou por terceiros), essa função é chamada. Ela parece simples, mas na verdade é preciso um pouco de trabalho para entendê-la. Às vezes é útil inserir um ponto de interrupção aqui (você vai querer usar `--debug-brk` quando executar seu aplicativo; caso contrário, todo middleware será adicionado antes que o ponto de interrupção seja definido), mas pode ser complicado: é surpreendente o volume de middleware incluído em uma aplicação típica.

Renderização de views (lib/application.js, app.render)

Essa é outra função muito complexa, mas é útil quando precisamos depurar problemas difíceis relacionados às views. Se você a percorrer, verá como o view engine é selecionado e chamado.

Extensões de requisição (lib/request.js)

Você ficará surpreso com como é fácil entender esse arquivo. A maioria dos métodos que o Express adiciona aos objetos de requisição é composta de funções de conveniência muito simples. Raramente é necessário percorrer esse código ou definir pontos de interrupção devido à sua simplicidade. No entanto, com frequência é útil examiná-lo para saber como alguns dos métodos de conveniência do Express funcionam.

Envio de respostas (lib/response.js, res.send)

Não importa como construirmos a resposta – usando `.send`, `.render`, `.json` ou `.jsonp` – acabaremos chegando a essa função (a exceção é `.sendFile`). Logo, esse é um local útil para a definição de um ponto de interrupção, porque ele deve ser chamado para todas as respostas. Você poderá então usar a pilha de chamadas para ver como chegou aqui, o que será útil na descoberta de onde pode haver um problema.

Extensões de resposta (lib/response.js)

Embora haja alguma complexidade em `res.send`, a maioria dos outros métodos do objeto de resposta é muito simples. Ocasionalmente é útil inserir pontos de interrupção nessas funções para vermos como o aplicativo está respondendo às requisições.

Middleware static (node_modules/serve-static/index.js, function staticMiddleware)

Geralmente, quando os arquivos estáticos não estão sendo servidos como esperado, o problema é no roteamento, e não no middleware static: o roteamento tem precedência sobre o middleware static. Portanto, se você tiver um arquivo `public/test.jpg` e uma rota `/test.jpg`, o middleware static nunca será chamado porque a rota terá preferência. Mas, se você precisar saber particularidades sobre como os cabeçalhos são definidos

diferentemente para os arquivos estáticos, pode ser útil percorrer o `middleware static`.

Se você está coçando a cabeça se perguntando onde está todo esse `middleware`, na verdade há muito pouco `middleware` no Express (o `middleware static` e o roteador são as únicas exceções importantes).

Assim como é útil sondar o código-fonte do Express quando você estiver tentando resolver um problema difícil, pode ser preciso examinar o código-fonte do `middleware`. Há muito código para percorrermos, mas quero destacar três como mais importantes para o conhecimento do que está ocorrendo em uma aplicação Express:

Middleware de sessão (`node_modules/express-session/index.js`, function `session`)

Várias coisas fazem uma sessão funcionar, mas o código é muito simples. Pode ser interessante definir um ponto de interrupção nessa função se você estiver tendo problemas relacionados às sessões. Lembre-se de que é responsabilidade sua fornecer o engine de armazenamento para o `middleware de sessão`.

Middleware logger (`node_modules/morgan/index.js`, function `logger`)

O `middleware logger` existe como ajuda à depuração, e não para ser depurado. No entanto, há alguma sutileza na maneira como o logging funciona que você só verá percorrendo o `middleware logger` uma ou duas vezes. Na primeira vez que o fiz, me surpreendi em vários momentos, e comecei a usar o logging com mais eficiência em minhas aplicações, logo, recomendo percorrer esse `middleware` pelo menos uma vez.

Parsing do corpo codificado em URL (`node_modules/body-parser/lib/types/urlencoded.js`, function `urlencoded`)

A maneira como ocorre o parsing dos corpos de requisições costuma ser um mistério para muitas pessoas. Na verdade, não é tão complicado, e percorrer esse `middleware` o ajudará a entender como as requisições HTTP funcionam. Além de ser uma experiência de aprendizagem, você descobrirá que com frequência não é preciso examinar esse `middleware` na depuração.

Conclusão

Falamos *muito* sobre middleware neste livro. Não tenho como listar todos os pontos de referência que você poderia examinar em sua inspeção das partes internas do Express, mas espero que o que destaquei aqui revele alguns dos mistérios desse framework e o encoraje a explorar seu código-fonte quando necessário. O middleware varia muito não só em qualidade, mas também em acessibilidade: alguns são realmente difíceis de entender, enquanto outros são cristalinos como água. Seja qual for o caso, não tenha medo de examinar: se for muito complicado, deixe de lado (a não ser, claro, que precise entendê-lo), e, se não for, você pode aprender algo.

CAPÍTULO 21

Distribuição online

Chegou o grande dia: você passou semanas ou meses elaborando um trabalho esmerado e agora seu site ou serviço está pronto para lançamento. Não será tão fácil como “apertar um botão” e o site estará online... ou será?

Neste capítulo (que você deve ler *semanas* antes do lançamento, e não no dia exato!), conheceremos alguns dos serviços de hospedagem e registro de domínio disponíveis, técnicas para a passagem de um ambiente de teste para o de produção, e o que devemos considerar na seleção de serviços de produção.

Hospedagem e registro de domínio

As pessoas costumam se confundir em relação à diferença entre *registro de domínio* e *hospedagem*. Se você está lendo este livro, provavelmente não está confuso, mas aposto que conhece pessoas que se confundem, como seus clientes ou seu gerente.

Todos os sites e serviços da internet podem ser identificados por um (ou mais de um) *endereço de Protocolo de Internet (IP, Internet Protocol)*. Esses números não são particularmente amigáveis para os humanos (e essa situação vai piorar quando a adoção do IPv6 aumentar), mas o computador precisa deles para exibir uma página web. É nessa hora que os *nomes de domínio* entram em cena. Eles mapeiam um nome que os humanos entendem (como *google.com*) para um endereço IP (74.125.239.13 ou 2601:1c2:1902:5b38:c256:27ff:fe70:47d1).

Uma analogia com o mundo real seria a diferença entre o nome de uma empresa e seu endereço físico. Um nome de domínio é como o nome de uma empresa (Apple), e o endereço IP seria o endereço físico (One Apple Park

Way, Cupertino, CA 95014). Se você precisasse visitar a sede da Apple, teria de saber o endereço físico. Felizmente, se souber o nome da empresa, deve conseguir o endereço. A outra razão que torna essa abstração útil é que uma empresa pode se mudar (tendo um novo endereço físico) e as pessoas ainda poderão encontrá-la (na verdade, a Apple transferiu sua sede física entre a primeira e a segunda edição deste livro).

A *hospedagem*, por outro lado, descreve os computadores que estão executando o site. Continuando com a analogia física, poderíamos comparar a hospedagem com os prédios que vemos quando chegamos ao endereço físico. O que costuma confundir as pessoas é o registro de domínio ter muito pouco a ver com a hospedagem, e nem sempre compramos o domínio na mesma entidade que pagamos pela hospedagem (da mesma forma que podemos comprar um terreno de uma pessoa e pagar outra para construir e fazer a manutenção de prédios).

Embora seja possível hospedar um site sem um nome de domínio, isso não é muito amigável: os endereços IP não são apropriados em termos de marketing! Geralmente, quando compramos hospedagem, recebemos automaticamente um subdomínio (que abordaremos em breve), que pode ser considerado como algo entre um nome de domínio apropriado ao marketing e um endereço IP (por exemplo, *ec2-54-201-235-192.us-west-2.compute.amazonaws.com*).

Após termos um domínio, e entrarmos online, o site poderá ser alcançado com várias URLs. Por exemplo:

- <http://meadowlarktravel.com/>
- <http://www.meadowlarktravel.com/>
- <http://ec2-54-201-235-192.us-west-2.compute.amazonaws.com/>
- <http://54.201.235.192/>

Graças ao mapeamento de domínio, todos esses endereços apontam para o mesmo site. Quando as requisições chegarem ao site, será possível executar ações com base na URL que foi usada. Por exemplo, se alguém acessar seu site a partir do endereço IP, você poderá fazer o redirecionamento automaticamente para o nome de domínio, embora isso não seja muito

comum já que não faz tanto sentido (é mais comum fazer o redirecionamento de <http://meadowlarktravel.com/> para <http://www.meadowlarktravel.com/>).

A maioria dos registradores de domínio oferece serviços de hospedagem (ou tem parcerias com empresas que o fazem). Exceto pela AWS, nunca achei as opções de hospedagem dos registradores particularmente interessantes, e não há problema em separar o registro do domínio e a hospedagem.

Sistema de nome de domínio

O *Sistema de Nome de Domínio* (DNS, Domain Name System) é o responsável por mapear nomes de domínio para endereços IP. É um sistema muito intrincado, mas há algumas coisas sobre o DNS que você deve saber como proprietário de um site.

Segurança

Você deve sempre se lembrar de que os *nomes de domínio são valiosos*. Se um hacker comprometer totalmente o serviço de hospedagem e assumir o controle, mas for possível manter o controle do domínio, você pode obter uma nova hospedagem e redirecionar o domínio. Se, por outro lado, seu *domínio* for comprometido, você terá um grande problema. Sua reputação está associada ao seu domínio, e bons nomes de domínio são cuidadosamente protegidos. Pessoas que perderam o controle dos domínios descobriram que isso pode ser devastador, e sempre existirá quem tente ativamente comprometer nosso domínio (principalmente se ele for curto ou fácil de lembrar) para vendê-lo barato, arruinar nossa reputação ou fazer chantagem. A conclusão é que *você deve levar a sério a segurança do domínio*, talvez até mesmo mais a sério do que os dados (dependendo de quanto eles forem valiosos). Vi pessoas gastarem muito tempo e dinheiro na segurança da hospedagem enquanto usavam o registro de domínio mais barato e incompleto que pudessem encontrar. Não cometa esse erro. (Felizmente, um registro de domínio de qualidade não é particularmente caro.)

