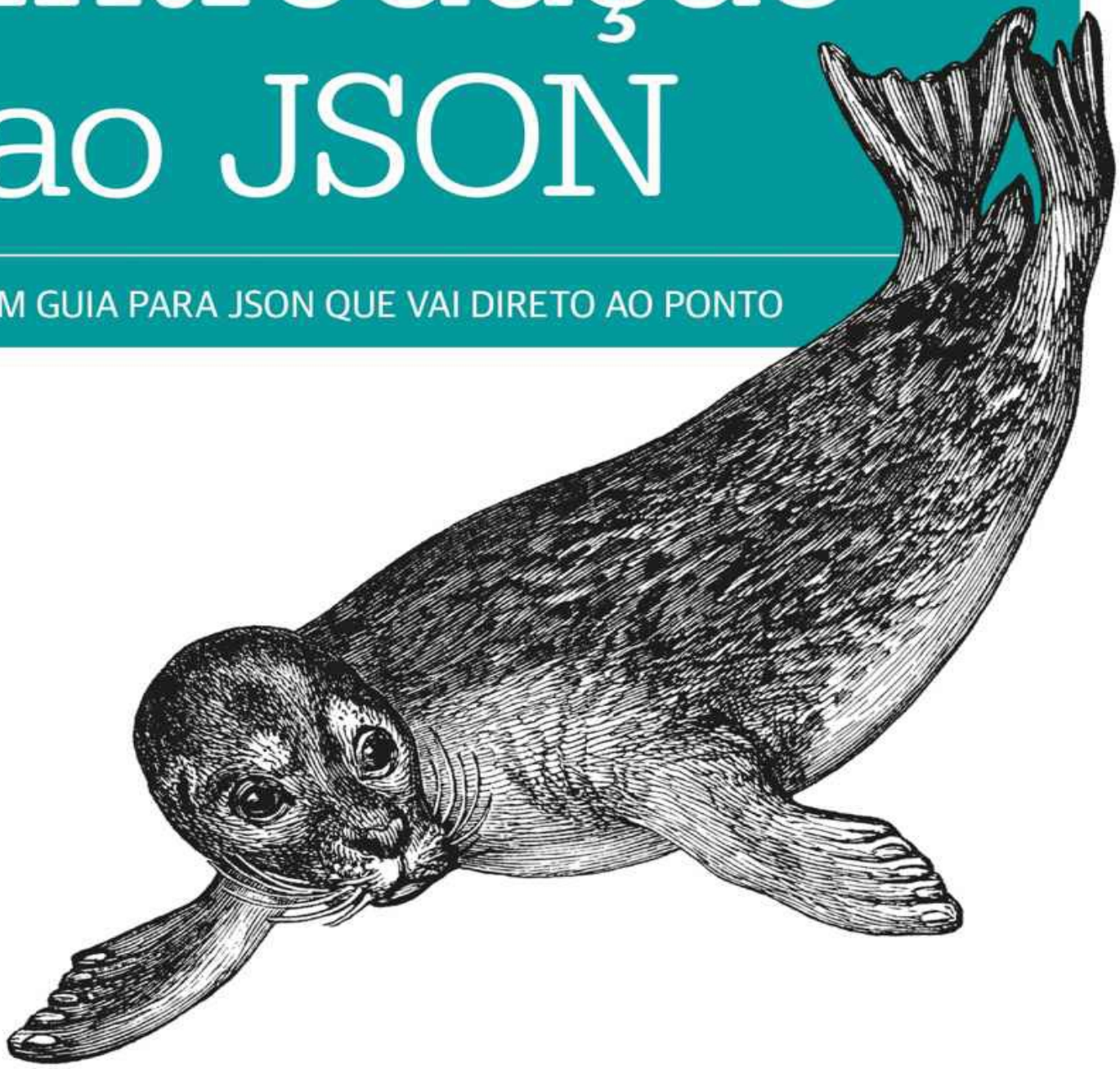
O'REILLY®

Introdução ao JSON

UM GUIA PARA JSON QUE VAI DIRETO AO PONTO



novatec

Lindsay Bassett

Introdução ao JSON

UM GUIA PARA JSON QUE VAI DIRETO AO PONTO

Lindsay Bassett

Novatec
São Paulo | 2019

Tradução em português autorizada da edição em inglês da obra Introduction to JavaScript Object Notation, ISBN 9781491929483 © 2015 Lindsay Bassett. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., que detém todos os direitos para publicação e venda desta obra.

Editor: Rubens Prates
Assistente editorial: Priscila A. Yoshimatsu
Tradução: Lúcia A. Kinoshita
Revisão gramatical: Milena Leal
Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-745-9

Histórico de edições impressas:

Setembro/2015 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Sumário

Prefácio

Capítulo 1 ■ O que é JSON?

JSON é um formato para intercâmbio de dados
O JSON é independente da linguagem de programação
Termos e conceitos essenciais

Capítulo 2 ■ Sintaxe do JSON

JSON é baseado nos objetos literais do JavaScript
Pares nome-valor
Sintaxe apropriada do JSON
Validação de sintaxe
JSON como um documento
Tipo de mídia JSON
Termos e conceitos essenciais

Capítulo 3 ■ Tipos de dados JSON

Uma olhada rápida nos tipos de dados
Tipos de dados JSON
Tipo de dado objeto do JSON
Tipo de dado string do JSON
Tipo de dado numérico do JSON
Tipo de dado booleano do JSON
Tipo de dado null do JSON
Tipo de dado array do JSON
Termos e conceitos essenciais

Capítulo 4 ■ JSON Schema

Contratos com validação mágica
Introdução ao JSON Schema
Termos e conceitos essenciais

Capítulo 5 ■ Preocupações relacionadas à segurança com JSON

Uma olhada rápida nos relacionamentos entre o lado cliente e o lado servidor
Cross-Site Request Forgery (CSRF)
Ataques de injeção de código

Cross-site Scripting (XSS)
Brechas de segurança: decisões relacionadas à arquitetura
Termos e conceitos essenciais
Capítulo 6 ■ XMLHttpRequest do JavaScript e APIs web
APIs web
XMLHttpRequest do JavaScript
Problemas de relacionamento e regras sobre compartilhamento
Cross-Origin Resource Sharing (CORS)
JSON-P
Termos e conceitos essenciais
Capítulo 7 ■ JSON e os frameworks do lado cliente
jQuery e JSON
AngularJS
Termos e conceitos essenciais
Capítulo 8 ■ JSON e NoSQL
O banco de dados CouchDB
API do CouchDB
Termos e conceitos essenciais
Capítulo 9 ■ JSON no lado servidor
Serializando, desserializando e solicitando JSON
ASP.NET
PHP
Uma infinidade de solicitações HTTP para JSON
Ruby on Rails
Node.js
Java
Termos e conceitos essenciais
Capítulo 10 ■ Conclusão
JSON como um arquivo de configuração
Quadro geral
Sobre a autora
Colofão

Prefácio

Uma lição que aprendi durante minha jornada como desenvolvedora web é que, não importa o quão devotada em espírito eu seja aos temas relacionados a esse assunto, encontrar tempo para essa devoção é outra questão. O mundo da tecnologia rapidamente em evolução não está preocupado com o quanto as pessoas estão ocupadas. Ele não diz: “Relaxe, use o seu tempo para estudar; eu entendo que você tem uma família e é difícil conseguir um momento tranquilo para estudar”. O que ele diz é: “Continue evoluindo; caso contrário, você ficará obsoleto”.

Escrevi este livro com esse sentimento em mente. Este livro é curto – ele evita intencionalmente assuntos como “a história do JSON”.

Embora eu seja grata a Douglas Crockford por ter criado o JSON, não faço nenhuma menção a ele nem aos anos em que o JSON evoluiu até se tornar o que é hoje. Este livro diz respeito ao JSON *como ele é hoje*. Se quiser ler a história do JSON, a Wikipedia tem uma síntese excelente (*https://en.wikipedia.org/wiki/JSON#History*).

Este livro tem como objetivo ir direto ao coração do JSON. É sobre ir direto ao ponto e fazer com que você fique rapidamente preparado para trabalhar. É para os profissionais ocupados de TI.

Público-alvo

Enquanto escrevia este livro para o profissional ocupado de TI, também pensei um pouco em quem é você. Você pode ser um desenvolvedor web iniciante de frontend. Quem sabe você esteja focado em desenvolvimento de aplicações web do lado servidor há anos e agora precisa aprender JSON para criar uma API web. Você pode ser um desenvolvedor de PHP, Ruby, C, Java ou ASP.NET. A lista não para por aí. Muitas pessoas diferentes em funções distintas querem e precisam conhecer o JSON.

Neste livro, evito o uso excessivo de jargões e explico os conceitos que são pré-requisitos para aqueles que são iniciantes em programação web. Procuro falar com todos vocês. Entretanto, devo fazer algumas pressuposições sobre o conhecimento que você já tem. Se o desenvolvimento web é uma novidade para você, este não deverá ser o primeiro livro a ser escolhido.

Suponho que você tenha pelo menos um entendimento básico de:

HTML

Você compreende o propósito do HTML e é capaz de reconhecer a estrutura e, no mínimo, algumas das tags em um documento HTML.

JavaScript

Você entende o propósito do JavaScript e sabe o que são uma tag <script>, uma função e uma variável. Não há problema algum em ser apenas um iniciante. Manterei meus exemplos de código simples.

Conceitos de programação

Fornecerei algumas explicações rápidas para conceitos como objeto e array para aqueles que são iniciantes e que possam precisar de uma revisão. No entanto, se você ainda não trabalhou com *nenhuma* linguagem de programação, este livro não é o ideal para começar.

Abordagem ao JSON

Inúmeras vezes, ao longo dos anos, precisei aprender novas tecnologias, em geral, durante um projeto com um prazo final. Compro livros grandes, vasculho tutoriais e tento absorver informações suficientes para saber o que estou fazendo. Enquanto vasculho centenas de páginas, procuro respostas a estas três perguntas básicas:

- O que é isso?
- Como posso usar isso?
- Como as pessoas mal-intencionadas poderão usar isso?

Quando escrevi este livro, procurei ir ao cerne dessas questões para que você não precisasse vasculhar muitas informações para encontrar as respostas.

Nos capítulos de 1 a 4, exploraremos o JSON em um nível mais específico. Inicialmente, responderei à questão definitiva relacionada a “O que é isso?”. A partir daí, daremos uma olhada na sintaxe, na validação da sintaxe, nos tipos de dados e na validação de esquema.

No capítulo 5, daremos uma olhada nas preocupações com segurança – um assunto importante. Esse capítulo inclui uma introdução aos conceitos de lado cliente e lado servidor que são importantes para o restante do livro. Esse capítulo responderá à pergunta “Como as pessoas mal-intencionadas poderão usar isso?”.

Os capítulos restantes analisarão as diversas funções do JSON como formato para intercâmbio de dados. Têm como objetivo responder à pergunta “Como posso usar isso?”.

Esses capítulos incluem vários exemplos de JSON e de tecnologias que interagem com ele. Alguns aspectos importantes a serem observados nos capítulos de 6 a 9 são:

Tecnologias

Discutirei diversas tecnologias, por exemplo, jQuery, AngularJS e CouchDB, de modo geral. Cada um desses assuntos é tão amplo que livros inteiros poderiam ser escritos (e foram) sobre eles. Deixei propositalmente de lado as instruções sobre instalação e explicações mais detalhadas sobre as tecnologias em si. A questão é expor você ao *modo como* essas tecnologias usam o JSON. Se quiser testar os exemplos apresentados com essas tecnologias, será necessário realizar algumas tarefas adicionais para configurar todo o ambiente. No entanto, os exemplos são simples. Se tudo estiver funcionando no nível básico, você deverá ser capaz de realizar alguns experimentos com eles.

Exemplos de código

Você verá muitos exemplos de código neste livro, alguns em linguagens de programação que poderão ser novas para você. A sintaxe dessas linguagens não será explicada. Não fique preocupado se você não compreender a sintaxe. O ponto importante é que você “entenda a essência” do que o código faz e a explicação para isso será fornecida.

Todos os exemplos de código usados neste livro estão disponíveis no repositório do GitHub para este livro (*https://github.com/lindsaybassett/json*).

A questão principal dos capítulos de 6 a 9 é expor você ao modo como o JSON é usado no mundo atualmente e fazer com que as ideias brotem em sua mente para serem usadas em seus próprios projetos. Se você nunca viu o JSON sendo usado como um documento em um banco de dados para o armazenamento de documentos, você pensaria em usá-lo em um projeto? Conhecer representa metade da batalha.

Em cada capítulo, procuramos prover um equilíbrio entre chegar ao cerne da questão e fornecer informações suficientes para que não reste nenhuma grande lacuna em sua educação. Em última instância, o livro todo está estruturado para que você esteja preparado para usar o JSON rapidamente sem sacrificar um entendimento mais profundo do que é o JSON e quais são os seus propósitos.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

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

Largura constante

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

Largura constante em negrito

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

Largura constante em itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário

ou determinados pelo contexto.

 Esse 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/lindsaybassett/json*.

Este livro está aqui para ajudá-lo a fazer seu trabalho. De modo geral, se este livro incluir exemplos de código, você poderá usar o código em seus programas e em sua documentação. Você não precisa nos contatar para pedir permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use diversas partes de código deste livro não requer permissão. Porém, vender ou distribuir um CD-ROM de exemplos de livros da O’Reilly requer permissão. Responder a uma pergunta mencionando este livro e citar o código de exemplo não requer permissão. Em contrapartida, incluir uma quantidade significativa de código de exemplos deste livro na documentação de seu produto requer permissão.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: *Introduction to JavaScript Object Notation*, de Lindsay Bassett (O’Reilly). Copyright 2015 Lindsay Bassett, 978-1-491-92948-3.

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

Como entrar em contato conosco

Envie seus comentários e questões 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

  http://www.novatec.com.br/catalogo/7522451-intro-json

- Página da edição original em inglês

  http://bit.ly/intro_JS_object_notation

Para obter mais informações sobre os livros da Novatec, acesse nosso site em: *http://www.novatec.com.br*.

Agradecimentos

Inicialmente, gostaria de agradecer ao meu marido Rhett por seu incentivo, pelo apoio em todos os meus empreendimentos de escrita e por ter sido compreensivo durante todos aqueles dias que passei escondida no escuro, debruçada sobre o meu laptop.

Gostaria de agradecer também a Douglas Crockford por ter criado o JSON e por ter me proporcionado um assunto divertido sobre o qual escrever. Sou muito grata aos revisores técnicos Shelley Powers e Tom Marrs por seus feedbacks construtivos que ajudaram a melhorar este livro. Qualquer erro deve-se exclusivamente a mim.

Foi uma verdadeira satisfação trabalhar com minha editora Meg Foley.

Por fim, gostaria de agradecer à O’Reilly por ter dado um lar ao meu livro e por tê-lo disponibilizado ao mundo. Sempre fui uma grande fã dos livros da O’Reilly e, após ter trabalhado com essa editora nesse ano, passei a ser mais fã ainda.

CAPÍTULO 1

O que é JSON?

Antes de darmos uma olhada no JSON a partir de um ponto de vista mais específico, vamos observá-lo a uma altura de aproximadamente dois mil metros. Do topo da montanha, podemos ver o JSON circulando pelo mundo, transportando dados em seu formato simples. Se observarmos o JSON através de nossos binóculos, veremos seus dados entre diversos caracteres de chaves ({}). No entanto, se dermos um passo para trás e observarmos *como* ele está sendo usado, veremos que, em última instância, ele é um formato para intercâmbio de dados.