Dada a importância da proteção da propriedade de seu domínio, você deve empregar boas práticas de segurança no que diz respeito ao registro. No mínimo, deve usar senhas fortes e exclusivas, e mantê-las em um regime de

higiene apropriado (não as colocar em uma nota adesiva colada ao seu monitor). Preferivelmente, use um registrador que ofereça a autenticação de dois fatores. Não tenha medo de fazer perguntas objetivas a ele sobre o que é exigido para a autorização de alterações em sua conta. Os registradores que recomendo são o AWS Route 53, Name.com e Namecheap.com. Todos os três oferecem a autenticação de dois fatores, e achei seu suporte bom e seus painéis de controle online simples e robustos.

Quando você registrar um domínio, deve fornecer um endereço de email de terceiros que esteja associado a esse domínio (isto é, se estiver registrando *meadowlarktravel.com*, não deve usar *admin@meadowlarktravel.com* como email do inscrito). Já que qualquer sistema de segurança é tão forte quanto seu elo mais fraco, você deve usar um endereço de email com boa proteção. É muito comum o uso de uma conta do Gmail ou do Outlook, e, se você a usar, deve empregar os mesmos padrões de segurança que usa para sua conta no registrador do domínio (uma boa higiene de proteção de senha e a autenticação de dois fatores).

Domínios de nível superior

O que aparece no fim do domínio (como *.com* ou *.net*) chama-se *domínio de nível superior* (TLD, top-level-domain). De um modo geral, há dois tipos de TLD: TLDs de código de país e TLDs genéricos. Os TLDs de código de país (como *.us*, *.es* e *.uk*) são projetados para fornecer uma categorização geográfica. No entanto, há poucas restrições para quem pode adquirir esses TLDs (afinal, a internet é realmente uma rede global), logo, com frequência eles são usados para domínios mais “engraçadinhos”, como *placeholder.it* e *goo.gl*.

Os TLDs genéricos (gTLDs) incluem os conhecidos *.com*, *.net*, *.gov*, *.fed*, *.mil* e *.edu*. Embora qualquer pessoa possa adquirir um domínio *.com* ou *.net* disponível, há restrições para os outros que mencionei. Para obter mais informações, consulte a Tabela 21.1.

Tabela 21.1 – gTLDs restritos.

TLD	Mais informações
<i>.gov</i> ,	https://www.dotgov.gov

.fed	
.edu	https://net.educause.edu/
.mil	Funcionários e fornecedores da área militar devem entrar em contato com seu departamento de TI ou com o <i>Sistema de Registro Unificado do Departamento de Defesa</i> (<i>Department of Defense Unified Registration System</i> - http://bit.ly/354JvZF)

A Corporação da Internet para a Atribuição de Nomes e Números (ICANN, Internet Corporation for Assigned Names and Numbers) é responsável pelo gerenciamento dos TLDs, embora delegue grande parte da administração para outras organizações. Recentemente, a ICANN autorizou muitos gTLDs novos, como *.agency*, *.florist*, *.recipes* e até mesmo *.ninja*. Para o futuro próximo, *.com* provavelmente permanecerá sendo o TLD “premium”, e o mais difícil de obter. Pessoas que foram suficientemente afortunadas (ou astutas) para comprar domínios *.com* nos anos de formação da internet receberam quantias enormes por domínios primordiais (por exemplo, o Facebook comprou o *fb.com* em 2010 por inacreditáveis 8,5 milhões de dólares).

Dada a escassez de domínios *.com*, as pessoas estão se voltando para TLDs alternativos, ou usando *.com.us* (nos Estados Unidos) para tentar ter um domínio que reflita com precisão sua empresa. Ao selecionar um domínio, você deve considerar como ele será usado. Se estiver preocupado principalmente com o marketing eletrônico (em que as pessoas estarão mais propensas a clicar em um link em vez de digitar um domínio), deve tentar obter um domínio fácil de lembrar ou significativo, e não um curto. Se quiser dar ênfase à propaganda impressa, ou tiver razões para achar que as pessoas inserirão sua URL manualmente em seus dispositivos, considere TLDs alternativos para obter um nome de domínio mais curto. Também é prática comum a existência de dois domínios: um curto, fácil de digitar, e um mais longo mais apropriado ao marketing.

Subdomínios

Enquanto o TLD vem depois do domínio, o subdomínio vem antes dele. Com certeza o subdomínio mais comum é *www*. Nunca dei muita atenção a esse subdomínio. Afinal, você está no computador, *usando* a World Wide Web; é claro que não vai se confundir se não houver um *www* para lembrá-lo do que

está fazendo. Logo, recomendo não usar um subdomínio em seu domínio principal: use *http://meadowlarktravel.com/* em vez de *http://www.meadowlarktravel.com/*. É mais curto e menos trabalhoso, e, graças aos redirecionamentos, não há o perigo de perdemos visitas de pessoas que comecem tudo automaticamente com *www*.

Os subdomínios também são usados para outros fins. Normalmente vejo coisas como *blogs.meadowlarktravel.com*, *api.meadowlarktravel.com* e *m.meadowlarktravel.com* (para um site móvel). Com frequência isso é feito por razões técnicas: pode ser mais fácil usar um subdomínio se, por exemplo, seu blog empregar um servidor totalmente diferente do resto do site. No entanto, um bom proxy pode redirecionar tráfego apropriadamente com base no subdomínio ou no path, logo, a decisão de se será melhor usar um subdomínio ou um path deve ser baseada mais no conteúdo do que na tecnologia (lembre-se do que Tim Berners-Lee disse sobre as URLs representarem a arquitetura das informações, e não a arquitetura técnica).

Recomendo que os subdomínios sejam usados para compartimentalizar partes significativamente diferentes de seu site ou serviço. Por exemplo, acho que seria um bom uso do subdomínio se você disponibilizasse sua API em *api.meadowlarktravel.com*. Os microsites (sites que têm uma aparência diferente do resto do site, geralmente realçando um único produto ou assunto) também são bons candidatos para os subdomínios. Outro uso sensato para os subdomínios seria para separar interfaces administrativas e interfaces públicas (*admin.meadowlarktravel.com*, somente para funcionários).

A menos que você especifique o contrário, o registrador do domínio redirecionará todo o tráfego para seu servidor independentemente do subdomínio. Ficará então a cargo do seu servidor (ou proxy) executar a ação apropriada com base no subdomínio.

Servidores de nome

A “cola” que faz os domínios funcionarem são os servidores de nome, que é o que você será solicitado a fornecer quando estabelecer a hospedagem para seu site. Geralmente, isso é muito simples, já que o serviço de hospedagem faz grande parte do trabalho para nós. Por exemplo, digamos que você

quisesse hospedar *meadowlarktravel.com* na *DigitalOcean* (<https://www.digitalocean.com>). Quando definir sua conta de hospedagem, receberá os nomes dos servidores de nome da DigitalOcean (há vários para o fornecimento de redundância). A DigitalOcean, como a maioria dos provedores de hospedagem, chama seus servidores de nome de *ns1.digitalocean.com*, *ns1.digitalocean.com* e assim por diante. Vá ao registrador do domínio, defina os servidores de nome para o domínio que deseja hospedar, e isso é tudo.

A maneira como o mapeamento funciona nesse caso é a seguinte:

1. O visitante do site navega para <http://meadowlarktravel.com/>.
2. O navegador envia a requisição para o sistema de rede do computador.
3. O sistema de rede do computador, que recebeu um endereço IP e um servidor DNS fornecidos pelo provedor de internet, solicita ao resolvidor DNS que resolva *meadowlarktravel.com*.
4. O resolvidor DNS sabe que *meadowlarktravel.com* é manipulado por *ns1.digitalocean.com*, logo, ele solicita a *ns1.digitalocean.com* que lhe dê um endereço IP para *meadowlarktravel.com*.
5. O servidor em *ns1.digitalocean.com* recebe a requisição, reconhece que *meadowlarktravel.com* é realmente uma conta ativa e retorna o endereço IP associado.

Embora esse seja o caso mais comum, não é a única maneira de configurar o mapeamento de domínio. Já que o servidor (proxy) que serve o site tem um endereço IP, podemos eliminar o intermediário registrando esse endereço IP nos resolvidores DNS (isso eliminaria o intermediário do servidor de nome *ns1.digitalocean.com* no exemplo anterior). Para essa abordagem funcionar, o serviço de hospedagem deve atribuir para você um endereço IP *estático*. Normalmente, os provedores de hospedagem fornecem para o(s) servidor(es) um endereço IP *dinâmico*, ou seja, ele pode mudar sem aviso, o que tornaria esse esquema ineficaz. Pode custar mais obter um endereço IP estático em vez de um dinâmico: verifique com seu provedor de hospedagem.

Se você quiser mapear seu domínio para o site diretamente (pulando os servidores de nome do host), terá de adicionar um registro A ou CNAME. O

registro A mapeia um domínio diretamente para um endereço IP, enquanto *CNAME* mapeia um nome de domínio para outro. Geralmente os registros *CNAME* são menos flexíveis, logo, os registros *A* são preferíveis.



Se você estiver usando a AWS para obter servidores de nome, além dos registros *A* e *CNAME*, também haverá um registro chamado *alias* que oferece muitas vantagens quando apontado para um serviço hospedado nesse provedor. Para obter mais informações, consulte a *documentação da AWS* (<https://amzn.to/2pUuDhv>).

Seja qual for a técnica usada, geralmente o mapeamento de domínio é armazenado rigorosamente em cache, o que significa que, quando você alterar os registros de domínio, pode levar até 48 horas para seu domínio ser anexado ao novo servidor. Lembre-se de que isso também depende da geografia: se seu domínio estiver operando em Los Angeles, seu cliente em Nova York pode ver o domínio anexado ao servidor anterior. Pela minha experiência, 24 horas costumam ser suficientes para os domínios serem resolvidos corretamente nos Estados Unidos continentais, com a resolução internacional levando até 48 horas.

Se você precisar que algo tenha exibição ao vivo em uma hora específica, não deve confiar nas alterações do DNS. Em vez disso, modifique seu servidor para fazer o redirecionamento para o “próximo” site ou página e faça as alterações de DNS antes da mudança real. Dessa forma, na hora fixada, você poderá fazer seu servidor passar para o live site e seus visitantes verão a mudança imediatamente, sem importar onde estejam no mundo.