JSON é um formato para intercâmbio de dados

Um formato para intercâmbio de dados é um formato-texto usado para trocar dados entre plataformas. Outro formato para intercâmbio de dados de que talvez você já tenha ouvido falar é o XML. O mundo precisa dos formatos para intercâmbio de dados como o XML e o JSON para trocar dados entre sistemas bem diferentes.

Pare um instante e imagine um mundo composto de centenas de pequenas ilhas isoladas em um vasto oceano. Cada ilha tem sua própria linguagem e os seus costumes. Todas as ilhas têm comerciantes marítimos que viajam distâncias enormes entre as ilhas. O comércio exterior é parte integrante da economia de todas as ilhas e contribui para o alto padrão de vida dos habitantes das ilhas. Se não fossem as gaivotas transportadoras altamente treinadas, isso não seria possível.

As gaivotas transportadoras deslocam-se de uma ilha para outra carregando uma folha de papel contendo dados de um relatório sobre as mercadorias com mais demanda. Dessa maneira, os comerciantes sabem para onde devem ir

em seguida e quais mercadorias devem adquirir antes de embarcar em suas longas viagens pelos oceanos. Esses dados importantes permitem que todos os habitantes das ilhas prosperem sem a ameaça da falta de produtos.

Tenha em mente que cada ilha fala uma língua diferente. Se os dados fossem transmitidos em diversas línguas, cada ilha teria de investir em pesquisadores para aprender todos os idiomas do mundo e empregar uma equipe de tradutores. Esse processo seria caro e consumiria muito tempo. No entanto, esse é um mundo inteligente, de modo que todas as ilhas concordaram em definir uma única linguagem com um formato padrão para transmitir seus dados comerciais. Cada ilha emprega apenas um tradutor, que entende o formato único dos dados dos relatórios comerciais trazidos pelas gaivotas transportadoras.

O mundo real da tecnologia é muito semelhante ao exemplo do mundo de ilhas imaginárias. Há um vasto oceano cheio de ilhas com linguagens, costumes e arquiteturas diferentes. A capacidade desses sistemas únicos de se comunicarem é essencial para muitos negócios e organizações. Se cada um desses sistemas precisasse de um tradutor para todas as diversas maneiras com as quais os outros sistemas estruturam seus dados, as comunicações consumiriam uma quantidade absurda de tempo e de recursos. Em vez disso, os sistemas concordam em usar um único formato de dados e empregam somente um tradutor.

O JSON é um formato para intercâmbio de dados com o qual muitos sistemas concordaram em usar na comunicação de dados. Você pode ter ouvido falar do JSON como um “formato para intercâmbio de dados” ou simplesmente como um “formato de dados”. Neste livro, vou me referir ao JSON como um formato para intercâmbio de dados, pois a definição de “intercâmbio” nos lembra que o propósito do formato de dados é ser trocado entre duas ou mais entidades.

Muitos sistemas, mas nem todos, concordaram em usar o JSON para a comunicação de dados. Há formatos para intercâmbio de dados como o XML (Extensible Markup Language, ou Linguagem de Marcação Extensível) que surgiram antes de o JSON sequer ter sido imaginado. O mundo real não é tão simples quanto o exemplo das ilhas. Muitos sistemas têm e continuam usando

outros formatos, por exemplo, o XML, ou formatos mais tabulares e delimitados como o CSV (Comma-Separated Values, ou Valores Separados por Vírgula). A decisão de cada ilha no mundo real para o formato de dados que será aceito na comunicação geralmente diz respeito ao modo como o formato de dados está relacionado aos costumes, à linguagem e à arquitetura da ilha.

No exemplo do mundo de ilhas, cada uma das centenas de ilhas tinha a sua própria língua. Os dados do relatório em papel transportado pelas gaivotas estavam em um formato independente do idioma, sobre o qual houve um consenso. Dessa maneira, um único tradutor para os dados dos relatórios comerciais poderá ser empregado em cada ilha. O mesmo vale para o JSON, exceto pelo fato de os dados serem transportados em redes na forma de zeros e uns, e não por gaivotas. O tradutor não é um ser humano: é um parser empregado pelo sistema que consome os dados para que eles possam ser lidos pelo sistema em que estão entrando.

O JSON é independente da linguagem de programação

JSON significa JavaScript Object Notation (Notação de Objetos JavaScript). O nome desse formato para intercâmbio de dados pode confundir as pessoas, fazendo-as achar que elas devam aprender JavaScript para entender e usar o JSON. Aprender JavaScript antes de conhecer o JSON pode ter o seu valor, pois o JSON foi criado a partir de um subconjunto do JavaScript; porém, se você não pretende usar o JavaScript em breve, isso não será necessário. Você poderá dedicar-se à linguagem ou às linguagens usadas em sua própria ilha, pois a essência de um formato para intercâmbio de dados é ser independente da linguagem.

O JSON é baseado nos objetos literais do JavaScript. Uma explicação detalhada sobre o “como” disso será mais adequado em nossa discussão sobre sintaxe (Capítulo 2) e sobre tipos de dados (Capítulo 3). Neste capítulo, o “porquê” é importante. Se um formato para intercâmbio de dados foi concebido para ser independente da linguagem, poderá parecer contraditório ter um formato de dados que não só tenha sido derivado de uma única linguagem, mas que também a apresenta em seu nome: JavaScript Object

Notation. Então, por que isso acontece?

Se retornarmos ao exemplo das ilhas, imagine, por um instante, como teria sido a reunião para selecionar o formato de dados. Quando os representantes de cada uma das centenas de ilhas chegaram a essa reunião e tentaram criar um único formato de dados, a primeira tarefa deles deve ter sido encontrar um terreno comum.

As linguagens de cada ilha podiam ser únicas, porém deve ter havido aspectos que os habitantes das ilhas identificaram como sendo comuns. A maioria das linguagens era falada principalmente com a voz humana e incluía uma forma escrita da linguagem, representada por algum tipo de caractere. Além disso, as expressões faciais e os movimentos das mãos também estavam presentes. Havia algumas ilhas problemáticas, em que as pessoas se comunicavam por outros meios, por exemplo, batendo varetas ou piscando, porém a maior parte das ilhas encontrou um terreno comum nas formas escrita e falada da língua.

No mundo real, há centenas de linguagens de programação. Algumas são mais populares e mais comumente usadas do que outras, porém o terreno das linguagens é diversificado. Quando os universitários se formam em ciência da computação preparando-se para uma carreira em programação, eles não estudam todas as linguagens de programação. Os alunos normalmente começam com uma linguagem e a linguagem em si não é tão importante quanto aprender os conceitos de programação universalmente aceitos. Depois que os alunos compreendem esses conceitos, eles podem aprender outras linguagens de programação com mais facilidade usando suas habilidades em reconhecer os recursos e as funcionalidades comuns.

Se separarmos a palavra “JavaScript” do nome “JavaScript Object Notation”, ficaremos com “Object Notation” (Notação de objetos). Com efeito, vamos esquecer totalmente o JavaScript. Poderíamos então dizer que estamos usando um formato de notação de objetos para intercâmbio de dados. “Objeto” é um conceito comum em programação, particularmente na POO (Programação Orientada a Objetos). A maioria dos alunos de ciência da computação que estuda programação aprenderá o conceito de objetos.

Sem mergulhar em uma explicação sobre os objetos, vamos voltar a nossa

atenção para a palavra “Notação”. *Notação* implica um sistema de caracteres para representar dados, por exemplo, números ou palavras. Com ou sem um entendimento sobre objetos em programação, não é difícil perceber a importância de ter uma notação para descrever algo que seja comum entre as linguagens de programação.

Retornando novamente ao exemplo das ilhas, os próprios habitantes das ilhas definiram uma notação que representava um laço comum entre a maioria das linguagens. A maior parte dos habitantes das ilhas tinha uma maneira semelhante de representar números com uma forma de contagem, e eles concordaram que poderiam entender uma série de símbolos para representar objetos do mundo real, por exemplo, trigo ou tecido. Até a ilha que se comunicava por piscadas achou esse formato aceitável.

Apesar de haver consenso entre a grande maioria das ilhas, ainda havia algumas ilhas, por exemplo, a ilha que se comunicava batendo varetas, que não achavam o formato compreensível. Um bom formato para intercâmbio de dados abrange a maioria, porém, geralmente, há aqueles que ficam de fora. Quando falamos dessa abrangência, o termo geralmente visto por aí é *portabilidade*. A portabilidade, ou seja, a compatibilidade na transferência de informações entre plataformas e sistemas, é o objetivo principal de um formato para intercâmbio de dados.

Retornando à notação, a notação do JSON pode ter se originado do JavaScript, porém a notação em si é a parte importante. O JSON não só é independente de linguagem, como também representa dados de uma maneira que faz alusão a elementos comuns em diversas linguagens de programação. Pelo modo como os dados são representados, por exemplo, como números e palavras, até mesmo as linguagens de programação que não sejam orientadas a objetos podem achar esse formato aceitável.

Termos e conceitos essenciais

Este capítulo discutiu os seguintes termos essenciais:

JSON

JavaScript Object Notation (Notação de Objetos JavaScript).

Notação

É um sistema de caracteres para representar dados, por exemplo, números ou palavras.

Formato para intercâmbio de dados

É o texto usado para trocar dados entre plataformas ou sistemas.

Portabilidade

Corresponde à transferência de informações entre plataformas de maneira compatível para ambos os sistemas.

Também discutimos os seguintes conceitos essenciais:

- O JSON é um formato para intercâmbio de dados.
- O JSON é independente de linguagem de programação (o JavaScript não é necessário para usá-lo).
- O JSON é baseado na notação de objetos literais do JavaScript (ênfase na palavra “notação”).
- O JSON representa os dados de maneira alinhada aos conceitos universais de programação.

CAPÍTULO 2

Sintaxe do JSON

JSON é baseado nos objetos literais do JavaScript

Em inglês (ou em português), a palavra “literal” é um adjetivo usado para indicar que o que está sendo dito é exatamente aquilo, e não uma metáfora. Quando um amigo diz: “Ela apareceu do nada e eu literalmente deixei cair o meu sanduíche”, ele está afirmando que deixar o sanduíche cair não é uma metáfora.

Em programação, a palavra “literal” é um substantivo. Um *literal* é um valor que é literalmente representado com dados. Ele é escrito exatamente como deve ser interpretado. Se você não tiver familiaridade com os conceitos de programação, isso poderá parecer estranho. Vamos dar uma olhada rápida nos literais.

Você carrega dinheiro vivo ou um cartão de débito em sua carteira? Quando paro em uma lanchonete e entrego uma nota de cinco dólares ao caixa para pagar o meu sanduíche, vejo meus cinco dólares saírem fisicamente de minha carteira. Quando passo o meu cartão de débito para pagar um sanduíche, sei que terei cinco dólares a menos em minha conta bancária, mesmo que eu não tenha visto isso acontecer.

Em programação, geralmente usamos variáveis para representar valores. Por exemplo, posso usar uma *variável* que chamarei de *x* em uma expressão como esta:

```
x = 5
```

Então, mais tarde, posso querer somar cinco a *x*:

```
x = x + 5
```

Nesse ponto, sabemos que o valor de *x* é 10, porém não *vemos* o número 10.

Nesse exemplo, *x* é uma variável e 5 é um literal. Em nosso exemplo da lanchonete, podemos dizer que a nota de cinco dólares era um literal e o cartão de débito era uma variável. Quando *vemos* o valor propriamente dito, ele é um valor literal.

No exemplo com "*x = 5*", 5 é um literal numérico. Um número é um tipo de dado. Outros tipos de dados são strings (compostas de caracteres), booleanos (verdadeiro ou falso), nulo (nada), coleções de valores e objetos. Representar um valor numérico de maneira que possamos vê-lo é simples e usamos um caractere numérico para isso. Representar um valor booleano também é simples e podemos usar true/false ou 0/1. Se tiver familiaridade com o conceito de objetos, você saberá que representar um objeto não é uma questão fácil nem simples. Se você não tiver familiaridade com esse conceito, também não será um problema.

Em programação, o conceito de objeto é semelhante ao modo como você descreveria um objeto do mundo real, por exemplo, seu calçado. Você pode descrever seu calçado com atributos ou propriedades como cor, estilo, marca e o tipo de palmilha. Alguns dos valores desses atributos podem ser um número, por exemplo, o tamanho do calçado, enquanto outros podem ser um valor booleano (verdadeiro/falso), como "tem cadarço". O exemplo 2.1 mostra um exemplo.

EXEMPLO 2.1 – USANDO JSON PARA DESCREVER O CALÇADO QUE ESTOU USANDO AGORA

```
{
   "brand": "Crocs",
   "color": "pink",
   "size": 9,
   "hasLaces": false
}
```

Não se preocupe ainda com a sintaxe no exemplo do calçado; chegaremos lá mais adiante neste capítulo. A questão principal no exemplo do calçado é que você (mesmo sendo um ser humano) é capaz de ler literalmente os atributos de meu calçado. O tipo de dado em meu exemplo de JSON para o calçado é um objeto. O valor literal do objeto expõe as propriedades ou os atributos de

maneira que é possível vê-los (e lê-los). Esses atributos ou propriedades do objeto calçado são representados por pares nome-valor.

O JSON é baseado em objetos literais do JavaScript. O termo essencial aqui é “baseado em”. Em JavaScript (e na maioria das linguagens de programação com objetos), o objeto pode incluir uma função. Sendo assim, não só posso representar as propriedades de meu calçado com um objeto JavaScript, como também poderia criar uma função chamada “walk”.

Entretanto, o intercâmbio de dados diz respeito aos dados, portanto o JSON não inclui as funções dos objetos literais do JavaScript. A maneira como o JSON é baseado nos objetos literais do JavaScript está puramente relacionada à representação sintática do objeto literal e de suas propriedades. Essa representação das propriedades é feita por meio de pares nome-valor.

Pares nome-valor

O conceito de pares nome-valor é muito difundido em computação. Esses pares são conhecidos também por outros nomes: pares chave-valor, pares atributo-valor e pares campo-valor. Neste livro, eles serão chamados de pares nome-valor.

Se tiver familiaridade com o conceito de pares nome-valor, o JSON parecerá natural a você. Se não tiver familiaridade com eles, também não haverá problemas. Vamos dar uma olhada rapidamente nos pares nome-valor.

Em um par nome-valor, inicialmente você deve declarar o nome. Por exemplo, `"animal"`. Todavia, um par implica dois elementos: um nome e um valor. Sendo assim, vamos dar um valor ao nosso nome (nesse caso, para `"animal"`). Para simplificar esse conceito neste capítulo, vamos usar um valor de string. Em pares nome-valor em JSON, o valor também pode ser um número, um booleano, null (nulo), um array ou um objeto. Vamos detalhar mais esse assunto quando usarmos outros tipos de dados além de strings no capítulo 3. Desse modo, para esse par nome-valor cujo nome é `"animal"` usaremos o valor de string `"cat"`:

```
"animal" : "cat"
```

`"animal"` é o nome e `"cat"` é o valor. Há diversas maneiras entre as quais

podemos escolher para delimitar, isto é, separar o nome do valor. Se eu apresentasse uma lista dos funcionários de uma empresa, juntamente com seus cargos, provavelmente eu daria a você uma lista com o seguinte aspecto:

- Bob Barker, diretor executivo
- Janet Jackson, diretor de operações
- Sr. Ed, diretor financeiro

Em minha lista de funcionários, usei vírgulas para separar os cargos (nomes) e os nomes dos funcionários (valores). Também coloquei o valor à esquerda e o nome à direita.

O JSON utiliza o caractere dois-pontos (:) para separar os nomes dos valores. O nome está sempre à esquerda e o valor está sempre à direita. Vamos dar uma olhada em mais alguns exemplos:

```
"animal" : "horse"
"animal" : "dog"
```

É simples, não é mesmo? Basta um nome e um valor e você terá um par nome-valor.

Sintaxe apropriada do JSON

Agora vamos dar uma olhada no que a sintaxe apropriada do JSON implica. O nome, que em nosso exemplo é `"animal"`, deve estar sempre entre aspas duplas. O nome entre aspas duplas pode ser qualquer string válida. Sendo assim, podemos ter um nome com o aspecto a seguir, e ele será um JSON perfeitamente válido:

```
"My animal": "cat"
```

Você pode até mesmo inserir um apóstrofo no nome:

```
"Lindsay's animal": "cat"
```

Agora que você sabe que esse é um JSON válido, vou dizer por que você não deve fazer isso. Os pares nome-valor usados em JSON constituem uma estrutura de dados amigável para diversos sistemas. Ter um espaço ou um caractere especial (diferente de a-z, 0-9) no nome não estaria levando a *portabilidade* em consideração. No capítulo 1, definimos esse termo essencial

como a “transferência de informações entre plataformas de maneira compatível para ambos os sistemas”. Podemos definir nossos dados JSON de modo a reduzir a portabilidade; sendo assim, dizemos que é importante evitar espaços ou caracteres especiais para ter o *máximo de portabilidade*.

Caso o nome no par nome-valor de seus dados JSON seja carregado na memória por um sistema na forma de um objeto, ele se transformará em uma “propriedade” ou em um “atributo”. Uma propriedade ou atributo em alguns sistemas pode incluir um caractere underscore (_) ou números, porém, na maioria dos casos, ater-se aos caracteres do alfabeto, ou seja, A–Z ou a–z, é considerado um formato adequado. Sendo assim, se eu quiser incluir várias palavras no nome, vou formatá-lo da seguinte maneira:

```
"lindsaysAnimal": "cat"
```

ou

```
"myAnimal": "cat"
```

O valor `"cat"` no exemplo tem aspas duplas. De modo diferente do nome no par nome-valor, o valor nem sempre tem aspas duplas. Se o nosso valor for um dado do tipo string, devemos usar aspas duplas. Em JSON, os demais tipos de dados são números, booleanos, arrays, objetos e null. Eles não estarão entre aspas duplas. O formato desses tipos será discutido no capítulo 3.

JSON quer dizer JavaScript Object Notation (Notação de objetos JavaScript). Sendo assim, o único elemento que está faltando é a sintaxe que o transforma em um objeto. Devemos usar chaves em torno de nosso par nome-valor para transformá-lo em um objeto. Então usamos uma chave no início e outra no fim:

```
{ "animal" : "cat" }
```

Ao formatar o seu JSON, pense em uma cerimônia de cavaleiros em que o mestre de cerimônia toca os ombros do novo cavaleiro com uma espada. Você é o mestre de cerimônia e deve tocar seus dados JSON com uma chave de cada lado para transformá-los em um objeto. “Eu vos nomeio sir JSON.” A cerimônia não estaria completa sem um toque em cada ombro.

Em JSON, vários pares nome-valor são separados por uma vírgula. Assim,

para expandir o exemplo com animal/cat, vamos acrescentar uma cor (color):

```
{ "animal" : "cat", "color" : "orange" }
```

Outra maneira de olhar para a sintaxe do JSON é através dos olhos do computador que a estiver lendo. De modo diferente dos seres humanos, os computadores são criaturas estritamente orientadas a regras e instruções. Ao usar qualquer um dos caracteres a seguir externamente a um valor de string (não cercado por aspas), você estará fornecendo uma instrução sobre a maneira como os seus dados deverão ser lidos:

- { (chave à esquerda) quer dizer “iniciar um objeto”;
- } (chave à direita) quer dizer “finalizar um objeto”;
- [(colchete à esquerda) quer dizer “iniciar array”;
-] (colchete à direita) quer dizer “finalizar array”;
- : (dois-pontos) quer dizer “separar um nome e um valor em um par nome-valor”;
- , (vírgula) quer dizer “separar um par nome-valor em um objeto” ou “separar um valor em um array”; também pode ser lida como “aqui vai o próximo”.

Se você se esquecer de “finalizar um objeto” com uma chave à direita, seu objeto não será reconhecido como um objeto. Se colocar uma vírgula no final de sua lista de pares nome-valor, você estará fornecendo a instrução “aqui vai o próximo” e não estará fornecendo esse valor. Sendo assim, é importante usar a sintaxe corretamente.

Uma história sobre aspas duplas em JSON

Certo dia, estava espiando por sobre os ombros de um aluno, observando a sua tela do computador. Ele estava me mostrando alguns dados JSON que estava prestes a validar (Exemplo 2.2).

EXEMPLO 2.2 – O “JSON” QUE NÃO PODIA SER VALIDADO

```
{
   title : "This is my title.",
   body : "This is the body."
}
```

Na validação, ele recebia um erro de parsing e ficava frustrado. Ele disse: “Olhe, não há nada de errado com esse dados!”.

Chamei a sua atenção para as aspas ausentes em torno de "title" e de "body". Ele disse: “Mas já vi JSON formatado de ambas as maneiras, com e sem aspas em torno dos nomes”. “Ah” – eu disse – “Quando você viu os nomes sem aspas ao redor, aquilo não era JSON. Era um objeto JavaScript”.

Essa confusão é compreensível. O JSON é baseado nos objetos literais do JavaScript, portanto é muito semelhante a ele. Um objeto literal em JavaScript não precisa de aspas ao redor do nome no par nome-valor. Em JSON, elas são absolutamente obrigatórias.

Outro ponto que causa confusão pode ser o uso de aspas simples no lugar de aspas duplas. Em JavaScript, um objeto pode ter aspas simples na sintaxe no lugar de aspas duplas (veja o exemplo 2.3).

EXEMPLO 2.3 – ESTE NÃO É UM JSON VÁLIDO

```
{
  'title': 'This is my title.',
  'body': 'This is the body.'
}
```

Em JSON, somente as aspas duplas são usadas e elas são absolutamente obrigatórias em torno do nome em um par nome-valor (veja o exemplo 2.4).

EXEMPLO 2.4 – JSON VÁLIDO

```
{
  "title": "This is my title.",
  "body": "This is the body."
}
```

Validação de sintaxe

De modo diferente dos computadores, um ser humano usando um teclado pode cometer um erro simplesmente ao digitar errado. Na verdade, é impressionante como não cometemos mais erros do que os que geramos. Validar JSON é uma parte importante de trabalhar com ele.

Seu IDE (Integrated Development Environment, ou Ambiente de desenvolvimento integrado) pode ter uma ferramenta de validação incluída para o seu JSON. Se o seu IDE aceita plug-ins e add-ons, você poderá encontrar uma ferramenta de validação nesses recursos caso ela ainda não

esteja integrada. Se você não usa um IDE ou não tem a menor ideia do que estou falando, também não haverá problemas.

Há diversas ferramentas online para formatação e validação de JSON. Um passeio rápido pela sua ferramenta de pesquisa em busca de "JSON validation" (validação JSON) fornecerá diversos resultados. Eis alguns que valem a pena mencionar:

JSON Formatter & Validator (http://jsonformatter.curiousconcept.com/)
: É uma ferramenta de formatação com opções e uma bela UI que destaca os erros. O JSON processado é exibido em uma janela que serve como uma ferramenta de visualização em estilo de árvore/nós e também como uma janela a partir da qual é possível copiar e colar o código formatado.

JSON Editor Online (https://www.jsoneditoronline.org/)
: Uma ferramenta de validação, formatação e visualização de JSON, com tudo em um só local. Um indicador de erro é exibido na linha em que estiver o erro. Durante a validação, informações úteis de erros de parsing são exibidas. A ferramenta de visualização apresenta seus dados JSON em um formato de árvore/nós.

JSON (http://jsonlint.com/)
: Uma ferramenta de validação para JSON somente com o essencial. Basta copiar, colar e clicar em "validate" (validar). Ela também faz a gentileza de formatar os seus dados JSON.

Essas são ferramentas para *validação de sintaxe*. Mais adiante, no capítulo 4, discutiremos outro tipo de validação chamada validação de conformidade. A validação de sintaxe está relacionada à forma do JSON em si, enquanto a validação de conformidade diz respeito a uma estrutura de dados única. No exemplo 2.5, a validação de sintaxe estará preocupada se o nosso JSON está correto (se tem chaves ao redor, se os pares nome-valor estão separados por vírgulas). A validação de conformidade estará preocupada em saber se nossos dados incluem um nome (name), uma raça (breed) e uma idade (age). Além disso, a validação de conformidade estará interessada em saber se o valor da idade é um número e se o valor do nome é uma string.

EXEMPLO 2.5 – EXEMPLO DE VALIDAÇÃO

```
{
   "name": "Fluffy",
   "breed": "Siamese",
   "age": 2
}
```

JSON como um documento

Você poderá achar que, em suas futuras experiências com JSON, você somente o criará no código e o passará por aí em um mundo invisível, e ele poderá ser inspecionado apenas por ferramentas de desenvolvedor. Entretanto, como um formato para intercâmbio de dados, o JSON pode constituir um documento próprio e residir em um sistema de arquivos. A extensão de arquivo para o JSON é fácil de lembrar: *.json*.

Desse modo, se eu salvar meu JSON com animal/cat em um arquivo e o armazenar em meu computador, esse arquivo poderá ser semelhante a: *C:\animals.json*.

Tipo de mídia JSON

Com frequência, ao passar dados para outras pessoas, você deverá informar-lhes qual é o tipo do dado previamente. Você pode ter ouvido falar disso como tipo de mídia da Internet, tipo de conteúdo ou tipo MIME. Esse tipo é formatado como *tipo*/*subtipo*. Um tipo do qual você já deve ter ouvido falar é text/html.

O tipo MIME para JSON é application/json.

O IANA (Internet Assigned Numbers Authority, ou Autoridade para atribuição de números da Internet) mantém uma lista completa dos tipos de mídia (*http://www.iana.org/assignments/media-types/media-types.xhtml*).

Termos e conceitos essenciais

Este capítulo discutiu os seguintes termos essenciais:

Literal

Um valor escrito exatamente como deve ser interpretado.

Variável

Um valor que pode ser alterado e é representado por um identificador, por exemplo, *x*.

Máximo de portabilidade (no intercâmbio de dados)

Transcender a portabilidade básica do formato de dados ao garantir que os dados propriamente ditos sejam compatíveis entre sistemas e plataformas diferentes.

Par nome-valor

Um par nome-valor (ou par chave-valor) é uma propriedade ou um atributo com um nome e um valor correspondente.

Validação de sintaxe

É a validação que diz respeito à forma do JSON.

Validação de conformidade

É a validação que diz respeito às estruturas de dados únicas.

Também discutimos os seguintes conceitos essenciais:

- O JSON é baseado na representação sintática das propriedades dos objetos literais do JavaScript. Isso *não* inclui as funções dos objetos literais do JavaScript.
- No par nome-valor em JSON, o nome deve estar sempre entre aspas duplas.
- No par nome-valor em JSON, o valor pode ser uma string, um número, um booleano, null (nulo), um objeto ou um array.
- A lista de pares nome-valor em JSON deve estar entre chaves.
- Em JSON, vários pares nome-valor são separados por uma vírgula.
- Os arquivos JSON utilizam a extensão *.json*.
- O tipo de mídia para JSON é application/json.

CAPÍTULO 3

Tipos de dados JSON

Se você já aprendeu uma ou duas linguagens de programação, provavelmente você entende o que são os tipos de dados. Se não entender, também não será um problema. Vamos dar uma olhada neles rapidamente.

Uma olhada rápida nos tipos de dados

Pense no que aconteceria se você entregasse um martelo a um garotinho que não saiba nada sobre ferramentas e você não lhe diz para que serve essa ferramenta. Provavelmente, haverá danos materiais e corporais. Se essa criança for bem-comportada e tiver boa coordenação, podemos dar um conjunto de instruções a ela sobre o uso do martelo. Em vez de correr por aí destruindo objetos, a criança usaria o martelo somente para bater em pregos e removê-los (lembre-se de que é uma criança bem-comportada). Além disso, quando você disser à criança “Você poderia me passar o martelo, por favor?”, ela não lhe passaria a chave de fenda. Saber o que é algo com antecedência e como usá-lo é tão útil em computação quanto no mundo real.

Em computação, com muita frequência, devemos saber qual é o tipo de dado com o qual estamos lidando, pois podemos realizar tarefas distintas com tipos diferentes de dados. Posso multiplicar um número por outro número, mas não posso multiplicar uma palavra por um número. Se eu tiver uma lista de palavras, poderei colocá-la em ordem alfabética. Porém, não posso ordenar o número 5 em ordem alfabética. Desse modo, em programação, quando um método (ou função) diz “Você poderia me passar o número, por favor?”, se soubermos o que é um número, não cometeremos o erro de lhe passar a palavra “ketchup”.

Em ciência da computação, há um conjunto de tipos de dados que são

chamados de tipos primitivos. A palavra “primitivo” evoca imagens de homens da caverna na Idade das Pedras, sentados ao redor de uma fogueira, grunhindo e afiando lanças. Isso não quer dizer que os tipos primitivos de dados sejam brutos como os homens das cavernas; significa que eles estão entre os primeiros tipos mais básicos de dados. Assim como o homem moderno e o homem das cavernas, alguns dos tipos de dados mais modernos e progressistas têm suas raízes nesses tipos primitivos de dados:

- Números (por exemplo, 5 ou 5,09)
  - Inteiro
  - Número de ponto flutuante
  - Número de ponto fixo
- Caracteres e strings (por exemplo, “a”, “A” ou “apple”)
- Booleanos (isto é, verdadeiro ou falso)

Em linguagens de programação diferentes, os tipos de dados “gravados a ferro e fogo” geralmente são chamados de tipos primitivos de dados – ou tipos built-in – da linguagem. Isso quer dizer que a definição do tipo e o que pode ser feito com ele não podem mudar. A linguagem de programação não permitirá que você redefina o que significa somar dois números. Esses tipos primitivos de dados variam de uma linguagem para outra e, com frequência, incluirão acréscimos à lista anterior, por exemplo, podem incluir um byte ou uma referência (ou um ponteiro ou handler).

Além dos tipos primitivos de dados, há outros tipos usados na maioria das linguagens de programação. Geralmente, eles são chamados de tipos de dados compostos, pois resultam de uma fusão ou de uma composição dos tipos primitivos de dados. Os tipos de dados compostos, assim como um castelo de areia, têm uma estrutura. Se observarmos o castelo de areia por partes, poderemos ver que a estrutura é constituída de areia, gravetos e água. Se separarmos a estrutura de dados de um tipo de dado composto, veremos que ela é constituída de nossos tipos primitivos de dados.