Hospedagem

Inicialmente pode parecer complicado selecionar um serviço de hospedagem. O Node se popularizou solidamente, e todo mundo alardeia oferecer sua hospedagem para atender à demanda. Como você deve selecionar um provedor de hospedagem vai depender de suas necessidades. Se tiver razões para crer que seu site será o próximo Amazon ou Twitter, terá preocupações diferentes das que teria se estivesse construindo um site para seu cube de filatelistas local.

Hospedagem tradicional ou em nuvem?

“Nuvem” é um dos termos técnicos mais nebulosos que surgiu nos últimos anos. Na verdade, é apenas uma maneira extravagante de dizer “a internet” ou “parte da internet”. No entanto, não é totalmente inútil. Embora não faça parte da definição técnica do termo, geralmente a hospedagem na nuvem implica certa comoditização dos recursos de computação. Isso significa que não pensamos mais em um “servidor” como uma entidade física distinta: ele é simplesmente um recurso homogêneo em algum local na nuvem, e um é tão bom quanto o outro. É claro que estou sendo simplista: os recursos de computação são distinguidos (e precificados) de acordo com sua memória, número de CPUs etc. A diferença está entre sabermos em que servidor o aplicativo está hospedado (e nos importarmos com isso) e sabermos que ele está hospedado em *algum* servidor na nuvem, e pode ser facilmente transferido para outro sem tomarmos conhecimento (ou nos preocuparmos).

A hospedagem em nuvem também é altamente *virtualizada*. Isto é, em geral o servidor no qual o aplicativo é executado não é uma máquina física, e sim uma máquina virtual sendo executada em um servidor físico. Essa ideia não foi introduzida pela hospedagem em nuvem, mas passou a representá-la.

Embora a hospedagem em nuvem tenha origem simples, atualmente ela significa muito mais do que apenas “servidores homogêneos”. Os principais provedores de nuvem oferecem vários serviços de infraestrutura que (teoricamente) reduzem o peso da manutenção e fornecem alto grau de escalabilidade. Esses serviços incluem armazenamento de banco de dados, armazenamento de arquivos, filas de rede, autenticação, processamento de vídeo, serviços de telecomunicações, engines de inteligência artificial e muito mais.

Inicialmente a hospedagem em nuvem pode ser um pouco desconcertante, por não sabermos nada sobre a máquina física na qual o servidor está sendo executado, acreditando que nossos servidores não serão afetados por outros que estejam em execução no mesmo computador. No entanto, na verdade nada mudou: quando sua conta de hospedagem chegar, você ainda estará pagando basicamente pela mesma coisa: alguém se encarregando do hardware físico e dos recursos de rede que disponibilizam suas aplicações

web. A única coisa que mudou é que você ficou mais distante do hardware.

Acredito que a hospedagem “tradicional” (na falta de um termo melhor) acabará desaparecendo. Isso não quer dizer que as empresas de hospedagem sairão do mercado (embora inevitavelmente isso vá ocorrer com algumas); apenas começarão a oferecer elas próprias hospedagem na nuvem.

XaaS

Ao considerar a hospedagem na nuvem, você deparará com os acrônimos SaaS, PaaS, IaaS e FaaS:

Software como Serviço (SaaS, Software as a Service)

Geralmente o SaaS descreve um software (sites, aplicativos) que nos é fornecido: temos apenas de usá-lo. Um exemplo seria o Google Documents ou o Dropbox.

Plataforma como Serviço (PaaS, Platform as a Service)

O PaaS nos fornece toda a infraestrutura (sistemas operacionais, rede – tudo isso é manipulado). Só temos de criar as aplicações. Embora haja uma linha tênue entre PaaS e IaaS (e, como desenvolvedor, você se verá com frequência transpondo essa linha), esse é o modelo de serviço que por vezes estamos discutindo neste livro. Se está executando um site ou web service, provavelmente o PaaS é o que procura.

Infraestrutura como Serviço (IaaS, Infrastructure as a Service)

O IaaS fornece maior flexibilidade, mas há um preço. Tudo que obtemos são máquinas virtuais e uma rede básica conectando-as. Logo, somos responsáveis por instalar e fazer a manutenção de sistemas operacionais, bancos de dados e políticas de rede. A menos que você precise desse nível de controle sobre seu ambiente, geralmente é melhor usar o PaaS. (É bom ressaltar que o PaaS nos permite *escolher* os sistemas operacionais e a configuração de rede: só não precisamos cuidar disso sozinhos).

Funções como Serviço (FaaS, Functions as a Service)

O FaaS inclui serviços como o AWS Lambda, o Google Functions e o

Azure Functions, que fornecem uma maneira de executar funções individuais na nuvem sem que nós próprios precisemos configurar o ambiente de runtime. É a essência do que normalmente está sendo chamado de arquitetura “serverless”.

Os gigantes

As empresas que basicamente conduzem a internet (ou, pelo menos, que estão muito envolvidas na condução da internet) perceberam que, com a comoditização dos recursos de computação, elas têm outro produto viável para vender. A Amazon, a Microsoft e o Google oferecem serviços de computação em nuvem, e seus serviços são muito bons.

Todos esses serviços são precificados de maneira semelhante: se suas necessidades de hospedagem forem modestas, haverá uma diferença de preço mínima entre os três. Se você tiver necessidades de largura de banda ou armazenamento muito grandes, terá de avaliar os serviços com mais cuidado, já que a diferença de custo pode ser maior, dependendo das necessidades.

Embora normalmente não nos lembremos da Microsoft quando consideramos plataformas open source, eu não me esqueceria do Azure. Além de a plataforma ser estabelecida e robusta, a Microsoft tem se esforçado para torná-la amigável não só ao Node, mas à comunidade open source. A Microsoft oferece um período de teste de um mês, o que é uma ótima maneira de determinar se o serviço atende às necessidades; se você estiver considerando um desses três grandes serviços, definitivamente recomendo o período de teste gratuito para avaliação do Azure. A Microsoft fornece APIs Node para todos os seus principais serviços, inclusive seu serviço de armazenamento em nuvem. Além da excelente hospedagem do Node, o Azure oferece um ótimo sistema de armazenamento em nuvem (com uma API JavaScript), assim como um bom suporte ao MongoDB.

A Amazon oferece o conjunto de recursos mais abrangente, incluindo SMS (mensagem de texto), armazenamento em nuvem, serviços de email, serviços de pagamento (e-commerce), DNS e mais. Além disso, fornece uma camada de uso gratuita, facilitando a avaliação.

A plataforma de nuvem do Google avançou muito e agora oferece a

hospedagem robusta do Node e, como era de se esperar, uma excelente integração com seus próprios serviços (o mapeamento, a autenticação e a busca sendo particularmente interessantes).

Além dos “grandes três”, vale a pena considerar o *Heroku* (<https://www.heroku.com>), que há algum tempo atende pessoas que desejam hospedar aplicações Node rápidas e eficientes. Também tive muita sorte com a *DigitalOcean* (<https://www.digitalocean.com>), que dá mais ênfase ao fornecimento de contêineres e a um número limitado de serviços de maneira amigável para o usuário.

Hospedagem de boutique

Serviços de hospedagem menores, que chamarei de serviços de hospedagem de “boutique” (na falta de uma palavra melhor), podem não ter a infraestrutura ou os recursos da Microsoft, da Amazon ou do Google, mas isso não significa que não ofereçam algo de valor.

Já que os serviços de hospedagem de boutique não podem competir em termos de infraestrutura, geralmente enfocam o serviço e o suporte ao cliente. Se precisar de muito suporte, considere um serviço de hospedagem de boutique. Se você tiver um provedor de hospedagem que lhe agradou, não hesite em perguntar se ele oferece (ou pretende oferecer) a hospedagem do Node.

Implantação

Ainda me surpreende que, em 2020, as pessoas ainda estejam usando o FTP para implantar suas aplicações. Se você estiver, *por favor pare*. O FTP não é de forma alguma seguro. Além de todos os arquivos serem transmitidos sem criptografia, o *nome de usuário e a senha* também são. Se seu provedor de hospedagem não lhe der outra opção, encontre um novo provedor. Se não houver saída, certifique-se de usar uma senha exclusiva que você não esteja usando para mais nada.

No mínimo, você deve usar o SFTP ou o FTPS (não confunda), mas o que deve considerar mesmo é um serviço de *distribuição contínua* (CD, continuous delivery).

A ideia existente por trás da CD é a de que nunca estamos longe demais de

uma versão que possa ser lançada (semanas ou até mesmo dias). Geralmente a CD é usada ao mesmo tempo que a *integração contínua* (CI, continuous integration), que são processos automatizados que integram e testam o trabalho dos desenvolvedores.

Em geral, quanto mais automatizamos nossos processos, mais fácil é o desenvolvimento. Imagine adicionar alterações, ser notificado automaticamente de que os testes unitários foram bem-sucedidos, que o mesmo ocorreu com os testes de integração e ver suas alterações online... em questão de minutos! É um ótimo objetivo, mas você tem de fazer algum trabalho anterior para preparar tudo, e haverá alguma manutenção com o tempo.

Embora essas etapas sejam semelhantes (executar testes unitários, executar testes de integração, implantar em servidores de teste, implantar em servidores de produção), o processo de definir pipelines CI/CD (um termo que você ouvirá muito quando discutir a CI/CD) varia substancialmente.

Você deve examinar algumas das opções disponíveis para a CI/CD e selecionar uma que atenda a suas necessidades:

AWS CodePipeline (<https://amzn.to/2CzTQAo>)

Se você estiver executando a hospedagem na AWS, o CodePipeline deve ser a primeira opção de sua lista, já que será o modo mais fácil de realizar a CI/CD. Ele é mais robusto, porém achei-o um pouco menos amigável para o usuário do que algumas das outras opções.

Microsoft Azure Web Apps (<http://bit.ly/2CEsSI0>)

Se estiver executando a hospedagem no Azure, o Web Apps será sua melhor opção (está notando alguma tendência aqui?). Não tenho muito experiência com esse serviço, mas ele parece bastante apreciado na comunidade.