Um exemplo de um tipo de dado composto comumente usado em linguagens de programação é o tipo de dado enumerado. Anteriormente, mencionei a ordenação de uma lista em ordem alfabética. Uma lista de palavras pode ser

representada com tipos diferentes de dados em linguagens de programação distintas (por exemplo, como uma lista ou um array). Se separarmos essa estrutura de dados, veremos que ela é composta de caracteres e de strings, que são tipos primitivos de dados. O tipo de dado enumerado é uma estrutura de dados que pode ser enumerada. Posso mencionar cada item da estrutura, um por um, e posso também contar quantos itens há. Veja o exemplo 3.1.

Exemplo 3.1 – "Deixe-me enumerar as boas qualidades de sua personalidade" pode ser representado em programação como um array literal

```
[
  "witty",
  "charming",
  "brave",
  "bold"
]
```

Não é preciso entender a estrutura de dados do array literal desse exemplo para ver que podemos mencionar cada uma dessas "boas qualidades", uma a uma, além de afirmar que há quatro delas.

Outro tipo de dado composto é o tipo de dado objeto. No capítulo 2, exploramos o tipo de dado objeto, pois o JavaScript Object Notation é baseado na notação de objeto literal do JavaScript. Além disso, usei o JSON para descrever o calçado que eu estava usando (veja o exemplo 3.2).

Exemplo 3.2 – Meu calçado descrito em JSON

```
{
  "brand": "Crocs",
  "color": "pink",
  "size": 9,
  "hasLaces": false
}
```

Esse objeto literal nos permite ver que o tipo de dado objeto, nesse caso, é composto de pares nome-valor. Se essa estrutura de dados for decomposta, veremos que ela é constituída de tipos primitivos de dados: string, número e booleano. Os nomes (`"brand"`, `"color"`, `"size"`, `"hasLaces"`) nos pares nome-valor são todos do tipo string. Os valores `"Crocs"` e `"pink"` são ambos do tipo string.

O valor "9" é do tipo numérico e o valor "false" é do tipo booleano.

Tipos de dados JSON

Embora as linguagens de programação possam variar quando se trata de tipos compostos e possam ter pequenas diferenças até mesmo quanto aos tipos primitivos adicionais, a maioria delas compartilha os tipos primitivos que mencionei anteriormente neste capítulo:

- Números (por exemplo, 5 ou 5,09)
  - Inteiro
  - Número de ponto flutuante
  - Número de ponto fixo
- Caracteres e strings (por exemplo, “a”, “A” ou “apple”)
- Booleanos (isto é, verdadeiro ou falso)

O tipo de dado objeto é uma estrutura de dados comum a algumas das linguagens de programação mais populares, por exemplo, Java e C#, mas não a todas. Pelo fato de o JSON ser baseado na notação de objetos literais e no tipo de dado objeto, você poderia achar que isso seria um problema em um formato para intercâmbio de dados. Afinal de contas, o objetivo de um formato de dados é permitir a comunicação entre dois sistemas diferentes, e o território comum deverá ser expresso nesse formato.

Lembre-se de que a estrutura de dados que forma o tipo de dado objeto, que é composto, pode ser separada em tipos primitivos de dados. Depois que a estrutura de dados do objeto for decomposta nesses tipos nativos, até mesmo para as linguagens de programação em que não há o tipo de dado objeto, esses dados serão bem compreensíveis.

Os tipos de dados JSON são:

- Objeto
- String
- Número
- Booleano

- Null (nulo)
- Array

Tipo de dado objeto do JSON

O tipo de dado objeto do JSON é simples. Em sua raiz, o JSON é um objeto. É uma lista de pares nome-valor entre chaves. Ao criar um par nome-valor que também seja um objeto em seus dados JSON, a sua estrutura começará a ficar aninhada. No exemplo 3.3, isso é mostrado por meio da descrição de uma pessoa com objetos aninhados.

EXEMPLO 3.3 – OBJETOS ANINHADOS

```
{
   "person": {
      "name": "Lindsay Bassett",
      "heightInInches": 66,
      "head": {
         "hair": {
            "color": "light blond",
            "length": "short",
            "style": "A-line"
         },
         "eyes": "green"
      }
   }
}
```

O par nome-valor de nível mais alto nesse caso é "person", cujo valor é um objeto. Esse objeto tem três pares nome-valor: "name", "heightInInches" e "head". O par nome-valor com "name" tem um valor de string igual a "Lindsay Bassett". O par nome-valor com "heightInInches" tem um valor numérico. O par nome-valor com "head" tem um valor do tipo objeto. O par nome-valor com "hair" também tem um valor do tipo objeto, com três pares nome-valor do tipo string: "color", "length" e "style". O objeto "head" também tem um par nome-valor com "eyes" cujo valor é "green".

Tipo de dado string do JSON

Exploramos rapidamente o tipo de dado string do JSON anteriormente neste livro no exemplo com animal/cat:

```
{ "animal" : "cat" }
```

O valor "cat" é do tipo string. No mundo real, a menos que esse dado seja para um pet shop, os valores de string nos dados não serão tão simples assim. Até mesmo um pet shop poderá ter mais a dizer em seus dados do que uma única palavra como “cat”. Talvez o pet shop queira comunicar os detalhes de sua última promoção:

> Today at Bob’s Best Pets you can get a free 8 oz. sample bag of Bill’s Kibble with your purchase of a puppy. Just say “Bob’s the best!” at checkout.[1]

A string JSON pode ser composta de qualquer caractere Unicode; todos os caracteres do texto promocional apresentado são válidos. Um valor de string sempre deve estar entre aspas *duplas*.

Na seção “Uma história sobre aspas duplas em JSON” do capítulo 2, mencionei que as aspas simples em torno de um valor de string não são válidas (Exemplo 3.4).

EXEMPLO 3.4 – ESTE NÃO É UM JSON VÁLIDO

```
{
  'title': 'This is my title.',
  'body': 'This is the body.'
}
```

Isso pode ser confuso, especialmente se você já viu objetos literais em JavaScript que usassem aspas simples. Em JavaScript, podemos usar aspas simples ou aspas duplas de forma indistinta. Entretanto, é importante lembrar que o JSON não é um objeto literal do JavaScript; ele é somente *baseado* nesses objetos. Em JSON, somente as aspas duplas são permitidas em torno de um valor de string.

Também mencionamos no capítulo anterior que o JSON é lido por um parser. Aos olhos de um parser, quando um valor começa com um caractere de aspas duplas ("), espera-se que a string de texto seja finalizada com outro caractere de aspas duplas. Isso representará um problema se a string de texto contiver

aspas duplas.

Por exemplo, suponha que estejamos realizando uma promoção em um pet shop, em que os clientes devam dizer “Bob’s the best!” (Bob’s é o melhor) no caixa para receber um pacote gratuito de ração. Se usarmos o código do exemplo 3.5, vamos ter um problema, pois não podemos simplesmente colocar os dados da promoção entre aspas duplas.

EXEMPLO 3.5 – ESTE CÓDIGO NÃO FUNCIONARÁ

```
{
   "promo": "Say "Bob's the best!" at checkout for free 8oz bag of kibble."
}
```

Há aspas dentro do valor e o parser lerá o primeiro caractere de aspas na frente de “Bob” no texto promocional como o final da string. Em seguida, quando o parser encontrar o restante do texto simplesmente isolado ali, sem que pertença a um par nome-valor, um erro será gerado. Para lidar com essa situação, devemos escapar nosso caractere de aspas dentro de qualquer valor de string inserindo um caractere de barra invertida (\) antes dele, conforme mostrado no exemplo 3.6.

EXEMPLO 3.6 – USAR UM CARACTERE DE BARRA INVERTIDA PARA ESCAPAR AS ASPAS EM STRINGS RESOLVE O PROBLEMA

```
{
   "promo": "Say \"Bob's the best!\" at checkout for free 8oz bag of kibble."
}
```

Esse caractere de barra invertida informará o parser que o caractere de aspas não finaliza a string. Depois que o parser realmente carregar a string na memória, qualquer caractere de barra invertida que venha antes de um caractere de aspas será removido e o texto surgirá do outro lado conforme desejado.

As aspas não são os únicos caracteres que devem ser escapados quando se trata de strings JSON. Como o caractere de barra invertida é usado para escapar outros caracteres, devemos também escapar a barra invertida. Por exemplo, o JSON mostrado no exemplo 3.7, que tem como finalidade informar o local em que está o meu diretório *Program Files*, produzirá um

erro. Para corrigir esse problema, devemos escapar o caractere de barra invertida acrescentando outro caractere de barra invertida, conforme mostrado no exemplo 3.8.

EXEMPLO 3.7 – A BARRA INVERTIDA USADA NESTE CÓDIGO PRODUZIRÁ UM ERRO

```
{
  "location": "C:\Program Files"
}
```

EXEMPLO 3.8 – O CARACTERE DE BARRA INVERTIDA DEVE SER ESCAPADO COM OUTRO CARACTERE DE BARRA INVERTIDA

```
{
  "location": "C:\\Program Files"
}
```

Além dos caracteres de aspas duplas e de barra invertida, devemos escapar os seguintes caracteres:

- \/ (barra para frente)
- \b (backspace)
- \f (form feed, ou alimentação de formulário)
- \t (tabulação)
- \n (quebra de linha)
- \r (carriage return)
- \u seguido de caracteres hexadecimais (por exemplo, o emoticon da carinha feliz, \u263A)

O JSON mostrado no exemplo 3.9 provocará um erro de parser porque os caracteres de tabulação e de quebra de linha devem ser escapados. O exemplo 3.10 mostra como corrigir o problema.

EXEMPLO 3.9 – OS CARACTERES DE TABULAÇÃO E DE QUEBRA DE LINHA USADOS NESTE JSON PROVOCARÃO UM ERRO

```
{
  "story": "\t Once upon a time, in a far away land \n there lived a princess."
}
```

EXEMPLO 3.10 – JSON COM OS CARACTERES DE TABULAÇÃO E DE QUEBRA DE LINHA ESCAPADOS

```
{
   "story": "\\t Once upon a time, in a far away land \\n there lived a princess."
}
```

Tipo de dado numérico do JSON

Os números são uma informação comum a ser passada por aí nos dados. Quantidade em estoque, dinheiro, latitude/longitude e a massa da Terra são dados que podem ser representados como números – veja o exemplo 3.11.

EXEMPLO 3.11 – REPRESENTANDO NÚMEROS EM JSON

```
{
   "widgetInventory": 289,
   "sadSavingsAccount": 22.59,
   "seattleLatitude": 47.606209,
   "seattleLongitude": -122.332071,
   "earthsMass": 5.97219e+24
}
```

Um número em JSON pode ser inteiro, decimal, um número negativo ou um valor exponencial.

Meu estoque de equipamentos contém 289 itens. A quantidade em estoque normalmente é representada por um inteiro. Geralmente, não vendo metade de um item, portanto meu número para o estoque jamais incluirá um ponto decimal.

Minha triste conta bancária tem U$ 22,50. Embora algumas linguagens de programação tenham um tipo de dado para moeda, geralmente representamos o dinheiro em JSON como um número decimal e omitimos o $.

Se você der uma olhada na latitude e na longitude da cidade de Seattle, verá que ambas são representadas por números decimais e que a longitude é um número decimal negativo. Os números negativos são representados pelo sinal de menos padrão antes do número.

Além disso, estou representando o número bem grande correspondente à

massa da Terra em quilogramas usando a Notação E. A Notação E é particularmente conveniente para dados científicos e é um tipo de número aceito.

Tipo de dado booleano do JSON

Em linguagem natural, duas das respostas mais simples que temos para as perguntas são “sim” e “não”. Se você perguntar ao seu amigo “Você quer uma torrada com ovos?”, ele responderá “sim” ou “não”.

Na programação de computadores, o tipo de dado booleano é simples. Ele pode ser verdadeiro (true) ou falso (false). Se você perguntar ao seu computador “Você quer uma torrada com ovos?”, ele responderá “true” ou “false”.

Em algumas linguagens de programação, o valor literal para true pode ser 1, e 0 é usado para false. Às vezes, os caracteres para os valores literais podem ser letras maiúsculas e minúsculas – por exemplo, True ou TRUE e false ou FALSE. Em JSON, o valor literal do tipo de dado booleano é sempre escrito com letras minúsculas: true ou false. Qualquer outro tipo de letra resultará em erro. No exemplo 3.12, os booleanos são usados para informar dados sobre as minhas preferências para o café da manhã e o almoço.

EXEMPLO 3.12 – PREFERÊNCIAS

```
{
   "toastWithBreakfast": false,
   "breadWithLunch": true
}
```

Tipo de dado null do JSON

Quando não tivermos nada de algum item, você poderá achar apropriado dizer que há zero itens. Neste momento, tenho zero relógios de pulso. O problema é que zero é um número.

Isso implica que estávamos contando, antes de tudo.

E se houvesse um formato-padrão para descrever o pulso de uma pessoa em JSON que incluísse alguns atributos? Veja os exemplos 3.13 e 3.14.

Exemplo 3.13 – Meu vizinho Bob poderá ter esta descrição

```
{
  "freckleCount": 0,
  "hairy": true,
  "watchColor": "blue"
}
```

Exemplo 3.14 – Minha descrição poderá ter este aspecto

```
{
  "freckleCount": 1,
  "hairy": false,
  "watchColor": null
}
```

Não tenho uma cor para o relógio de pulso porque não estou usando nenhum relógio. Em programação, null é uma maneira de dizer zero ou nada sem a necessidade de usar um número. O valor para a cor do relógio não pode ser definido; sendo assim, é null.

Entretanto, null não deve ser confundido com undefined, um valor com o qual você poderá se deparar em JavaScript. undefined não é um tipo de dado JSON, porém, em JavaScript, undefined é o que você obterá ao tentar acessar um objeto ou uma variável que não exista. Em JavaScript, undefined tem uma relação com um objeto ou com uma variável cujo nome e o valor não existam, enquanto null tem uma relação somente com o valor de um objeto ou de uma variável. null é um valor que quer dizer “nenhum valor”. Em JSON, null sempre deve ser escrito somente com letras minúsculas.

Tipo de dado array do JSON

Vamos agora explorar o tipo de dado array. Se você não tiver familiaridade com arrays, isso não será um problema. Vamos dar uma olhada rápida no que é um array.

Pense em uma embalagem que armazene uma dúzia de ovos. A embalagem tem doze compartimentos disponíveis para os ovos. Quando comprei os ovos, havia doze. Esse array terá um tamanho igual a 12 e conterá 12 ovos. Veja o exemplo 3.15.

Exemplo 3.15 – Este é um array de strings (por questões de simplicidade, usarei a string "egg" para cada um dos ovos nos compartimentos)

```
{
  "eggCarton": [
    "egg",
    "egg",
    "egg",
    "egg",
    "egg",
    "egg",
    "egg",
    "egg",
    "egg",
    "egg",
    "egg",
    "egg"
  ]
}
```

Observe que tenho um par nome-valor. O nome é "eggCarton" e o valor é um array. O array está sempre entre colchetes ([]). Dentro do array, temos uma lista e cada item da lista está separado por uma vírgula. Isso pode parecer semelhante ao modo como formatamos nossos pares nome-valor, porém a principal diferença está no fato de a lista ter somente valores. Esses valores podem ser de qualquer tipo de dado JSON válido (string, número, objeto, booleano, array e null).

Agora suponha que retirei dois ovos para fazer torrada com ovos no meu café da manhã. Minha embalagem com os ovos continua com doze compartimentos, porém estão faltando dois ovos. Veja o exemplo 3.16.

Exemplo 3.16 – Retirei dois ovos da embalagem para preparar o café da manhã.

```
{
  "eggCarton": [
    "egg",
    null,
    "egg",
    "egg",
```

```
      "egg",
      "egg",
      "egg",
      "egg",
      "egg",
      "egg",
      null,
      "egg"
   ]
}
```

Como podemos ver, retirei os ovos de dois compartimentos específicos. Esses compartimentos ficaram vazios, pois os ovos não existem mais. Representamos isso com null.

Um array tem um índice para cada um dos “compartimentos”. Começamos com 0, portanto o primeiro compartimento tem um índice igual a 0, o segundo tem um índice igual a 1, e assim sucessivamente. O último compartimento tem um índice igual a 11. Sendo assim, retirei os ovos dos índices 1 e 10. Esse é um array válido.

Se eu colocar o número 5 no compartimento vazio no índice 10, na maioria das linguagens de programação, esse será um array inválido. Veja o exemplo 3.17.

EXEMPLO 3.17 – ESTE NÃO SERÁ UM ARRAY VÁLIDO NA MAIORIA DAS LINGUAGENS DE PROGRAMAÇÃO

```
{
   "eggCarton": [
      "egg",
      null,
      "egg",
      "egg",
      "egg",
      "egg",
      "egg",
      "egg",
      "egg",
      "egg",
```

```
      5,
      "egg"
   ]
}
```

Eu disse “na maioria das linguagens de programação”, porém, em JSON, misturar e combinar tipos de dados é válido. Vou dizer o motivo e, em seguida, direi por que você não deverá fazer isso em seus dados JSON.

Em JavaScript, você pode definir uma variável. No exemplo 3.18, temos uma variável chamada something e lhe atribuímos o número 5 como o seu valor.

EXEMPLO 3.18 – DEFININDO UMA VARIÁVEL EM JAVASCRIPT

```
var something = 5;
```

Na linha logo a seguir, poderemos alterar essa variável para que ela tenha um valor do tipo string (Exemplo 3.19).

EXEMPLO 3.19 – ALTERANDO ESTA VARIÁVEL PARA QUE ELA TENHA UM VALOR DE STRING

```
something = "bob";
```

Podemos também alterá-la para que ela tenha um valor do tipo objeto (Exemplo 3.20).

EXEMPLO 3.20 – ALTERANDO ESTA VARIÁVEL PARA QUE ELA TENHA UM VALOR DO TIPO OBJETO

```
something = { person: "bob" };
```

O valor de meu var something (variável) pode ser um número, uma string, um array, null ou um objeto. Na maioria das linguagens de programação, as variáveis não podem mudar tanto assim. Normalmente, declaramos something como um inteiro, uma string ou um objeto. Desse modo, quando declarar sua variável chamada something, você especificará int something = 5. Você poderia usar string something = "bob" ou Person something = new Person("bob"). Portanto, na maioria das linguagens de programação, ao declarar um array, você estará declarando com antecedência o tipo de dado que deverá estar presente em seus contêineres e isso não poderá simplesmente ser alterado depois.

O JSON é um formato para intercâmbio de dados. Se você entregar o seu array JSON para alguém que não vá utilizá-lo com JavaScript, seu array

provocará um erro quando for submetido ao parsing.

Por exemplo, suponha que você esteja participando de uma convenção em que os comerciantes estejam vendendo coleções de pedras. Você tem uma coleção de pedras para vender. Uma pessoa aparece e quer comprar a sua coleção de 50 pedras; ele as leva consigo, porém, ao chegar em casa, descobre que a coleção de pedras não contém 50 pedras, mas tem 20 pedras, 20 gravetos e 10 chicletes (um dos quais já estava mascado).

Vamos dar uma olhada mais detalhada em alguns exemplos de arrays para cada tipo de dado. Em JSON, o array pode ser constituído de qualquer um dos tipos de dados possíveis. Sendo assim, podemos ter um array de strings, um arrays de números, um array de booleanos, um array de objetos ou um array de arrays. Um array de arrays é chamado de array multidimensional. Vamos dar uma olhada em alguns exemplos.

Suponha que tenhamos uma lista de nomes de alunos que se inscreveram para um curso. Isso pode ser representado por um array de strings (Exemplo 3.21).

EXEMPLO 3.21 – USANDO UM ARRAY DE STRINGS PARA REPRESENTAR UMA LISTA DE ALUNOS

```
{
  "students": [
    "Jane Thomas",
    "Bob Roberts",
    "Robert Bobert",
    "Thomas Janerson"
  ]
}
```

Depois que os alunos fizerem uma prova, podemos usar um array de números para representar suas notas (Exemplo 3.22).

EXEMPLO 3.22 – USANDO UM ARRAY DE NÚMEROS PARA REPRESENTAR AS NOTAS DA PROVA

```
{
  "scores": [
    93.5,
    66.7,
```

```
    87.6,
    92
  ]
}
```

Se for necessário criar um gabarito para uma prova do tipo verdadeiro/falso, poderemos usar um array de booleanos (Exemplo 3.23).

EXEMPLO 3.23 – USANDO UM ARRAY DE BOOLEANOS PARA REPRESENTAR AS RESPOSTAS DE UMA PROVA DO TIPO VERDADEIRO/FALSO

```
{
  "answers": [
    true,
    false,
    false,
    true,
    false,
    true,
    true
  ]
}
```

Um array de objetos pode ser usado para representar a prova completa, incluindo as perguntas e as respostas (Exemplo 3.24).

EXEMPLO 3.24 – USANDO UM ARRAY DE OBJETOS PARA REPRESENTAR AS PERGUNTAS E AS RESPOSTAS DE UMA PROVA

```
{
  "test": [
    {
      "question": "The sky is blue.",
      "answer": true
    },
    {
      "question": "The earth is flat.",
      "answer": false
    },
    {
```

```
            "question": "A cat is a dog.",
            "answer": false
        }
    ]
}
```

Para representar as respostas de três provas diferentes, um array de arrays – isto é, um array multidimensional – poderá ser usado (Exemplo 3.25).

EXEMPLO 3.25 – USANDO UM ARRAY DE ARRAYS PARA REPRESENTAR AS RESPOSTAS DE TRÊS PROVAS DIFERENTES

```
{
    "tests": [
        [
            true,
            false,
            false,
            false
        ],
        [
            true,
            true,
            true,
            true,
            false
        ],
        [
            true,
            false,
            true
        ]
    ]
}
```

Termos e conceitos essenciais

Este capítulo discutiu os seguintes termos essenciais:

Tipo de dado string do JSON

É um valor de string, por exemplo, "my string", entre aspas duplas.

Tipo de dado booleano do JSON

É um valor igual a verdadeiro ou falso.

Tipo de dado numérico do JSON

É um valor numérico, por exemplo, 42, que pode ser um inteiro, um decimal ou um valor exponencial, positivo ou negativo.

Tipo de dado null do JSON

É um valor null (nulo) que representa um valor vazio.

Tipo de dado array do JSON

Um array é uma coleção ou uma lista de valores e os valores podem ser do tipo string, número, booleano, objeto ou array; os valores em um array devem estar entre colchetes ([]) e são delimitados por vírgula.

Tipo de dado objeto do JSON

O tipo de dado objeto é um conjunto de pares nome-valor delimitados por uma vírgula e deve estar entre chaves ({}).

Também discutimos os seguintes conceitos essenciais:

- Os valores true ou false do tipo de dado booleano em JSON devem sempre usar letras minúsculas (ou seja, true, e não True ou TRUE).
- O valor null do tipo de dado nulo em JSON é sempre escrito com letras minúsculas (ou seja, null, e não NULL ou Null).
- Uma diferença fundamental entre um objeto e um array está no fato de um objeto ser uma lista ou uma coleção de pares nome-valor, enquanto um array é uma lista ou uma coleção de valores.
- Outra diferença importante entre um array e um objeto é que todos os valores de um array *devem* ter o mesmo tipo de dado.

---

1 N.T.: “Hoje, no Bob’s Best Pets você poderá levar uma amostra da embalagem de 250 gramas de Bill’s Kibble na compra de um filhote. Basta dizer “Bob’s é o melhor!” no caixa.”

CAPÍTULO 4

JSON Schema

No capítulo 3, discutimos os tipos de dados JSON. A importância e a utilidade dos tipos de dados foram discutidas. Saber com antecedência o que é algo e para que serve (lembre-se do garoto com o martelo) representa um mundo de diferença.

Na maioria dos cenários com formatos para intercâmbio de dados, os dados são criados para serem enviados pela Internet ou por uma rede para a outra parte. Essa parte normalmente espera receber um formato desejado para o documento que ela estiver esperando, com determinada estrutura e certos tipos de dados. Geralmente, essas partes fornecem uma documentação que explica o formato esperado e disponibilizam exemplos.

Mesmo quando a mais detalhada e bela das documentações é fornecida, não é difícil gerar erros em seus dados. Para esclarecer, não são os erros de sintaxe de que estamos falando aqui. São erros de compreensão, por exemplo, “Enviei uma maçã e você estava esperando uma laranja”. Neste livro, vou me referir a esse tipo de validação como *validação de conformidade* para que ela possa ser diferenciada da validação de sintaxe.

Nesse cenário, o processo normalmente ocorre por meio dos passos a seguir:

1. Você acabou de criar seus dados e está se sentindo confiante.
2. Você envia seus dados pela Internet para a outra parte. Sua conexão com a Internet está lenta hoje e o arquivo de dados que você está enviando é enorme, portanto isso demora alguns minutos.
3. Você obtém uma resposta indicando um erro porque seus dados não estavam formatados conforme o esperado. Sua autoconfiança diminui. Se estiver com sorte, a resposta com o erro terá alguma informação significativa, por exemplo, o que você fez de errado.

4. Você vasculha a documentação deles, encontra o que você acha que fez de errado, corrige e retorna ao passo 1.

Esse cenário já estava presente no intercâmbio de dados antes mesmo de o JSON existir. Felizmente, as pessoas que trabalham no mercado de tecnologia são solucionadoras de problemas e, desse modo, o conceito de esquema (schema) surgiu.

Contratos com validação mágica

No mundo real, geralmente usamos contratos entre duas partes quando o resultado é importante. Ao assinar um contrato que determine que terminarei um projeto para alguém, os detalhes são especificados nesse contrato. Concordo que entregarei a nave espacial no dia 31 de agosto e que o produto final incluirá uma espaçonave totalmente funcional com suporte à vida, lasers e três motores.

Suponha agora que vivemos em um mundo de encantamentos e magia. Quando a empresa para a qual estou desenvolvendo o projeto me entregou o contrato, ela acrescentou uma pitada de magia. A qualquer momento, posso tocar o contrato com minha varinha de condão e ele dirá se eu já cumpri a minha parte do trato. Jamais precisarei entrar em uma reunião para declarar que “Terminei!” e receber esta resposta embaraçosa: “O que você me diz do terceiro motor que você prometeu colocar na espaçonave? Onde está ele?”. A qualquer momento, posso verificar se *realmente* já concluí o projeto e poderei entrar na reunião com autoconfiança.

Um esquema no intercâmbio de dados é muito semelhante a esse mundo imaginário de encantamentos e magia. Antes de enviar nossos dados, a qualquer momento, podemos validá-los para verificar se estão de acordo com o esquema e descobrir se são aceitáveis. Quando fizermos um intercâmbio de dados usando um esquema, o processo será bem diferente de nosso cenário sem esquema:

1. Você deve validar se seus dados estão em conformidade com o seu esquema e corrigir qualquer erro identificado. Geralmente, você receberá informações úteis sobre os erros.

2. Você acabou de criar seus dados e está se sentindo confiante.
3. Você envia seus dados pela Internet e obtém uma resposta indicando sucesso. Missão cumprida.

Além disso, o JSON Schema pode ser usado na outra extremidade da transação pela parte que está aceitando os dados. Um JSON Schema pode representar a primeira linha de defesa na aceitação dos dados para verificar se os dados são adequados. Ele pode responder a todas as perguntas a seguir antes de os dados serem processados:

Os tipos de dados dos valores estão corretos?

Podemos especificar se um valor deve ser um número, uma string etc.

Essas informações incluem os dados obrigatórios?

Podemos especificar quais dados são obrigatórios e quais não são.

Os valores estão no formato exigido?

Podemos especificar intervalos, mínimos e máximos.

Introdução ao JSON Schema

Embora o JSON seja razoavelmente maduro, o JSON Schema continua em desenvolvimento. Em abril de 2015, o JSON Schema estava no draft 4. Isso não quer dizer que você não deva usar o JSON Schema – quer dizer apenas que ele continua evoluindo para melhor servir o mundo.

Um JSON Schema é escrito em JSON, portanto ler ou escrever um é um processo somente um pouco diferente. No primeiro par nome-valor de nosso JSON, devemos declará-lo como um schema document (documento de esquema, veja o exemplo 4.1).

EXEMPLO 4.1 – O NOME DESTA DECLARAÇÃO SEMPRE SERÁ "$SCHEMA" E O VALOR SEMPRE SERÁ O LINK PARA A VERSÃO DO DRAFT

```
{
  "$schema": "http://json-schema.org/draft-04/schema#"
}
```

O segundo par nome-valor em nosso JSON Schema Document será o título

(veja o exemplo 4.2).

Exemplo 4.2 – Formato para um documento que representa um gato (cat)

```
{
   "$schema": "http://json-schema.org/draft-04/schema#",
   "title": "Cat"
}
```

No terceiro par nome-valor de nosso JSON Schema Document, definiremos as propriedades que queremos que sejam incluídas no JSON. O valor de "properties" é essencialmente um esqueleto dos pares nome-valor do JSON que queremos. Em vez de um valor literal, temos um objeto que define o tipo de dado e, opcionalmente, a descrição (Exemplo 4.3).

Exemplo 4.3 – Definindo as propriedades de um gato (cat)

```
{
   "$schema": "http://json-schema.org/draft-04/schema#",
   "title": "Cat",
   "properties": {
      "name": {
         "type": "string"
      },
      "age": {
         "type": "number",
         "description": "Your cat's age in years."
      },
      "declawed": {
         "type": "boolean"
      }
   }
}
```

Podemos então validar se o nosso JSON está de acordo com o JSON Schema (Exemplo 4.4).

Exemplo 4.4 – Este JSON está de acordo com o nosso JSON Schema para "Cat"

```
{
```

```
  "name": "Fluffy",
  "age": 2,
  "declawed": false
}
```

Anteriormente, afirmei que um JSON Schema pode responder às seguintes perguntas:

Os tipos de dados dos valores estão corretos?
Podemos especificar se um valor deve ser um número, uma string etc.

Essas informações incluem os dados obrigatórios?
Podemos especificar quais dados são obrigatórios e quais não são.

Os valores estão no formato exigido?
Podemos especificar intervalos, mínimos e máximos.

Com esse exemplo bem simples do gato, a primeira pergunta foi respondida. Pudemos confirmar que o JSON para o gato “Fluffy” tem os tipos corretos de dados para os valores de name (nome), age (idade) e declawed (sem unhas). Vamos responder à segunda pergunta: essas informações incluem os dados obrigatórios?