Travis CI (<https://travis-ci.org/>)

O Travis CI já existe há algum tempo e tem uma grande e leal base de usuários e boa documentação.

Semaphore (<https://semaphoreci.com/>)

O Semaphore é fácil de instalar e configurar, mas não oferece muitos recursos e seus planos básicos (de baixo custo) são lentos.

Google Cloud Build (<http://bit.ly/2NGuIys>)

Ainda não testei o Google Cloud Build, mas ele parece robusto e, como o CodePipeline e o Azure Web Apps, talvez seja a melhor opção se você estiver executando a hospedagem no Google Cloud.

CircleCI (<https://circleci.com/>)

O CircleCI é outro recurso de CI que já existe há algum tempo e é bastante apreciado.

Jenkins (<https://jenkins.io/>)

O Jenkins é outro beneficiado com uma grande comunidade. Pela minha experiência acho que ele não acompanhou as práticas modernas de implantação tão bem quanto algumas das outras opções exibidas aqui, mas lançou uma nova versão que parece promissora.

No fim das contas, os serviços de CI/CD automatizarão as atividades que você criar. Ainda será preciso escrever o código, determinar o esquema de versionamento, criar testes unitários e de integração de alta qualidade e uma maneira de executá-los, e entender a infraestrutura de implantação. É muito fácil automatizar os exemplos deste livro: quase tudo pode ser implantado em um único servidor executando uma instância do Node. No entanto, à medida que você começar a aumentar sua infraestrutura, a complexidade do pipeline CI/CD também crescerá.

O papel do Git na implantação

O ponto mais forte (e a maior fraqueza) do Git é sua flexibilidade. Ele pode ser adaptado a quase qualquer fluxo de trabalho imaginável. Para facilitar, recomendo a criação de um ou mais branches *especificamente para a implantação*. Por exemplo, você poderia ter um branch de produção e um de teste. Como usará esses branches vai depender de seu fluxo de trabalho.

Uma abordagem popular é executar o fluxo do mestre para o teste e depois para

a produção. Assim, quando algumas alterações no mestre estiverem prontas para entrar online, você poderá trazê-las para o teste. Quando elas forem aprovadas no servidor de teste, o branch de teste será trazido para a produção. Embora esse esquema faça sentido, não gosto da desordem que ele cria (fusões em todos os locais). Além disso, se você tiver muitos recursos que precisem ser testados e trazidos para a produção em ordens diferentes, isso pode se tornar rapidamente confuso.

Acho que uma abordagem melhor seria fundir o mestre no teste e, quando você estiver pronto para entrar online, fundir o mestre na produção. Dessa forma, o teste e a produção ficarão menos associados: você poderia até mesmo ter vários branches de teste para ver como recursos distintos se saem antes de entrarem online (e fundir algo que não seja o mestre nesses branches). Só quando alguma coisa for aprovada para a produção é que você a trará para o branch de produção.

O que você deve fazer quando precisar reverter alterações? É nesse momento que tudo pode se complicar. Há várias técnicas para a reversão de alterações, como aplicar o inverso de um commit para desfazer commits anteriores (git revert); porém, além de essas técnicas serem complicadas, elas também podem causar problemas mais à frente. A maneira típica de manipular isso é criando tags (por exemplo, git tag v1.2.0 no branch de produção) sempre que você fizer uma implantação. Se precisar voltar para uma versão específica, terá essa tag disponível.

É responsabilidade sua e de sua equipe optar por um fluxo de trabalho do Git. Mais importante do que o fluxo de trabalho que você selecionar será a consistência com que o usar, e o treinamento e a comunicação que o cercarem.



Já discutimos o valor de mantermos os arquivos binários (multimídia e documentos) separados do repositório de código. A implantação baseada no Git oferece outro incentivo para essa abordagem. Se você tiver 4 GB de dados multimídia em seu repositório, será demorado cloná-los, e haveria uma cópia desnecessária de todos os dados para cada servidor de produção.

Implantação manual baseada no Git

Se você ainda não estiver pronto para executar a etapa de definir a CI/CD, poderia começar com uma implantação manual baseada no Git. A vantagem dessa abordagem é que você não terá problemas com as etapas e desafios envolvidos na implantação, o que lhe ajudará quando migrar para a automação.

Para cada servidor no qual você quiser fazer a implantação, será preciso clonar o repositório, extrair o branch de produção e definir a infraestrutura necessária para a inicialização/reinicialização do aplicativo (o que dependerá da escolha da plataforma). Quando você atualizar o branch de produção, terá de ir a cada servidor, executar `git pull --ff-only`, executar `npm install --production` e reiniciar o aplicativo. Se suas implantações não forem frequentes, e você não tiver muitos servidores, talvez isso não represente um sofrimento tão grande, mas, se a atualização for mais frequente, tudo se desatualizará com mais rapidez, e pode ser melhor encontrar alguma maneira de automatizar o sistema.



O argumento `--ff-only` de `git pull` só permite fast-forward pulls, impedindo a fusão ou o rebasing automático. Se você souber que o pull só ocorrerá em fast-forward, pode omiti-lo com segurança, mas, se se habituar a fazê-lo, nunca chamará acidentalmente uma fusão ou um rebase!

Na verdade, o que você está fazendo aqui é replicando a maneira como trabalha no desenvolvimento, exceto por estar em um servidor remoto. Os processos manuais estão sempre sujeitos aos erros humanos, e só recomendo essa abordagem como um trampolim para o desenvolvimento mais automatizado.

Conclusão

A implantação de seu site (principalmente na primeira vez) deveria ser uma ocasião estimulante. Deveria haver champanhe e festa, mas geralmente há muito trabalho, problemas e madrugadas passadas em claro. Vi muitos sites serem lançados às três da manhã por uma equipe irritada e exausta. Felizmente, isso está mudando, em parte graças à implantação em nuvem.

Independentemente da estratégia de implantação adotada, a coisa mais importante que você pode fazer é iniciar as implantações na produção mais cedo, antes de o site entrar online. Não é necessário incluir o domínio, logo, o público não precisa saber. Se você já tiver implantado o site nos servidores de produção algumas vezes antes do dia do lançamento, suas chances de um lançamento bem-sucedido serão muito mais altas. Idealmente, seu site funcional deve ser executado no servidor de produção bem antes do lançamento: tudo que você terá de fazer é passar do site antigo para o novo.

CAPÍTULO 22

Manutenção

Você lançou o site! Parabéns, não precisará mais pensar nisso. Como? *Preciso* continuar me preocupando? Bem, nesse caso, continue lendo.

Embora isso tenha acontecido algumas vezes em minha carreira, é uma exceção à regra terminarmos um site e nunca mais precisarmos alterá-lo (e, quando acontece, em geral é porque alguma outra pessoa está se encarregando do trabalho e não porque ele não precise ser feito). Lembro-me claramente do “post-mortem” de lançamento de um site. Suspirei e disse “Não deveríamos chamar de *pós-parto*?”¹. Na verdade, lançar um site é um nascimento, e não o fim. Uma vez que ele é lançado, ficamos presos a análises, esperando ansiosamente a reação do cliente, acordando às três da manhã para ver se ele ainda está funcionando. É o seu bebê.

Pensar no escopo de um site, projetá-lo e construí-lo: todas essas atividades podem ser planejadas exaustivamente. No entanto, o que em geral recebe pouca atenção é o *planejamento da manutenção* do site. Este capítulo fornecerá algumas sugestões para a execução dessa etapa.

Princípios da manutenção

Haver um plano de longevidade

Fico sempre surpreso quando um cliente concorda com o preço de construção de um site, mas não se discute quanto tempo é esperado que ele dure. Em minha experiência, vi que, se fizermos um bom trabalho, os clientes ficarão felizes em pagar por ele. O que os clientes *não* apreciam é o inesperado: serem informados após três anos de que seu site precisa ser reconstruído quando tinham uma expectativa não externada de que ele durasse cinco anos.

A internet avança rapidamente. Mesmo se você construir um site com a melhor e mais recente tecnologia que puder encontrar, ele parecerá uma relíquia em apenas dois anos. Ou pode funcionar por sete, envelhecendo, mas de maneira graciosa (isso é muito menos comum!).

Definir expectativas sobre a longevidade do site é parte arte, parte vendas e parte ciência. A parte ciência envolve algo que todos os cientistas, mas muito poucos desenvolvedores web, fazem: manter registros. Imagine se você tivesse um registro de todos os sites que sua equipe já lançou, o histórico de requisições de manutenção e falhas, as tecnologias usadas e o tempo que passou até cada site ser reconstruído. Há muitas variáveis, claro, dos membros da equipe envolvida, de economia, das mudanças na tecnologia, mas isso não significa que tendências relevantes não possam ser descobertas nos dados. Você pode descobrir que certas abordagens de desenvolvimento funcionam melhor para sua equipe, ou certas plataformas ou tecnologias. O que posso quase garantir que você descobrirá é uma relação entre “procrastinação” e defeitos: quanto mais você adiar a atualização na infraestrutura ou o upgrade da plataforma que está causando problemas, pior será. A existência de um bom sistema de rastreamento de problemas e a manutenção de registros meticulosos permitirão que você forneça para o cliente um cenário muito melhor (e mais realista) de qual será o ciclo de vida do projeto.

A parte vendas se resume a dinheiro, claro. Se um cliente puder pagar para ter seu site totalmente reconstruído a cada três anos, ele não deve ter muitos problemas de obsolescência da infraestrutura (no entanto, terá outros). Por outro lado, alguns clientes precisarão aproveitar ao máximo o dinheiro investido, desejando que o site dure por cinco ou até mesmo sete anos. (Conheci sites que se arrastaram por ainda mais tempo, mas acho sete anos a expectativa de vida mais realista para que eles continuem sendo úteis.) Temos de ser responsáveis com esses dois tipos de clientes, e ambos trazem os próprios desafios. No caso dos clientes que têm muito dinheiro, não cobre simplesmente porque eles podem pagar: use essa maior disponibilidade financeira para lhes dar algo extraordinário. Para os clientes de orçamento apertado, você terá de encontrar maneiras criativas de projetar seu site para fornecer maior longevidade diante do contínuo avanço da tecnologia. Esses

dois extremos apresentam os próprios desafios, mas eles podem ser superados. O que é importante, no entanto, é que você *saiba* quais são as expectativas.