Quando solicitamos dados, geralmente há propriedades (ou campos) para os quais deve haver um valor e outras que são opcionais. Por exemplo, quando crio uma conta nova em um site de compras, devo preencher o formulário com o endereço de entrega. Esse formulário de endereço exige que eu forneça o meu nome, a rua, a cidade, o estado e o CEP. Opcionalmente, posso incluir o nome de uma empresa, um número de apartamento e uma segunda linha com um endereço residencial. Se eu deixar algum campo obrigatório em branco, não poderei prosseguir com a criação da conta.

Para usar essa lógica de campos obrigatórios no JSON Schema, acrescentamos um quarto par nome-valor após `"$schema"`, `"title"` e `"properties"`. Esse par nome-valor tem o nome `"required"` e um valor cujo tipo do dado é um array. O array inclui os campos obrigatórios.

No exemplo 4.5, inicialmente acrescentamos outro campo para `"description"`. Em seguida, adicionamos um quarto par nome-valor, `"required"`, com um array contendo os valores obrigatórios como o valor desse array. `"name"`, `"age"` e

"declawed" são obrigatórios, portanto foram adicionados a essa lista. Deixamos "description" de fora, pois ele não é obrigatório.

EXEMPLO 4.5 – DEFININDO OS CAMPOS OBRIGATÓRIOS

```
{
   "$schema": "http://json-schema.org/draft-04/schema#",
   "title": "Cat",
   "properties": {
      "name": {
         "type": "string"
      },
      "age": {
         "type": "number",
         "description": "Your cat's age in years."
      },
      "declawed": {
         "type": "boolean"
      },
      "description": {
         "type": "string"
      }
   },
   "required": [
      "name",
      "age",
      "declawed"
   ]
}
```

Com a adição de "required" em nosso JSON Schema, o JSON do exemplo 4.6 será válido. Esse JSON está de acordo com o nosso JSON Schema para "Cat", com os campos obrigatórios "name", "age" e "declawed". Incluímos o par nome-valor "description", que é opcional.

EXEMPLO 4.6 – JSON VÁLIDO

```
{
   "name": "Fluffy",
   "age": 2,
```

```
  "declawed": false,
  "description" : "Fluffy loves to sleep all day."
}
```

Podemos também deixar o campo "description" de fora, pois ele não está incluído na lista de campos obrigatórios. O JSON do exemplo 4.7 está de acordo com o nosso JSON Schema para "Cat", com os campos obrigatórios "name", "age" e "declawed".

EXEMPLO 4.7 – JSON VÁLIDO SEM O CAMPO "DESCRIPTION"

```
{
  "name": "Fluffy",
  "age": 2,
  "declawed": false
}
```

É importante observar que, se o par nome-valor "required" não for incluído em seu JSON Schema com o array contendo os nomes obrigatórios, nada será obrigatório. Um objeto JSON sem nenhum par nome-valor será considerado válido. Sem o array em "required", o JSON do exemplo 4.8 será considerado válido para o JSON Schema "Cat".

EXEMPLO 4.8 – JSON VÁLIDO

```
{}
```

A terceira e última pergunta que podemos responder com o nosso JSON Schema é: os valores estão no formato exigido? Respondemos à pergunta sobre os tipos de dados de nossos valores, porém, em geral, devemos ter um formato específico para o tipo. Por exemplo, posso exigir um nome de usuário, porém esse nome não deverá exceder vinte caracteres. Além disso, posso pedir que você pense em um número entre 10 e 100. Podemos expressar esses requisitos específicos em nosso JSON Schema.

No JSON para o gato, temos requisitos como o nome ser uma string e a idade ser um número. Entretanto, não queremos que alguém nos forneça dados com um nome realmente longo para o gato, um nome excessivamente curto ou um número negativo para a idade do gato. Em nosso JSON Schema, podemos definir um tamanho mínimo e máximo para uma string e um valor mínimo

para um número.

No exemplo 4.9, uma validação foi incluída para garantir que o nome do gato tenha no mínimo três e no máximo vinte caracteres. Além do mais, garantimos que a idade submetida para o gato não seja um número negativo.

EXEMPLO 4.9 – VALIDAÇÃO DO JSON PARA O GATO

```
{
   "$schema": "http://json-schema.org/draft-04/schema#",
   "title": "Cat",
   "properties": {
      "name": {
         "type": "string",
         "minLength": 3,
         "maxLength" : 20
      },
      "age": {
         "type": "number",
         "description": "Your cat's age in years.",
         "minimum" : 0
      },
      "declawed": {
         "type": "boolean"
      },
      "description": {
         "type": "string"
      }
   },
   "required": [
      "name",
      "age",
      "declawed"
   ]
}
```

O JSON do exemplo 4.10 não é válido para o JSON Schema "Cat", pois o valor do nome excede "maxLength" e o valor da idade é menor que "minimum".

EXEMPLO 4.10 – JSON INVÁLIDO

```
{
   "name": "Fluffy the greatest cat in the whole wide world",
   "age": -2,
   "declawed": false,
   "description" : "Fluffy loves to sleep all day."
}
```

O JSON do exemplo 4.11 é válido com o JSON Schema "Cat" e está de acordo com os requisitos para os valores.

EXEMPLO 4.11 – ESTE JSON É VÁLIDO

```
{
   "name": "Fluffy",
   "age": 2,
   "declawed": false,
   "description" : "Fluffy loves to sleep all day."
}
```

Se voltarmos para a comparação de um esquema com um contrato, poderemos ver que os detalhes de nosso contrato podem ser bastante específicos. Os exemplos fornecidos neste capítulo são introdutórios e representam somente a ponta do iceberg. O JSON Schema aceita inclusive expressões regulares (padrões de caracteres, por exemplo, o formato de um endereço de email) e enumeradas (uma lista de valores possíveis). Se quiser dominar o JSON Schema, acesse as páginas a seguir, em que você poderá encontrar links para as especificações:

- A página inicial do JSON Schema (*http://json-schema.org/*)
- Especificação da validação com o JSON Schema (*http://json-schema.org/latest/json-schema-validation.html*)

Há uma lista extensa e crescente de bibliotecas, projetos para linguagens de programação específicas e frameworks que usam JSON Schema. Uma pesquisa rápida no Google em busca de “JSON Schema Validation [insira o nome da linguagem de programação aqui]” deverá fornecer os resultados necessários se você quiser incluir a validação com JSON Schema em seu projeto. Além disso, há algumas ferramentas de validação online que são independentes de linguagem de programação e são ótimas para realizar

experimentos com o JSON Schema:

- JSON Schema Lint (*http://jsonschemalint.com/draft4/*)
- JSON Schema Validator (*http://www.jsonschemavalidator.net/*)

Se você acessar o site do JSON Schema Lint, duas áreas de texto serão apresentadas: uma para o JSON Schema e outra para o documento JSON a ser validado. Se eu colar o esquema do exemplo 4.9 e o JSON do exemplo 4.10, verei os seguintes erros:

- Field: data.name, Error: has longer length than allowed, Value: “Fluffy the greatest cat in the whole wide world”

(Campo: data.name, Erro: tamanho maior que o permitido, Valor: “Fluffy the greatest cat in the whole wide world”)

- Field: data.age, Error: is less than minimum, Value: -2

(Campo: data.age, Erro: menor que o mínimo, Valor: -2)

Se acessar o site do JSON Schema Validator, também verei as mesmas duas áreas de texto. Novamente, se eu colar o esquema do exemplo 4.9 e o JSON do exemplo 4.10, verei os erros. Além disso, as linhas numeradas do JSON exibirão um x vermelho, mostrando em que ponto estão os erros no JSON.

- Message: String ‘Fluffy the greatest cat in the whole wide world’ exceeds maximum length of 20, Schema Path: #/properties/name/maxLength

(Mensagem: String ‘Fluffy the greatest cat in the whole wide world’ excede o tamanho máximo de 20, Path no esquema: #/properties/name/maxLength)

- Message: Integer -2 is less than minimum value of 0, Schema Path: #/properties/age/minimum

(Mensagem: Inteiro -2 é menor que o valor mínimo de 0, Path no esquema: #/properties/age/minimum)

O JSON Schema Validator não só destaca os números das linhas em que o erro ocorreu, como também nos fornece os paths dos requisitos do esquema que estão provocando falhas na validação. As duas ferramentas de validação podem ter descrito os erros de forma um pouco diferente, mas ambas identificaram os mesmos erros.

Termos e conceitos essenciais

Este capítulo discutiu os seguintes termos essenciais:

JSON Schema
: Um contrato virtual para intercâmbio de dados.

Também discutimos os seguintes conceitos essenciais:

- Uma ferramenta de validação de JSON faz uma validação de sintaxe, enquanto o JSON Schema provê uma validação de conformidade.
- O JSON Schema pode servir como a primeira linha de defesa na aceitação dos dados ou como uma ferramenta para economia de tempo (poupando a nossa sanidade mental) para a parte que está disponibilizando os dados, pois garante que seus dados estarão de acordo com o que pode ser aceito.
- Um JSON Schema é capaz de responder às três perguntas a seguir para fazer uma validação de conformidade:

  Os tipos de dados dos valores estão corretos?
  : Podemos especificar se um valor deve ser um número, uma string etc.

  Essas informações incluem os dados obrigatórios?
  : Podemos especificar quais dados são obrigatórios e quais não são.

  Os valores estão no formato exigido?
  : Podemos especificar intervalos, mínimos e máximos.

CAPÍTULO 5

Preocupações relacionadas à segurança com JSON

O JSON sozinho não representa uma grande ameaça. Afinal de contas, ele é apenas um formato para intercâmbio de dados. Por si só, ele é somente um documento ou uma sequência de dados. As verdadeiras preocupações relacionadas à segurança com o JSON surgem devido à maneira como ele é usado. Neste capítulo, daremos uma olhada em duas das principais preocupações de segurança com o JSON na Web: Cross-site Request Forgery e Cross-site Scripting.

Antes de prosseguirmos na discussão sobre as preocupações relacionadas à segurança e passarmos para os capítulos restantes deste livro, devemos entender qual é a relação entre o lado cliente e o lado servidor. Vamos dar uma olhada rapidamente nesses relacionamentos para aqueles que ainda não compreendem esse conceito.

Uma olhada rápida nos relacionamentos entre o lado cliente e o lado servidor

Ao chegar no Pierre's Fine Dining para jantar, sento-me em uma mesa agradável, com guardanapos dobrados em formato de cisnes. Um homem alto usando calças sociais e uma bela camisa aproxima-se da mesa e diz: "Meu nome é Thomas e estou aqui para servi-la esta noite." Depois que me reconhece, ele abaixa seu tom de voz e diz: "A propósito, você é uma das minhas clientes favoritas." Ele levanta suas sobrancelhas e diz: "Aqui está o nosso menu especial."

Depois de dar uma olhada no menu especial, digo a Thomas o que quero para

o jantar e, após um tempo, ele traz o prato para a mesa. Tenho um jantar prazeroso e, em seguida, pago 200 dólares pelos pratos artisticamente dispostos com pouquíssima comida. Essa foi mais uma noite agradável no Pierre's Fine Dining, em que Thomas foi o servidor e eu, a cliente.

Seu navegador de Internet tem um relacionamento com os sites, assim como eu tenho um relacionamento com o Pierre's Fine Dining. Esse relacionamento é composto de um mundo movimentado de *solicitações* e de *respostas*. Faço o pedido de um prato e a cozinha responde preparando o prato específico que pedi e envia-o para que seja entregue a mim.

Quando você acessa o seu site predileto para dar uma olhada em fotos de lindos gatinhos, o navegador de Internet em seu computador é o cliente e o computador que hospeda as fotos dos lindos gatinhos é o servidor. Seu navegador de Internet faz uma solicitação que viaja pela Internet até o computador responsável pelo site com as fotos dos lindos gatinhos que recebo. O site com os belos gatinhos então serve a página em uma resposta que é enviada de volta pela Internet. Em seguida, seu navegador apresenta a página na tela para você.

Nesse relacionamento, traçamos uma linha na areia e dizemos que tudo o que é enviado como resposta pelo site com as fotos dos lindos gatinhos e que será tratado pelo navegador é chamado de código do lado cliente. No exemplo do restaurante, a resposta seria o prato de comida, e a mesa em que estou sentada seria o navegador de Internet. A mesa recebe o prato de comida, de modo que agora posso vê-lo e consumi-lo.

Dizemos então que tudo o que aconteceu antes de essa página de resposta ter sido enviada – essencialmente a criação dela – corresponde ao código do lado servidor. No exemplo do restaurante, o código do lado servidor seria tudo o que acontece na cozinha. Jamais vou até a cozinha e não vejo o que eles fazem lá para preparar o meu prato. A diferença sutil entre o exemplo do restaurante e o mundo real está no fato de o servidor não ser a pessoa que está correndo da cozinha para a mesa e vice-versa. A cozinha é o servidor e essa pessoa é a Internet.

O público não vê o que o site com as fotos dos lindos gatinhos faz em sua cozinha metafórica. O site pode usar PHP, ASP.NET ou qualquer outra

linguagem de programação. O que estiver sendo usado em sua cozinha não é importante para o meu navegador, desde que o site entregue uma resposta com o código do lado cliente.

A resposta obtida do site é uma combinação de HTML, CSS e JavaScript. Assim como posso inspecionar tudo na mesa do restaurante, posso pressionar F12 em meu navegador e, com as ferramentas de desenvolvedor, posso ver todo o HTML, o CSS e o JavaScript da página. Posso até mesmo ver o código JavaScript daquele popup irritante que fica piscando com os gatinhos dançando.

O lado cliente corresponde a tudo o que ocorre em um navegador de Internet do usuário, enquanto o lado servidor é tudo o que acontece no servidor em que o site está hospedado. Quando o código do lado cliente é referenciado, normalmente isso quer dizer JavaScript, HTML ou CSS. Quando o código do lado servidor é referenciado, geralmente isso implica linguagens do lado do servidor, como ASP.NET, Ruby on Rails ou Java.

Vamos agora dar uma olhada nas preocupações relacionadas à segurança, que é um assunto importante.

Cross-Site Request Forgery (CSRF)

O CSRF (Cross-site Request Forgery) é um exploit (exploração de falhas) que tira proveito da confiança de um site no navegador de Internet de um usuário. As vulnerabilidades de CSRF estão presentes há bastante tempo, muito antes de o JSON ter surgido.

Um exemplo de um exploit CSRF com JSON seria algo como o seguinte.

Você faz login no site de um banco. Esse site tem um URL JSON que inclui algumas informações confidencias sobre você (Exemplo 5.1).

EXEMPLO 5.1 – SUAS INFORMAÇÕES CONFIDENCIAIS NO FORMATO JSON

```
[
  {
    "user": "bobbarker"
  },
  {
```

```
    "phone": "555-555-5555"
  }
]
```

Você poderá pensar: “Ei, estão faltando chaves nesse JSON!” Esse é um JSON válido. É um JSON perigosamente válido, pois é também um script JavaScript válido. Ele é chamado de *array JSON de nível mais alto* (top-level JSON array).

O site bancário de exemplo utiliza autenticação e cookies de sessão para garantir que essas informações sejam fornecidas somente a você – o usuário que fez login e que está registrado.

Uma pessoa mal-intencionada nesse exemplo descobre o URL para os dados JSON confidenciais no site do banco e o coloca em uma tag <script> em *seu* próprio site. Se não souber exatamente o que é uma tag <script>, você poderá encontrá-las nos bastidores da maioria das páginas web modernas. Isso é conhecido como code-behind (código por trás) e é um documento HTML. Se acessar o seu site favorito, clicar com o botão direito do mouse e selecionar a opção para visualizar o código-fonte, você verá uma ou mais tags semelhantes àquela mostrada no exemplo 5.2.

EXEMPLO 5.2 – EXEMPLO DE UMA TAG <SCRIPT> (O ATRIBUTO “SRC” DESSA TAG ESPECIFICA A LOCALIZAÇÃO DO SCRIPT)

```
<script src="https://code.jquery.com/jquery-2.1.4.min.js"></script>
```

Os navegadores de Internet têm regras para compartilhamento entre sites com domínios diferentes (*http://domainone.com* e *http://domaintwo.com* são domínios diferentes). A pessoa mal-intencionada utiliza a tag <script> porque ela é isenta dessas regras de compartilhamento (Exemplo 5.3). Geralmente, será necessário e normal usar um script hospedado por outro site, portanto a tag <script> constitui uma exceção. O JSON usa um array de nível mais alto que o transforma em um JavaScript válido, portanto ele consegue sair ileso.

EXEMPLO 5.3 – EXEMPLO DA APARÊNCIA DE UMA TAG <SCRIPT> NO SITE DA PESSOA MAL-INTENCIONADA

```
<script src="https://www.yourspecialbank.com/user.json"></script>
```

Um aspecto fundamental para isso funcionar está no seu relacionamento com

o site do banco. Sem esse relacionamento, o link na tag <script> não retornará dados confidenciais. Esse link não hospeda simplesmente os seus dados confidenciais. É um link dinâmico que retorna os dados confidenciais da pessoa que estiver logada. Ao fazer login junto ao banco, você iniciou um relacionamento em que o banco acredita que você é quem você diz ser.

Esse exploit depende dessa confiança. Para que a pessoa mal-intencionada explore essa confiança, ela precisará que você acesse o site dela contendo a tag <script> roubada enquanto você estiver logado em seu banco. Para isso, a pessoa mal-intencionada pode enviar um milhão de emails às pessoas dizendo que "há uma mensagem importante para você no site de seu banco". Esses emails geralmente são formatados exatamente como os emails do banco, com os quais as pessoas têm familiaridade (ou do site em que a exploração de falhas está sendo feita). Se as pessoas não verificam os cabeçalhos dos emails para ver a sua proveniência ou se não passam o mouse sobre o link para questionar se o verdadeiro domínio dos sites em que elas confiam será acessado, é provável que elas cliquem nesse link.

Como exemplo, vamos supor que você estivesse doente e que não estava pensando com clareza; sendo assim, você clicou no link. Além disso, você não fez logout do site de seu banco da última vez que o acessou, portanto a sua sessão continua existindo. Nesse momento, você está em um estado de relacionamento confiável com o banco. Depois que a página do site da pessoa mal-intencionada for carregada, talvez você perceba que tenha acessado um local estranho e saia. Porém, será tarde demais. O site da pessoa mal-intencionada será capaz de obter os dados JSON confidenciais e poderá enviá-los para os seus próprios servidores para serem armazenados. Ai.

O que o banco e os desenvolvedores web poderiam fazer de modo diferente para evitar um exploit CSRF?

Para começar, o banco poderia ter transformado o array no valor de um objeto JSON. Isso fará com que os dados JSON sejam um JavaScript inválido. Veja o exemplo 5.4.

EXEMPLO 5.4 – AO COLOCAR O ARRAY DENTRO DE UM OBJETO, ELE NÃO SERÁ MAIS UM JAVASCRIPT VÁLIDO, POSSÍVEL DE SER CARREGADO EM UMA TAG <SCRIPT>

```
{
```

```
  "info": [
    {
      "user": "bobbarker"
    },
    {
      "phone": "555-555-5555"
    }
  ]
}
```

Além disso, se o banco tivesse permitido somente solicitações POST em vez de permitir solicitações GET para obter os dados JSON confidenciais, a pessoa mal-intencionada não poderia ter usado o link em seu URL. GET e POST são dois métodos HTTP usados para comunicação com o servidor. GET serve para fazer uma solicitação de dados e pode retornar uma resposta. POST é usado na submissão de dados e também pode retornar uma resposta. Se um servidor permitir uma solicitação GET para um link, o link poderá ser feito diretamente em um navegador ou em uma tag <script>. No entanto, o link não pode ser feito diretamente com POST. Sem a conveniência da tag <script>, a pessoa mal-intencionada estaria de mãos atadas por causa das políticas de compartilhamento de recursos, impedindo-a de executar outras tarefas no lado cliente que possam enganar o banco fazendo-o confiar nela.

Isso não quer dizer que o intercâmbio de dados HTTP com JSON deva ser totalmente limitado ao método POST. Uma boa regra geral para decidir se as solicitações GET devem ou não ser permitidas para uma página ou um recurso consiste em perguntar: essa página deverá ser acessada diretamente pelo URL ou usada em uma tag <script>? Se a resposta for “não”, o método GET não deverá ser permitido para evitar que uma pessoa qualquer a acesse por meio de um URL ou da tag <script>.

Dados *confidenciais* também são fundamentais nesse exploit. Se o JSON contiver somente uma lista de espécies de pássaros, a pessoa mal-intencionada provavelmente não criará um site para tentar roubar esses dados. No entanto, ao preparar-se contra ameaças à segurança, os hábitos são importantes. Se você não adquirir o hábito de usar arrays de nível mais alto em seu JSON e não desenvolver o hábito de usar GET no lugar de POST, você

não estará sujeito a ser aquela pessoa que escreverá um código responsável por um exploit de grandes proporções, deixando uma multidão de clientes zangados.

Ataques de injeção de código

Os ataques de injeção de código podem assumir diversas formas. Em última instância, eles dependem de encontrar brechas de segurança passíveis de exploração. O ataque CSRF que acabamos de discutir é um ataque que depende da confiança. Um *ataque de injeção de código* é um ataque que depende da capacidade de injetar um código malicioso em um site inocente.

Cross-site Scripting (XSS)

Os ataques XSS (Cross-site Scripting) são um tipo de ataque de injeção de código. Uma brecha de segurança frequente com o JSON ocorre quando o JavaScript busca uma string com JSON e a transforma em um objeto JavaScript.

Lembre-se de que o JSON por si só é apenas texto. Em programação, se quisermos fazer algo útil com essa representação textual de um objeto, ela deverá ser carregada na memória na forma de um objeto. Esses dados poderão então ser manipulados, inspecionados e usados na lógica de programação.

Em JavaScript, uma maneira de fazer isso é por meio de uma função chamada eval(), que aceita uma string, compila e então a executa.

No exemplo 5.5, o código JavaScript utiliza a função eval() para carregar o objeto animal/cat na memória. As propriedades do objeto podem então ser acessadas pelo código. O alerta na terceira linha fará um popup em que se lê "cat" ser apresentado no navegador.

EXEMPLO 5.5 – ACESSANDO AS PROPRIEDADES DE UM OBJETO

```
var jsonString = '{"animal":"cat"}';
var myObject = eval("(" + jsonString + ")");
alert(myObject.animal);
```

O exemplo 5.5 é relativamente inofensivo porque o JSON está diretamente

definido no código. Normalmente, o JSON será proveniente de outro servidor. Esse servidor, em geral, será um servidor de terceiros, sobre o qual você não terá nenhum controle. Por exemplo, se pedisse algum dado JSON ao servidor do Facebook, eu não teria nenhum controle sobre o JSON que ele me disponibilizaria. Se o servidor tiver uma falha explorada ou, de alguma forma, o JSON for interceptado, poderei receber um código malicioso.

O problema com a função eval() está no fato de ela aceitar uma string, compilando-a e executando-a de maneira indiscriminada. Se o meu JSON for proveniente de um servidor de terceiros e for substituído por um script malicioso, meu site totalmente inocente compilará e executará esse código malicioso nos navegadores de Internet das pessoas que visitarem o meu site.

No código JavaScript do exemplo 5.6, substituí a string JSON por um pouco de código JavaScript. Quando esse código for executado, a função eval() exibirá um alerta em que se lê “this is bad”.

EXEMPLO 5.6 – UM ALERTA DA FUNÇÃO EVAL()

```
var jsonString = "alert('this is bad')";
var myObject = eval("(" + jsonString + ")");
alert(myObject.animal);
```

À medida que o JSON evoluiu ao longo dos anos, essa vulnerabilidade foi reconhecida. A função JSON.parse() lida com essa vulnerabilidade pelo fato de ser seletiva. Essa função somente fará parse de JSON e não executará scripts. No exemplo 5.7, eval() foi substituída por JSON.parse().

EXEMPLO 5.7 – SUBSTITUINDO EVAL() POR JSON.PARSE()

```
var jsonString = '{"animal":"cat"}';
var myObject = JSON.parse(jsonString);
alert(myObject.animal);
```

Em desenvolvimento web, outra preocupação que geralmente tem o mesmo peso que as questões relacionadas à segurança é o suporte a diferentes navegadores. À medida que a Web evolui, aqueles que hospedam sites devem decidir quais navegadores e quais versões eles devem tratar. Essencialmente, eles devem decidir quem será deixado para trás. Sempre haverá pessoas que não atualizam seus navegadores de Internet ou que usam um navegador de

Internet menos popular, que não acompanha os padrões em evolução.

A função JSON.parse(), muito mais segura, atualmente é aceita pela maioria dos navegadores de Internet em suas versões mais recentes. No entanto, algumas versões mais antigas de navegadores de Internet usadas por um pequeno percentual de usuários continuam ativas e não aceitam essa função. Se JSON.parse() for usada, esse pequeno percentual de usuários não poderá utilizar as funcionalidades do site que dependam do JSON. Com frequência, esse problema é tratado por meio da definição de maneiras elegantes de falhar. Por exemplo, em vez de ter uma página web cheia de erros evidentes, considere uma página que capture esses erros e exiba uma mensagem como "Please update your browser to the latest version" (Por favor, atualize o seu navegador para a versão mais recente).

Brechas de segurança: decisões relacionadas à arquitetura

No início do capítulo, eu disse que "o JSON sozinho não representa uma grande ameaça". Isso continua valendo, porém há alguns cenários em que uma ameaça pode ser diretamente incluída nos dados JSON, embora esses dados continuem sendo válidos.

Vamos dar uma olhada em um JSON perfeitamente inocente (Exemplo 5.8).

EXEMPLO 5.8 – JSON PERFEITAMENTE INOCENTE

```
{
   "message": "hello, world!"
}
```

Agora vamos supor que eu hospede um site que armazene mensagens em um banco de dados e as exiba em uma página web para um usuário ler. Jamais ouvi falar desse exploit e, sendo assim, tenho uma página em que um usuário pode enviar uma mensagem contendo qualquer informação a outro usuário. Em minha página de mensagens, solicito a string JSON do servidor e, no lado cliente, utilizo a função JavaScript eval() para converter a resposta com a string JSON em um objeto JavaScript. Uso esse objeto JavaScript no código do lado cliente para exibir o valor da mensagem diretamente no HTML.

Vamos dar uma olhada em um JSON um pouco menos inocente no Exemplo 5.9.

EXEMPLO 5.9 – JSON NÃO TÃO INOCENTE

```
{
   "message": "<div onmouseover=\"alert('gotcha!')\">hover here.</div>"
}
```

O JSON não tão inocente do exemplo inclui JavaScript. Apesar de esse JavaScript estar incluído no par nome-valor do JSON, ele é simplesmente uma string de texto. Esse é um JSON perfeitamente válido e há muitos locais em que esse JSON poderia ser usado e não seria uma ameaça.

Entretanto, o JavaScript do exemplo, quando exibido no HTML da página de mensagens, será uma ameaça. O "JSON não tão inocente" mostrado aqui fará um alerta ser apresentado com a mensagem "gotcha!" sempre que o usuário passar o cursor do mouse sobre a mensagem na tela. O problema é que algo muito pior poderia ser feito além da apresentação de um alerta. Uma pessoa mal-intencionada poderia incluir um script para acessar todas as suas mensagens privadas da página e enviá-las para o seu próprio servidor para lê-las.

O que eu poderia ter feito para evitar isso? Para começar, eu poderia ter tomado medidas para impedir o uso de HTML em minhas mensagens. Isso pode envolver uma validação tanto do lado cliente quanto do lado servidor. Além disso, eu poderia garantir que qualquer caractere HTML incluído na mensagem seja escapado de modo que os caracteres HTML como sejam exibidos como <div> na página (<div> não funcionará como um HTML válido). Todas essas medidas serão específicas ao código do lado cliente e do lado servidor usados em minha arquitetura.

A maioria dos ataques de injeção de código envolve uma arquitetura que não tenha sido criada com uma pergunta importante em mente: como uma pessoa mal-intencionada poderá explorar isso? A decisão de permitir o uso de HTML no JSON e de exibir os valores diretamente na página foi uma decisão aparentemente inocente relacionada à arquitetura. Para evitar ataques de injeção de código, o segredo é pensar nos exploits possíveis e executar os passos adicionais (e às vezes árduos) para evitá-los.

Termos e conceitos essenciais

Este capítulo discutiu os seguintes termos essenciais:

Lado servidor (em desenvolvimento web)

As operações que ocorrem nos bastidores, ou seja, no servidor em que uma página ou um recurso está sendo solicitado. O servidor disponibiliza a resposta que será processada e/ou carregada pelo navegador de Internet.

Lado cliente (em desenvolvimento web)

As operações que ocorrem no navegador de Internet a partir do ponto em que uma página solicitada é carregada. Normalmente, elas são constituídas de HTML, CSS e JavaScript.

Cross-Site Request Forgery (CSRF)

Um exploit (exploração de falha) que tira proveito da confiança de um site no navegador de um usuário.

Array JSON de nível mais alto (top-level JSON array)

Um array JSON que existe externamente a um par nome-valor e que está no nível mais alto do documento.

Ataque de injeção de código

Um ataque que depende da injeção de dados em uma aplicação web para facilitar a execução ou a interpretação de dados maliciosos.

Ataque Cross-site Scripting (XSS) com JSON

Um tipo de ataque de injeção de código que tira proveito de um site inocente ao interceptar ou substituir dados JSON disponibilizados por terceiros ao site por um script malicioso.

Também discutimos os seguintes conceitos essenciais:

- O JSON por si só não é uma ameaça. Ele é constituído somente de texto.
- Três aspectos que devem ser lembrados e que levam em consideração as questões relacionadas à segurança com JSON:
  - Não utilize arrays de nível mais alto. Os arrays de nível mais alto constituem um JavaScript válido ao qual um link poderá ser feito por

meio de uma tag <script>, permitindo que esse código seja usado.

– Utilize HTTP POST no lugar de GET para dados JSON que não sejam voltados para o público.

Uma solicitação HTTP GET pode ser usada como um link em um URL e pode ser colocada em uma tag <script>.

– Utilize JSON.parse() no lugar de eval(). A função eval() compilará e executará a string passada para ela, o que pode deixar o seu código suscetível a ataques. JSON.parse() faz parse somente de JSON.

• As brechas de segurança geralmente são introduzidas por meio de decisões relacionadas à arquitetura quando a pergunta básica “Como uma pessoa mal-intencionada poderá explorar isso?” não é feita.