Para concluir, há a parte arte da questão. É ela que amarra tudo: saber o que o cliente pode pagar e onde você pode convencê-lo honestamente a gastar mais para obter valor onde precisa. Também é a arte de conhecer o futuro da tecnologia e prever quais serão obsoletas em cinco anos e quais ficarão mais robustas.

É claro que não há como prever algo com absoluta certeza. Você pode apostar em tecnologias erradas, mudanças de pessoal podem alterar totalmente a cultura técnica de sua empresa, e fornecedores de tecnologia podem sair do mercado (embora geralmente esse seja um problema menor no universo open source). A tecnologia que você pensou que seria sólida durante o tempo de vida de seu produto pode acabar sendo uma moda passageira, e será preciso encarar a decisão de uma reconstrução mais cedo do que o esperado. Por outro lado, às vezes a equipe certa se reúne na hora exata e é criado algo que sobrevive além de qualquer expectativa. No entanto, nenhuma dessas incertezas devem impedi-lo de ter um plano: melhor termos um plano que dê errado do que agir sem meta.

Já deve ter ficado claro para você que acho o JavaScript e o Node tecnologias que ainda estarão por aí durante algum tempo. A comunidade do Node é vibrante e entusiasmada, e amplamente baseada em uma linguagem que claramente é *vencedora*. Talvez o mais importante seja que o JavaScript é uma linguagem multiparadigma: orientada a objetos, funcional, procedural, síncrona, assíncrona – ela engloba tudo. Isso a torna uma plataforma convidativa para desenvolvedores de muitos backgrounds diferentes, e é em grande parte responsável pelo ritmo de inovação no ecossistema JavaScript.

Use o controle de código-fonte

Isso pode parecer óbvio, mas não se trata apenas de *usar* o controle de código-fonte, e sim de usá-lo *bem*. Por que você está usando o controle de código-fonte? Entenda as razões e certifique-se de que as ferramentas as apoiem. Há muitas razões para o uso do controle de código-fonte, mas a que

me parece trazer o maior benefício é a atribuição: saber que alteração foi feita quando e por quem, para podermos pedir mais informações se necessário. O controle de versões é uma das melhores ferramentas para sabermos o histórico de nossos projetos e como trabalhamos juntos como uma equipe.

Use um rastreador de problemas

Os rastreadores de problemas (issue trackers) nos levam de volta à ciência do desenvolvimento. Sem uma maneira sistemática de registrarmos o histórico de um projeto, nenhum insight é possível. Você já deve ter ouvido falar que a definição de insanidade é “fazer a mesma coisa várias vezes e esperar resultados diferentes” (com frequência atribuída dubiamente a Albert Einstein). Parece realmente insano repetirmos nossos erros várias vezes, mas como evitar se não soubermos que erros estamos cometendo?

Registre tudo: cada defeito que o cliente relatar; cada falha que você encontrar antes de o cliente ver; cada reclamação, cada pergunta, todos os elogios. Registre quanto tempo durou, quem corrigiu, que commits do Git estavam envolvidos e quem aprovou a correção. A arte aqui é encontrar ferramentas que não tornem isso muito demorado ou oneroso. Um sistema de rastreamento de problemas inadequado definhará, sem ser usado, o que é pior do que ser inútil. Um sistema eficaz produzirá insights vitais em sua empresa, sua equipe e seus clientes.

Pratique uma boa higiene

Não estou falando sobre escovar os dentes – no entanto, você também deve fazer isso –, falo sobre o controle de versões, teste, revisões de código e rastreamento de problemas. As ferramentas só serão úteis se você realmente as usar, e usá-las da maneira correta. As revisões de código são uma ótima maneira de encorajar a higiene porque *qualquer coisa* pode ser abordada, desde a discussão do uso do sistema de rastreamento de problemas em que a requisição se originou aos testes que tiveram de ser adicionados para a verificação da correção nos comentários de commits do controle de versões.

Os dados que você coletar no sistema de rastreamento de problemas devem ser revisados periodicamente e discutidos com a equipe. A partir deles, você

terá insights sobre o que está ou não funcionando. Você pode ficar surpreso com o que descobrir.

Não procrastine

A procrastinação institucional pode ser uma das coisas mais difíceis de combater. Geralmente é algo que não parece tão ruim: você nota que sua equipe está demorando horas em uma atualização semanal que poderia ser melhorada drasticamente por uma pequena refatoração. Cada semana de atraso na refatoração será outra semana na qual você estará pagando o custo da ineficiência.² E o pior é que alguns custos podem aumentar com o tempo.

Um ótimo exemplo é não atualizar dependências de software. À medida que o software for ficando mais velho, e os membros da equipe mudarem, será mais difícil encontrar pessoas que se lembrem (ou até mesmo entendam) do antigo software. A comunidade de suporte começará a desaparecer, e, em breve, a tecnologia estará obsoleta e você não conseguirá nenhum tipo de suporte. Com frequência ouvimos esse problema ser descrito como *dívida técnica*, que é algo muito real. Embora você deva evitar a procrastinação, conhecer a longevidade do site pode ajudar nessas decisões: se estiver na hora de reprojeter o site inteiro, não valerá a pena eliminar a dívida técnica que foi gerada.

Faça verificações de QA rotineiras

Para cada um de seus sites, você deve ter uma verificação de QA de rotina *documentada*. Essa verificação deve incluir um verificador de link, a validação de HTML e CSS e a execução de testes. O segredo aqui é *documentar*: se os itens que compõem a verificação de QA não forem documentados, inevitavelmente você perderá coisas. Um checklist de QA documentado para cada site não só ajuda a evitar que verificações sejam omitidas, como também permite que novos membros da equipe sejam imediatamente eficientes. Idealmente, o checklist deve poder ser executado por um membro não técnico da equipe. Isso fará com que seu gerente (possivelmente) não técnico confie na equipe e permita que você distribua responsabilidades de QA se não houver um departamento para essa área.

Dependendo de seu relacionamento com o cliente, você também pode querer compartilhar com ele seu checklist de QA (ou parte dele); é uma boa maneira de lembrá-lo pelo que está pagando e que os seus interesses estão sendo considerados.

Como parte de sua verificação de QA de rotina, recomendo usar o *Google Webmaster Tools* (<http://bit.ly/2qH3Y7L>) e o *Bing Webmaster Tools* (<https://binged.it/2qPwF2c>). Eles são fáceis de configurar e fornecem uma visão muito importante do site: como os principais mecanismos de busca o veem. Ele o alertará sobre qualquer problema com o arquivo *robots.txt*, problemas de HTML que estejam interferindo na obtenção de bons resultados de busca e muito mais.

Monitore a análise

Se você não estiver executando análises em seu site, precisa começar agora: elas fornecem um insight vital não só sobre a popularidade do site, mas também sobre como seus usuários estão usando-o. O Google Analytics (GA) é ótimo (e gratuito!), e, mesmo se você complementá-lo com serviços de análise adicionais, não há por que não o incluir no site.

Com frequência você conseguirá identificar problemas sutis de UX verificando a análise. Certas páginas não estão tendo o tráfego esperado? Isso pode indicar um problema de navegação ou de promoções, ou pode ser uma questão de SEO. Suas taxas de rejeição estão altas? Talvez o conteúdo das páginas precise de alguma adaptação (as pessoas estão conhecendo seu site pela busca, mas, quando chegam nele, percebem que não é o que procuram). É preciso que um checklist de análise acompanhe o checklist de QA (ele poderia até mesmo fazer parte do checklist de QA). Esse checklist deve ser um “documento em progresso”, porque, durante o tempo de vida do site, você ou seu cliente podem querer fazer mudanças de prioridades em relação a que conteúdo seria o mais importante.

Otimize o desempenho

Vários estudos têm mostrado o efeito dramático do desempenho sobre o tráfego de um site. Estamos em um mundo veloz, e as pessoas esperam que

seu conteúdo seja distribuído com rapidez, principalmente em plataformas móveis. O princípio número um do ajuste de desempenho é *primeiro rastrear o perfil e depois otimizar*. “Rastrear o perfil” (profiling) significa descobrir o que está realmente tornando o site lento. Se você passar dias acelerando a renderização do conteúdo, mas o problema forem os plugins de mídia social, desperdiçará muito tempo e dinheiro.

O *Google PageSpeed Insights* (<http://bit.ly/2Qa3l15>) é uma ótima maneira de avaliar o desempenho do site (e agora seus dados são registrados no Google Analytics para podermos monitorar tendências de desempenho). Além de fornecer uma pontuação (escore) geral para o desempenho móvel e de desktop, ele também dá sugestões priorizadas sobre como melhorar o desempenho.

A menos que você esteja tendo problemas de desempenho atualmente, não deve precisar fazer verificações periódicas (monitorar o Google Analytics em busca de alterações significativas nas pontuações de desempenho deve ser suficiente). No entanto, é gratificante observar o aumento no tráfego quando o desempenho melhora.

Priorize o rastreamento de leads

No mundo da internet, o sinal mais forte que os visitantes podem dar para indicar interesse em seu produto ou serviço são as informações de contato. Você deve tratar essas informações com o máximo de cuidado. Qualquer formulário que colete um email ou número de telefone deve ser testado rotineiramente como parte de seu checklist de QA, e deve *sempre* haver redundância quando você coletar essas informações. A pior coisa que você pode fazer para um possível cliente é coletar informações e depois as perder.

Já que o rastreamento de leads (lead tracking) é tão crítico para o sucesso do site, recomendo estes cinco princípios para a coleta de informações:

Tenha um fallback para o caso de o JavaScript falhar

É aconselhável a coleta de informações com Ajax – com frequência resulta em uma melhor experiência de usuário. No entanto, se o JavaScript falhar por alguma razão (o usuário poderia desativá-lo ou um script do site poderia

ter um erro, impedindo o Ajax de funcionar corretamente), mesmo assim o envio do formulário precisa funcionar. Uma ótima maneira de testar isso é desativando o JavaScript e usando o formulário. Não há problema se a experiência do usuário não for a ideal: o importante é que seus dados não sejam perdidos. Para implementar esse princípio, tenha *sempre* um parâmetro `action` válido e funcional em sua tag `<form>`, mesmo se normalmente usar Ajax.

Se usar Ajax, obtenha a URL no parâmetro `action` do formulário

Embora isso não seja estritamente necessário, ajuda a nos impedir de esquecer acidentalmente do parâmetro `action` nas tags `<form>`. Se você associar o Ajax ao envio bem-sucedido sem JavaScript, será muito mais difícil perder dados de clientes. Por exemplo, a tag de seu formulário poderia ser `<form action="/submit/email" method="POST">`; em seguida, em seu código Ajax, você obteria a ação (`action`) do formulário no DOM e a usaria no código Ajax de envio.

Forneça pelo menos um nível de redundância

Provavelmente você vai querer salvar os leads em um banco de dados ou em um serviço externo como o Campaign Monitor. No entanto, e se seu banco de dados falhar, o Campaign Monitor não funcionar ou houver um problema na rede? Você continuará não querendo perder esse lead. Uma maneira comum de fornecer redundância é enviando um email além de armazenar o lead. Se você usar essa abordagem, não deve empregar o endereço de email da pessoa, e sim um endereço compartilhado (como `dev@meadowlarktravel.com`): a redundância não ajudará se fizermos o envio para uma pessoa e ela tiver saído da empresa. Você também poderia armazenar o lead em um banco de dados de backup, ou até mesmo em um arquivo CSV. Porém, *sempre* que seu armazenamento primário falhar, é preciso que haja algum mecanismo para alertá-lo sobre a falha. Coletar um backup redundante é a primeira parte da batalha; estar consciente das falhas e tomar medidas apropriadas é a segunda.

Em caso de falha total no armazenamento, informe ao usuário

Digamos que você tivesse três níveis de redundância: seu armazenamento primário usa o Campaign Monitor, e, se ele falhar, o backup será feito em um arquivo CSV e um email será enviado para *dev@meadowlarktravel.com*. Se *todos* esses canais falharem, o usuário deve receber uma mensagem dizendo algo como “Desculpe, estamos com dificuldades técnicas. Favor tentar novamente depois ou entre em contato com *support@meadowlarktravel.com*”.

Trabalhe com uma confirmação positiva, e não com a ausência de erro

É muito comum o manipulador Ajax retornar um objeto com uma propriedade `err` no caso de falha; o código cliente então apresenta algo semelhante a: `if(data.err){ /* informa ao usuário sobre a falha */ } else { /* agradece ao usuário pelo envio bem-sucedido */ }`. Evite tal abordagem. Não há nada de errado em definir uma propriedade `err`, mas, se houver um erro em seu manipulador Ajax, que leve o servidor a responder com um código 500 ou com algo que não seja JSON válido, *essa abordagem pode falhar silenciosamente*. O lead do usuário desaparecerá e ele continuará sem informações. Em vez disso, forneça uma propriedade `success` para o envio bem-sucedido (mesmo se o armazenamento primário falhar: se as informações do usuário forem registradas por *algo*, você pode retornar `success`). O código do lado do cliente será então `if(data.success){ /* agradece ao usuário pelo envio bem-sucedido */ } else { /* informa ao usuário sobre a falha */ }`.

Evite falhas “invisíveis”

Vejo isso o tempo todo: os desenvolvedores estão com pressa e registram os erros de maneiras que nunca são verificadas. Seja em um logfile, uma tabela em um banco de dados, um log de console no lado do cliente ou um email que vá até um endereço inativo, o resultado final é o mesmo: *o site tem problemas de qualidade que não estão sendo percebidos*.

A defesa número um que você pode ter contra esse problema é *fornecer um método padrão fácil para o registro de erros*. Documente-o. Não o torne difícil. Não o torne obscuro. Certifique-se de que todos os desenvolvedores que estiverem envolvidos em seu projeto o conheçam. Essa abordagem poderia ser tão simples quanto expor uma função `meadowlarkLog` (log é com

frequência usado por outros pacotes). Não importa se a função está fazendo o registro em um banco de dados, um arquivo simples, um email ou alguma combinação desses recursos: o importante é que ela seja padrão. Isso também permitirá que você melhore seu mecanismo de logging (por exemplo, os arquivos simples são menos úteis quando fazemos o scale out do servidor, logo, você teria de modificar sua função meadowlarkLog para fazer o registro em um banco de dados). Uma vez que tiver o mecanismo de logging definido, documentado e todos em sua equipe o conhecerem, adicione “logs de verificação” ao checklist de QA e crie instruções de como usá-los.

Refatoração e reutilização de código

Algo trágico que vejo o tempo todo é a reinvenção da roda repetindo-se sem fim. Geralmente são apenas coisas pequenas: tarefas menores que as pessoas acham mais fácil recriar do que procurar em algum projeto executado meses atrás. Todos esses pequenos fragmentos reescritos acabam se tornando volumosos. Pior, afrontam uma boa QA: provavelmente você não vai querer criar testes para todos esses fragmentos (e, se o fizer, dobrará o tempo gasto por não reutilizar código existente). Cada fragmento – de execução da mesma tarefa – pode ter bugs diferentes. É um mau hábito.

O desenvolvimento no Node e no Express oferece algumas maneiras interessantes de combater esse problema. O Node trouxe o namespacing (pelos módulos) e os pacotes (pelo npm), e o Express traz o conceito de middleware. Com essas ferramentas à disposição, fica muito mais fácil desenvolver código reutilizável.

Registro privado do npm

Os registros do npm são um ótimo local para armazenar código compartilhado; é para isso que o npm foi projetado, afinal. Além do armazenamento simples, há o versionamento, e uma maneira conveniente de incluir esses pacotes em outros projetos.

No entanto, há uma desvantagem: a menos que você esteja trabalhando em uma empresa totalmente open source, pode não querer criar pacotes npm para todo o seu código reutilizável. (Também pode haver outras razões além da

proteção à propriedade intelectual: seus pacotes poderiam ser tão específicos da empresa ou do projeto que não faria sentido disponibilizá-los em um registro público).

Uma maneira de manipular isso é com os *registros privados do npm*. Atualmente eles oferecem os Orgs, que permitem publicar pacotes privados e fornecem aos desenvolvedores logins pagos, o que os autoriza a acessar esses pacotes. Consulte *npm* (<https://www.npmjs.com/products>) para obter mais informações sobre os Orgs e os pacotes privados do npm.

Middleware

Como vimos no decorrer deste livro, criar middleware não é algo grande, assustador e complicado: fizemos isso várias vezes no livro e, após algum tempo, você o fará sem nem mesmo pensar. Portanto, a próxima etapa é inserir middleware reutilizável em um pacote e incluí-lo em um registro npm.

Se você achar que seu middleware é muito específico de um projeto para ser inserido em um pacote reutilizável, deve considerar sua refatoração de modo a ser configurado para um uso mais geral. Lembre-se de que você pode passar objetos de configuração para o middleware para torná-lo mais útil em várias situações. Aqui está uma visão geral das maneiras mais comuns de expor middleware em um módulo Node. Todos os códigos a seguir supõem que você esteja usando esses módulos como um pacote, e esse pacote chama-se meadowlark-stuff.

O módulo expõe a função de middleware diretamente

Use esse método se seu middleware não precisar de um objeto de configuração:

```
module.exports = (req, res, next) => {  
  // seu middleware entra aqui...lembre-se de chamar next()  
  // ou next('route') a menos que seja esperado que esse  
  // middleware seja um endpoint  
  next()  
}
```

Para usar esse middleware:

```
const stuff = require('meadowlark-stuff')
```

```
app.use(stuff)
```

O módulo expõe uma função que retorna o middleware

Use esse método se seu middleware precisar de um objeto de configuração ou outras informações:

```
module.exports = config => {  
  // é comum criar o objeto config  
  // se ele não for passado:  
  if(!config) config = {}  
  
  return (req, res, next) => {  
    // seu middleware entra aqui...lembre-se de chamar next()  
    // ou next('route') a menos que seja esperado que esse  
    // middleware seja um endpoint  
    next()  
  }  
}
```

Para usar esse middleware:

```
const stuff = require('meadowlark-stuff')({ option: 'my choice' })  
  
app.use(stuff)
```

O módulo expõe um objeto que contém o middleware

Use essa opção se quiser expor vários middlewares relacionados:

```
module.exports = config => {  
  // é comum criar o objeto config  
  // se ele não for passado:  
  if(!config) config = {}  
  
  return {  
    m1: (req, res, next) => {  
      // seu middleware entra aqui...lembre-se de chamar next()  
      // ou next('route') a menos que seja esperado que esse middleware  
      // seja um endpoint  
      next()  
    },  
    m2: (req, res, next) => {
```

```
    next()
  },
}
```

Para usar esse middleware:

```
const stuff = require('meadowlark-stuff')({ option: 'my choice' })
```

```
app.use(stuff.m1)
```

```
app.use(stuff.m2)
```

Conclusão

Quando construímos um site, geralmente damos ênfase ao lançamento, e por uma boa razão: há muito entusiasmo envolvido no lançamento. No entanto, um cliente que ficar encantado com um site recém-lançado será rapidamente um cliente insatisfeito se não tomarmos cuidado com a manutenção do site. Tratar seu plano de manutenção com o mesmo cuidado que você tem com o lançamento de um site fornecerá o tipo de experiência que faz os clientes voltarem.

¹ Quando isso ocorreu, o termo *pós-parto* era muito radical. Agora chamamos de *retrospectiva*.

² Mike Wilson de *Fuel* tem essa regra prática: “Na terceira vez que você fizer algo, em vez disso automatize”.

Recursos adicionais

Neste livro, forneci para você uma visão geral abrangente da construção de sites com o Express. Abordamos bastante material, mas mesmo assim só examinamos superficialmente os pacotes, técnicas e frameworks que estão disponíveis. Neste capítulo, discutiremos aonde você pode ir para obter recursos adicionais.

Documentação online

Para a documentação de JavaScript, CSS e HTML, a *Mozilla Developer Network (MDN)* (<https://developer.mozilla.org>) é sem igual. Quando preciso de documentação JavaScript, procuro diretamente na MDN ou acrescento “mdn” à consulta. Caso contrário, inevitavelmente o w3schools aparece na busca. Quem quer que esteja gerenciando a SEO do w3schools é um gênio, mas recomendo evitar esse site: acho a documentação com frequência severamente deficiente.

Onde a MDN for uma ótima referência de HTML, se você for iniciante em HTML5 (ou mesmo se não for), deve ler *Dive Into HTML5* (<http://diveintohtml5.info>) de Mark Pilgrim. O WHATWG mantém uma excelente *especificação “living standard” do HTML5* (<http://developers.whatwg.org>); em geral é onde vou primeiro no caso de perguntas realmente difíceis de responder. Para concluir, as especificações oficiais do HTML e CSS estão localizadas no *site do W3C* (<http://www.w3.org>); são documentos crus e difíceis de ler, mas às vezes são os únicos recursos para os problemas mais complexos.

O JavaScript aderiu à *especificação de linguagem ECMA-262 ECMAScript* (http://bit.ly/ECMA-262_specs). Para rastrear a disponibilidade de recursos

JavaScript no Node (e vários navegadores), consulte o excelente *guia mantido por @kangax* (<http://bit.ly/36SoK53>).

A *documentação do Node* (<https://nodejs.org/en/docs>) é muito boa e abrangente e deve ser sua primeira opção de documentação autorizada sobre módulos Node (como o `http`, `https` e `fs`). A *documentação do Express* (<https://expressjs.com>) também é boa, mas não é tão abrangente como gostaríamos. A *documentação do npm* (<https://docs.npmjs.com/>) é abrangente e útil.

Periódicos

Há três periódicos gratuitos para os quais você deve se registrar e deve lê-los rigorosamente toda semana:

- *JavaScript Weekly* (<http://javascriptweekly.com>)
- *Node Weekly* (<http://nodeweekly.com>)
- *HTML5 Weekly* (<http://html5weekly.com>)

Esses três periódicos o manterão informado sobre as últimas notícias, serviços, blogs e tutoriais quando eles estiverem disponíveis.

Stack Overflow

Provavelmente você já usou o Stack Overflow (SO): desde seu início em 2008, ele se tornou o site online de perguntas e respostas predominante, e é o melhor recurso para termos nossas perguntas sobre JavaScript, Node e Express respondidas (e qualquer outra tecnologia abordada neste livro). O Stack Overflow é um site de P&R (Pergunta & Resposta) mantido por comunidade e baseado em reputação. O modelo de reputação é que é responsável pela qualidade do site e de seu contínuo sucesso. Os usuários podem ganhar reputação quando suas perguntas ou respostas recebem “upvotes” ou quando eles têm uma resposta aceita. Não é preciso ter reputação para fazer uma pergunta, e o registro é gratuito. No entanto, há algumas coisas que você pode fazer para aumentar as chances de sua pergunta ser respondida de maneira útil, o que discutiremos nesta seção.

A reputação é a moeda do Stack Overflow, e, embora haja pessoas que

queiram realmente ajudar, a possibilidade de ganhar reputação é a cereja do bolo que motiva boas respostas. Há muitas pessoas inteligentes no SO e estão todas competindo para fornecer a primeiro e/ou melhor resposta correta a uma pergunta (ainda bem que é fortemente desencorajado fornecer uma resposta rápida, porém insatisfatória). Veja o que você pode fazer para aumentar as chances de obter uma boa resposta à sua pergunta:

Seja um usuário informado do SO

Faça um passeio pelo SO (<http://bit.ly/2rFhSbb>) e leia “How do I ask a good question?” (<http://bit.ly/2p7Qnpw>). Se quiser, leia toda a documentação de ajuda (<http://bit.ly/36UnyOp>) – você ganhará um crachá se o fizer!

Não faça perguntas que já tenham sido respondidas

Seja aplicado, e tente descobrir se alguém já fez sua pergunta. Se você fizer uma pergunta que tiver uma resposta facilmente encontrável no SO, ela será rapidamente fechada como duplicata, e as pessoas lhe darão um downvote por isso, o que afetará negativamente sua reputação.

Não peça que escrevam seu código para você

Você perderá rapidamente pontos por sua pergunta e ela será fechada se perguntar simplesmente “Como faço X?”. A comunidade do SO espera que você faça um esforço para resolver seu problema antes de recorrer ao site. Descreva em sua pergunta o que tentou fazer e por que não está funcionando.

Faça uma pergunta de cada vez

Perguntas que incluem cinco itens – “como faço isso, depois aquilo, e também essas outras coisas e qual a melhor maneira de agir?” – são difíceis de responder e, portanto, desencorajadas.

Elabore um exemplo mínimo de seu problema

Respondo a muitas perguntas do SO, e as que pulo quase automaticamente são aquelas em que vejo três páginas de código (ou mais!). Simplesmente

pegar seu arquivo de 5.000 linhas e colá-lo em uma pergunta do SO não é uma boa maneira de ter sua pergunta respondida (mas as pessoas fazem isso o tempo todo). É uma abordagem preguiçosa que com frequência não é recompensada. Além de ser menos provável que você obtenha uma resposta útil, o processo de eliminar coisas que *não estão* causando o problema pode levar a que você mesmo resolva o problema (então não precisará fazer a pergunta no SO). Elaborar um exemplo mínimo será bom para suas habilidades de depuração e raciocínio crítico, e o tornará um bom usuário do SO.

Aprenda o Markdown

O Stack Overflow usa o Markdown para a formatação de perguntas e respostas. Uma pergunta bem formatada tem mais chances de ser respondida, logo, você deve reservar um tempo para aprender essa útil e onipresente *linguagem de marcação* (<http://bit.ly/2CB1L0a>).

Aceite e dê upvotes às respostas

Se alguém responder sua pergunta satisfatoriamente, você deve dar um upvote para a resposta e aceitá-la; isso aumenta a reputação de quem respondeu, e é a reputação que guia o SO. Se várias pessoas fornecerem repostas aceitáveis, selecione a que achar melhor e aceite-a, e dê um upvote para quem achar que ofereceu uma resposta útil.

Se você resolver seu problema antes de outra pessoa, responda a sua própria pergunta

O SO é um recurso de comunidade: se você tiver um problema, há chances de que outras pessoas também o tenham. Se conseguir resolvê-lo, responda a sua própria pergunta para ajudar os outros.

Se quiser ajudar a comunidade, considere responder você mesmo às perguntas: é divertido e gratificante, e pode levar a benefícios mais tangíveis que um score de reputação arbitrário. Se houver uma pergunta para a qual você não receber respostas úteis durante dois dias, forneça uma *recompensa* pela resposta, usando a própria reputação. Ela será removida de sua conta imediatamente, e não será restituída. Se alguém responder à pergunta

satisfatoriamente, e você aceitar a resposta, essa pessoa receberá a recompensa. É claro que você precisa ter reputação para pagar uma recompensa: a recompensa mínima é de 50 pontos de reputação. Embora você possa obter reputação fazendo perguntas de qualidade, geralmente é mais rápido obtê-la fornecendo respostas de qualidade.

Responder perguntas de outras pessoas também tem o benefício de ser uma ótima maneira de aprender. Geralmente percebo que aprendo mais respondendo a perguntas dos outros do que tendo minhas perguntas respondidas. Se quiser conhecer totalmente uma tecnologia, aprenda os aspectos básicos e depois comece a responder a perguntas de outras pessoas no SO. Inicialmente você será superado por quem já é especialista, mas não demorará a perceber que *se tornou* um deles.

Para concluir, não hesite em usar sua reputação para favorecer sua carreira. É válido colocar uma boa reputação em um currículo. Funcionou para mim e, agora que estou em posição de entrevistar pessoas, sempre me impressiona ver uma boa reputação no SO (considero uma “boa” reputação no SO uma pontuação acima de 3.000; reputações com cinco dígitos são *ótimas*). Uma boa reputação no SO informa que alguém é não só competente em sua área, mas também tem boa capacidade de comunicação e geralmente é útil.

Contribuindo para o Express

O Express e o Connect são projetos open source, logo, qualquer pessoa pode enviar *requisições pull* (terminologia do GitHub para as alterações enviadas que gostaríamos que incluíssem no projeto). Não é fácil fazer isso: os desenvolvedores são profissionais e a autoridade máxima em seus projetos. Não estou desencorajando a contribuição, apenas dizendo que você tem de dedicar um esforço significativo para ser um colaborador bem-sucedido, e não pode tratar os envios levianamente.

O processo de contribuição está bem documentado na *home page do Express* (<http://bit.ly/2q7WD0X>). Os mecanismos envolvem fazer um fork do projeto em sua conta do GitHub, clonar esse fork, fazer suas alterações, inseri-las no GitHub e criar uma requisição pull (PR, pull request), que será revisada por uma ou mais pessoas do projeto. Se nossos envios forem pequenos ou forem

correções de bugs, podemos ser bem-sucedidos simplesmente enviando a requisição pull. Se você estiver tentando fazer algo maior, deve se comunicar com um dos principais desenvolvedores e discutir sua contribuição. Não queremos perder horas ou dias em um recurso complicado para acabar descobrindo que ele não se encaixa na visão do responsável pela manutenção ou já está sendo manipulado por outra pessoa.

A outra maneira de contribuir (indiretamente) com o desenvolvimento do Express e do Connect é publicando pacotes npm – especificamente, middleware. A publicação de seu próprio middleware não demandará a aprovação de ninguém, mas isso não significa que você pode encher o registro do npm com middleware de baixa qualidade. Planeje, teste, implemente e documente, e seu middleware será mais bem-sucedido.

Se você publicar os próprios pacotes, o mínimo que deve fornecer é:

Nome do pacote

Embora o nome do pacote seja o que você quiser, é claro que é preciso selecionar algo que ainda não tenha sido usado, o que às vezes é um desafio. Atualmente os pacotes npm dão suporte ao namespacing por conta, logo, você não precisa competir globalmente pelos nomes. Se estiver criando middleware, é costumeiro prefixar o nome do pacote com connect- ou express-. Nomes de pacotes fáceis de lembrar que não tenham nenhuma relação específica com o que eles fazem são aceitáveis, mas seria ainda melhor um nome que sugerisse o que o pacote faz (um bom exemplo de nome de pacote fácil, mas apropriado, é zombie, para a emulação de navegador headless).

Descrição do pacote

A descrição de seu pacote deve ser curta, concisa e explicativa. Esse é um dos principais campos que são indexados quando as pessoas procuram pacotes, logo, é melhor que seja descritivo em vez de inteligente (há espaço para a inteligência e o humor em sua documentação, não se preocupe).

Autor/colaboradores

Ganhe algum crédito. Não se acanhe.

Licença(s)

Esse item costuma ser negligenciado, e não há nada mais frustrante que encontrar um pacote sem licença (deixando-nos inseguros quanto a se podemos usá-lo em um projeto). Não seja essa pessoa. A *licença MIT* (http://bit.ly/mit_license) é uma opção fácil se você não quiser restrições a como seu código será usado. Se você quiser ser open source (e permanecer assim), outra opção popular é a *licença GPL* (http://bit.ly/gpl_license). Também é importante incluir os arquivos de licença no diretório raiz de seu projeto (eles devem começar com *LICENSE*). Para ter cobertura máxima, use uma licença dupla MIT e GPL. Para ver um exemplo disso em *package.json* e em arquivos *LICENSE*, consulte meu *pacote* connect-bundle (<http://bit.ly/connectbundle>).

Versão

Para o sistema de versionamento funcionar, você precisa dar uma versão a seus pacotes. Observe que o versionamento do npm fica separado dos números de commits no repositório: você pode atualizar o repositório como quiser, mas ele não mudará o que as pessoas verão quando usarem o npm para instalar seu pacote. É preciso incrementar o número de versão e fazer a republicação para as alterações serem refletidas no registro do npm.

Dependências

Você deve se esforçar para ser conservador quanto às dependências em seus pacotes. Não estou sugerindo reinventar constantemente a roda, mas as dependências aumentam o tamanho e a complexidade do licenciamento do pacote. No mínimo, você deve se certificar de não listar dependências das quais não precisa.

Palavras-chave

Junto com a descrição, as palavras-chave são o outro metadado importante usado para as pessoas que estiverem tentando encontrar seu pacote, logo, selecione palavras-chave apropriadas.

Repositório

Você deve ter um. O GitHub é o mais comum, mas outros são bem-vindos.

README.md

O formato de documentação padrão tanto para o GitHub quanto para o npm é o *Markdown* (<http://bit.ly/33IxnwS>). É uma sintaxe fácil semelhante à dos wikis que você pode aprender rapidamente. Uma documentação de qualidade será muito importante se você quiser que seu pacote seja usado. Quando acesso uma página do npm e não há documentação, geralmente apenas saio dela sem uma investigação mais detalhada. No mínimo, você deve descrever o uso básico (com exemplos). Melhor ainda é todas as opções estarem documentadas. Descrever como executar testes já é mais do que o esperado.

Quando você estiver pronto para publicar seu pacote, o processo será muito fácil. Registre-se para obter uma *conta gratuita do npm* (<https://npmjs.org/signup>) e siga estas etapas:

1. Digite `npm adduser` e faça login com suas credenciais do npm.
2. Digite `npm publish` para publicar seu pacote.

Isso é tudo! Provavelmente você vai querer criar um projeto a partir do zero, e testar seu pacote usando `npm install`.

Conclusão

Espero sinceramente que este livro tenha fornecido todas as ferramentas necessárias para você começar a usar esse empolgante stack de tecnologias. Em nenhum momento de minha carreira me senti tão revigorado por uma nova tecnologia (apesar do participante ímpar e essencial que é o JavaScript) e espero ter conseguido transmitir parte da elegância e da promessa desse stack. Embora eu construa sites profissionalmente há muitos anos, acho que, graças ao Node e ao Express, entendo a maneira como a internet funciona em um nível mais profundo do que jamais havia vivenciado. Acho que essa é uma tecnologia que melhora realmente a compreensão, em vez de tentar ocultar os detalhes, fornecendo ao mesmo tempo um framework para a construção rápida e eficiente de sites.

Seja você um iniciante no desenvolvimento web, ou apenas no Node e no Express, dou-lhe as boas-vindas às fileiras de desenvolvedores JavaScript. Espero ansiosamente o ver nos grupos de usuários e conferências e, o mais importante, ver o que você construirá.

Sobre o autor

Ethan Brown é diretor de tecnologia na VMS, onde é responsável pela arquitetura e implementação do VMSPRO, software baseado em nuvem para suporte a decisões, análise de risco e concepção criativa para grandes projetos. Com mais de 20 anos de experiência em programação web, Ethan abraçou a stack JavaScript como a plataforma web do futuro.

Colofão

Os animais da capa de *Programação web com Node e Express* são uma cotovia negra (*Melanocorypha yeltoniensis*) e uma cotovia de asa branca (*Melanocorypha leucopter*). Os dois pássaros são parcialmente migratórios e são conhecidos por se distanciar de seu habitat mais adequado nas estepes do Cazaquistão e da região central da Rússia. Além de reproduzirem-se nesses locais, as cotovias negras macho também passam o inverno nas estepes do Cazaquistão, enquanto as fêmeas migram para o sul. As cotovias de asa branca, por outro lado, voam para o oeste e o norte para além do Mar Negro durante os meses de inverno. O alcance global desses pássaros vai ainda mais longe: a Europa constitui de um quarto à metade do alcance global da cotovia de asa branca e só de cinco por cento a um quarto do alcance global da cotovia negra.

As cotovias negras têm esse nome pela coloração negra que cobre quase o corpo inteiro dos machos da espécie. As fêmeas, ao contrário, lembram a coloração do macho somente em suas pernas negras e nas penas negras de suas asas posteriores. Uma combinação de cinza-escuro e claro cobre o resto da fêmea.

As cotovias de asa branca possuem um padrão distintivo de penas das asas nas cores preta, branca e castanha. Faixas cinza descendo as costas complementam a parte inferior do corpo em uma cor branca pálida. Os machos diferem das fêmeas da espécie somente em suas coroas castanhas.

Tanto as cotovias negras quanto as de asa branca revelam o distintivo chamado melodioso que tornou as cotovias de todos os tipos admiradas nas imaginações de escritores e músicos por séculos. Os dois pássaros comem insetos e sementes quando adultos e fazem ninhos no solo. As cotovias negras foram vistas levando esterco para seus ninhos para construir paredes ou assentar algum tipo de pavimento, embora a causa desse comportamento não tenha sido identificada.

Muitos dos animais das capas da O'Reilly estão em perigo; todos eles são importantes para o mundo.

A ilustração da capa é de Karen Montgomery e foi baseada em uma gravura em preto e branco do livro *The Royal Natural History* de Lydekker.




Design de Microsserviços com Django

Uma visão geral das ferramentas e práticas

—
Akos Hochrein

novatec

apress®



Design de Microsserviços com Django

Hochrein, Akos

9786586057072

144 páginas

[Compre agora e leia](#)

Explore os microsserviços usando o framework Django baseado em Python e analise suas vantagens e desvantagens. Este livro descreverá os microsserviços, como eles conversam entre si e como são criados usando a linguagem de programação Python e o framework web Django.

Começaremos compreendendo quais são as principais diferenças entre os microsserviços e as arquiteturas monolíticas. Em seguida, o livro explora com minúcias o modo como os microsserviços são criados e quais são os modelos comuns que surgiram em nosso mercado. Também veremos com detalhes os padrões de comunicação e de responsabilidades, além de analisar as metodologias que agilizarão a evolução de sua arquitetura, escrevendo uma quantidade menor de código, porém distribuído, e utilizando a linguagem de programação Python e o framework web Django. No final do livro, você terá uma sólida compreensão das arquiteturas dos microsserviços. Com um conjunto de ferramentas abrangente e robusto em mãos, poderá começar a trabalhar em direção a sistemas mais escaláveis, resilientes e possíveis de manter. O que você verá: - entenda as vantagens e as desvantagens da adoção de microsserviços; - faça o design de sistemas e arquiteturas com vistas à resiliência e à distribuição de responsabilidades; - trabalhe com ferramentas para escalar os sistemas distribuídos, tanto na

dimensão técnica como na dimensão organizacional; - analise o que há de essencial no framework web Django.

[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](#)

Fundos de Investimento Imobiliário

ASPECTOS GERAIS E
PRINCÍPIOS DE ANÁLISE

novatec

Roni Antônio Mendes

Fundos de Investimento Imobiliário

Mendes, Roni Antônio

9788575226766

256 páginas

[Compre agora e leia](#)

Você sabia que o investimento em imóveis é um dos preferidos dos brasileiros? Você também gostaria de investir em imóveis, mas tem pouco dinheiro? Saiba que é possível, mesmo com poucos recursos, investir no mercado de imóveis por meio dos Fundos de Investimento Imobiliário (FIIs). Investir em FIIs representa uma excelente alternativa para aumentar o patrimônio no longo prazo. Além disso, eles são ótimos ativos geradores de renda que pode ser usada para complementar a aposentadoria. Infelizmente, no Brasil, os FIIs são pouco conhecidos. Pouco mais de 100 mil pessoas investem nesses ativos. Lendo este livro, você aprenderá os aspectos gerais dos FIIs: o que são; as vantagens que oferecem; os riscos que possuem; os diversos tipos de FIIs que existem no mercado e como proceder para investir bem e com segurança. Você também aprenderá os princípios básicos para avaliá-los, inclusive empregando um método poderoso, utilizado por investidores do mundo inteiro: o método do Fluxo de Caixa Descontado (FCD). Alguns exemplos reais de FIIs foram estudados neste livro e os resultados são apresentados de maneira clara e didática, para que você aprenda a conduzir os próprios estudos e tirar as próprias conclusões. Também são apresentados conceitos gerais de como montar e gerenciar uma carteira de investimentos. Aprenda a investir em FIIs. Leia este livro.

[Compre agora e leia](#)