CAPÍTULO 6

XMLHttpRequest do JavaScript e APIs web

Pode ser difícil falar XMlHttpRequest do JavaScript e APIs web três vezes rapidamente, porém elas não são tão complicadas quanto parecem. Elas representam um relacionamento simples entre cliente e servidor. O XMLHttpRequest do JavaScript corresponde ao cliente fazendo solicitações, enquanto a API web é o servidor enviando as respostas.

No exemplo cliente-servidor do restaurante discutido no capítulo anterior, referi-me ao servidor como a cozinha e ao cliente como a pessoa que frequenta o restaurante. Este capítulo focará um tipo de cozinha e o modo como essa cozinha funciona.

Não é preciso dizer que nem todos os restaurantes funcionam da mesma maneira. Em alguns restaurantes, você poderá se dirigir a um balcão para fazer o seu pedido. Alguns restaurantes são abertos ao público, enquanto outros podem funcionar somente como uma lanchonete nas dependências de uma grande empresa.

Um relacionamento cliente-servidor do qual a maioria de nós participa é a navegação na Internet. Com frequência, pensamos em nós mesmos como se estivéssemos viajando ou explorando a Internet. Na verdade, geralmente ficamos sentados, olhando para uma tela. O navegador de Internet também não vai a lugar nenhum. Ele simplesmente fica parado, como se estivesse na mesa de um restaurante, fazendo solicitações e recebendo respostas. Depois que o servidor nos dá a resposta, ele terá concluído o seu trabalho conosco e prosseguirá para outra entidade que estiver fazendo uma solicitação.

A solicitação feita pelo navegador de Internet pede um recurso. Quando estamos “surfando pela Internet”, clicamos em um link com um URL ou

digitamos um URL diretamente em nosso navegador. URL quer dizer Universal *Resource* Locator (Localizador-padrão de recursos). Os URLs que usamos em nossa navegação pela Internet servem para localizar recursos HTML, o que nos permite ver os sites, incluindo aquele com as fotos de lindos gatinhos. Nesse cenário, o recurso que estamos pedindo tem um tipo de conteúdo igual a text/html.

Outro relacionamento cliente-servidor que não envolve diretamente os seres humanos é constituído de uma API web. Uma API web serve um conteúdo por meio de HTTP, assim como um site, porém não foi feita para os olhares humanos. Podemos pensar nela como em um restaurante cujos clientes sejam códigos, pois a maioria das solicitações a esses servidores é feita por códigos.

O código de programação geralmente não solicita imagens de lindos gatinhos para ver como nós fazemos. Seu apetite normalmente está associado à solicitação de dados. Este capítulo discute um tipo de cliente que solicita dados JSON (um recurso com um tipo de conteúdo igual a application/json) e o tipo de restaurante que atende a esses clientes: as APIs web.

Vamos começar dando uma olhada nas APIs web e no papel do JSON nessas APIs.

APIs web

De modo diferente dos seres humanos, o código não tem um par de olhos que queira ler um artigo ou ver uma figura. O código deve garantir que “algo” esteja em um formato que possa ser lido (que possibilite fazer parse). É aqui que um formato para intercâmbio de dados (como o JSON mostrado no exemplo 6.1) entra em cena.

EXEMPLO 6.1 – DADOS JSON SOBRE O CLIMA, DA API WEB OPENWEATHERMAP

```
{
   "dt": 1433383200,
   "temp": {
      "day": 293.5,
      "min": 293.5,
      "max": 293.5,
      "night": 293.5,
```

```
      "eve": 293.5,
      "morn": 293.5
   },
   "pressure": 1015.06,
   "humidity": 98,
   "weather": [
      {
         "id": 802,
         "main": "Clouds",
         "description": "scattered clouds",
         "icon": "03n"
      }
   ],
   "speed": 2.86,
   "deg": 134,
   "clouds": 44
}
```


O exemplo de dados JSON sobre o clima pode ser “lido” por qualquer código que seja capaz de fazer parsing de JSON. Esse recurso JSON pode ser solicitado por meio de um URL (o código do exemplo é um subconjunto do documento JSON completo):

http://api.openweathermap.org/data/2.5/forecast/daily?lat=35&lon=139&cnt=10&mode=json

Embora muitas APIs web públicas como a API OpenWeatherMap sejam para “leitura”, várias delas, como a API do PayPal, são mais interativas. Uma *API web* é um conjunto de instruções e de padrões para interagir com um serviço por meio de HTTP. Essa interação pode incluir as operações CRUD (Create, Read, Update e Delete, ou Criar, ler, atualizar e apagar) e a API web terá uma referência que define essas instruções e padrões.

Por exemplo, de acordo com a referência da API do PayPal, posso criar uma nova fatura com essa API ao postar dados JSON para o URL:

https://api.sandbox.paypal.com/v1/invoicing/invoices

No exemplo 6.2, temos o JSON que representa uma fatura a ser enviada como uma solicitação à API do PayPal:

Exemplo 6.2 – Uma fatura para a API do PayPal

```
{
   "merchant_info": {
      "email": "bob@bob.com",
      "first_name": "Bob",
      "last_name": "Bobberson",
      "business_name": "Bob Equipment, LLC",
      "phone": {
         "country_code": "001",
         "national_number": "5555555555"
      },
      "address": {
         "line1": "123 Fake St.",
         "city": "Somewhere",
         "state": "OR",
         "postal_code": "97520",
         "country_code": "US"
      }
   },
   "billing_info": [
      {
         "email": "someguy@someguy.com"
      }
   ],
   "items": [
      {
         "name": "Widgets",
         "quantity": 20,
         "unit_price": {
            "currency": "USD",
            "value": 89
         }
      }
   ],
   "note": "Special Widgets Order!",
   "payment_term": {
      "term_type": "NET_45"
```

```
  },
  "shipping_info": {
    "first_name": "Some",
    "last_name": "Guy",
    "business_name": "Not applicable",
    "address": {
      "line1": "456 Real Fake Dr",
      "city": "Some Place",
      "state": "OR",
      "postal_code": "97501",
      "country_code": "US"
    }
  }
}
```

Com a API do PayPal, após a fatura ter sido criada, poderei solicitar (ler), atualizar e apagar essa fatura.

As operações executadas nos bastidores em JavaScript, por exemplo, a solicitação aos dados de clima, são chamadas de assíncronas. As operações assíncronas são operações que ocorrem em background, sem a interrupção da transmissão principal.

No caso das operações JavaScript assíncronas, a “transmissão principal” corresponde ao que é exibido no navegador web. Por exemplo, uma página web de notícias poderia incluir uma barra lateral com dados do clima em tempo real. Enquanto você lê seus artigos de notícia, o código em background poderá atualizar assincronamente a exibição dos dados do clima a cada 60 segundos. Essa operação não exigirá que a página seja recarregada nem interromperá a rolagem de sua página durante a leitura de seu artigo. O único elemento que mudará na página será a barra lateral com os dados do clima.

As operações assíncronas (em background) do JavaScript são chamadas de AJAX. AJAX quer dizer Asynchronous JavaScript and XML (JavaScript Assíncrono e XML). Quando fazemos solicitações de JSON nos bastidores, tecnicamente, isso seria um Asynchronous JavaScript and JSON (AJAJ). No entanto, o termo “AJAX” vem sendo usado há tanto tempo que ele se tornou

menos um acrônimo e mais uma palavra para descrever *qualquer* operação assíncrona em JavaScript. Neste livro e em outros lugares, você verá o termo AJAX usado para descrever Asynchronous JavaScript and JSON. Isso não é decorrente de um erro.

Vamos dar uma olhada em como podemos usar o AJAX com o JSON e o XMLHttpRequest do JavaScript.

XMLHttpRequest do JavaScript

Embora o XMLHttpRequest do JavaScript passe a impressão de que ele tem a ver com XML, nós o usamos para fazer solicitações HTTP. Na época em que seu nome foi criado contendo a palavra XML, o XML era a estrela do formato para intercâmbio de dados para fazer esses tipos de solicitação. Entretanto, o XMLHttpRequest *não está* restrito ao XML. Apesar do nome, ele é usado para fazer solicitação de dados JSON.

Anteriormente, usei frases contendo uma expressão semelhante a "código que busca". O XMLHttpRequest do JavaScript é um código que busca um recurso. Essa forma de expressão é comum ao falar de solicitações HTTP, embora, na verdade, o código não percorra o ciberespaço como um cachorro atrás de uma bola. O código permanece onde está e, educadamente, pede que o recurso seja entregue.

Se retornarmos ao exemplo do restaurante, podemos imaginar o código JavaScript sentado em uma mesa do restaurante. O código JavaScript não vai até a cozinha para solicitar dados JSON nem grita pelo restaurante. Há um protocolo em um restaurante. Esse protocolo exige que você chame um garçom que anotará o seu pedido e o levará para a cozinha.

Há também um protocolo que permite que um código envie solicitações aos servidores web, que podem estar a centenas ou a milhares de quilômetros de distância. O próprio alicerce da comunicação de dados que nos permite acessar nossos sites favoritos é baseado em um protocolo chamado *HTTP* (*HyperText Transfer Protocol*, ou Protocolo de transferência de hipertexto). Quando digitamos *http://www.cutelittlekittens.com* em nosso navegador de Internet, usamos o Hypertext Transfer Protocol para fazer a solicitação de um recurso. No caso do site dos lindos gatinhos, esse recurso é uma página

HTML contendo imagens de lindos gatinhos. O código para fazer uma solicitação de um recurso JSON utiliza o mesmo protocolo (HTTP).

Em JavaScript, o código que usa o protocolo para fazer essa solicitação é o XMLHttpRequest. O JavaScript é uma linguagem orientada a objetos, portanto XMLHttpRequest naturalmente é um objeto. Depois que for criado por meio da sintaxe new XmlHttpRequest() (Exemplo 6.3) e atribuído a uma variável, esse objeto terá funções que poderão ser chamadas para solicitar recursos de alguma localidade.

EXEMPLO 6.3 – UM OBJETO XMLHTTPREQUEST DO JAVASCRIPT

```
var myXmlHttpRequest = new XMLHttpRequest();
```

XMLHttpRequest é um objeto e o exemplo 6.3 mostra a criação de um novo objeto XMLHttpRequest. Mesmo que não conheça JavaScript, se você leu o livro até este ponto, seu conhecimento de JSON ajudará a visualizar um objeto JavaScript. Afinal de contas, o JSON é baseado na notação de objetos literais do JavaScript.

Dizemos que um objeto JavaScript tem propriedades. Essas “propriedades” são simplesmente pares nome-valor. As propriedades de XMLHttpRequest são: onreadystatechange, readyState, response, responseText, responseType, responseXML, status, statusText, timeout, ontimeout, upload e withCredentials. Observe a inconsistência no uso de letras maiúsculas e minúsculas nessas propriedades. Algumas usam camelCase, enquanto outras usam somente letras minúsculas. Isso poderá deixar você de cabelo em pé no futuro se você decidir escrever o seu próprio código com XMLHttpRequest, pois você não só deverá se lembrar de quais propriedades são chamadas, como também das letras maiúsculas e minúsculas.

Uma das principais diferenças entre um objeto típico em programação e um objeto JSON está no fato de o JSON não ter instruções executáveis. O JSON é composto somente de propriedades, pois foi concebido para o intercâmbio de dados. Um objeto típico em programação também incluirá funções (ou métodos). Por exemplo, posso descrever meu calçado em JSON com pares nome-valor como: "color": "pink" e "brand": "crocs". O que não posso fazer em JSON é dar vida a esse calçado ao chamar uma função (ou método) como

shoe.walk().

As funções disponíveis em XMLHttpRequest em que estaremos mais interessados são:

- open(*method*, *url*, *async* (opcional), *user* (opcional), *password* (opcional))
- send()

As propriedades disponíveis em XMLHttpRequest em que estaremos mais interessados são:

onreadystatechange
: Contém o valor de uma função que podemos definir em nosso código.

readyState
: Retorna um valor no intervalo de 0 a 4 que representa um código de status.

status
: Retorna o código de status HTTP (por exemplo, 200 para sucesso).

responseText
: Quando a resposta for igual a sucesso, essa propriedade conterá o corpo da resposta na forma de texto (JSON, se for isso que solicitamos).

Para aqueles que não tiverem familiaridade com JavaScript, aqui vai uma informação para dar um nó em sua cabeça: uma propriedade pode ter uma função como valor. Isso é possível em JavaScript porque uma função é um objeto. Um objeto é um tipo de dado e, sendo assim, pode ser atribuído a uma variável (propriedade), modificado e passado como informação. No mundo da programação, elas são chamadas de funções de primeira classe. A propriedade onreadystatechange tem uma função como valor.

O exemplo 6.4 cria um novo objeto XMLHttpRequest e obtém dados JSON da API OpenWeatherMap.

EXEMPLO 6.4 – UM NOVO OBJETO XMLHTTPREQUEST

```
var myXMLHttpRequest = new XMLHttpRequest();
var url = "http://api.openweathermap.org/data/2.5/weather?lat=35&lon=139";

myXMLHttpRequest.onreadystatechange = function() {
   if (myXMLHttpRequest.readyState === 4
```

```
        && myXMLHttpRequest.status === 200) {
      var myObject = JSON.parse(myXMLHttpRequest.responseText);
      var myJSON = JSON.stringify(myObject);
   }
}
myXMLHttpRequest.open("GET", url, true);
myXMLHttpRequest.send();
```

Na segunda linha de código desse exemplo, criei uma variável que contém o URL para um recurso JSON. Em seguida, criei uma função e a atribuí à propriedade onreadystatechange de myXMLHttpRequest. Essa função será executada sempre que a propriedade readyState mudar. Na função, verifico se readyState é igual a 4 (código para "concluído") e se o status HTTP é 200 (código para sucesso). Se ambas as condições forem verdadeiras, faço parse do JSON e gero um objeto JSON.

Dois termos que você ouvirá com frequência quando o JSON é transformado em texto a partir de um objeto e depois de um texto de volta a um objeto são *serialização* e *desserialização*. A serialização é o ato de converter o objeto em texto. A desserialização é o ato de converter o texto de volta em um objeto.

No exemplo 6.5, o JavaScript está fazendo uma desserialização com JSON.parse(). Em responseText, o JSON é simplesmente um texto como formato para intercâmbio de dados. Após o parse ser feito por JSON.parse(), os dados não serão mais JSON, mas um objeto JavaScript.

EXEMPLO 6.5 – DESSERIALIZAÇÃO

```
var myJSON = JSON.parse(myXMLHttpRequest.responseText);
```

Essa desserialização com JSON.parse() é necessária porque o JSON ainda não é um objeto. Lembre-se de que JSON quer dizer JavaScript Object *Notation* (Notação de objetos JavaScript). Enquanto os dados estiverem no formato JSON, eles serão uma representação literal de um objeto na forma de texto. Para que o JSON se torne um objeto real, ele deverá ser desserializado. Em JavaScript, podemos também serializar o JSON com JSON.stringify().

No exemplo 6.6, a variável myObject recebe o JSON após a desserialização. Agora ele é um objeto. A variável myJSON recebe o objeto serializado. Agora

os dados são JSON.

Exemplo 6.6 – Objeto desserializado e em seguida serializado

```
// a resposta JSON desserializada
var myObject = JSON.parse(myXMLHttpRequest.responseText);

// o objeto serializado
var myJSON = JSON.stringify(myObject);
```

Por fim, as duas últimas linhas em meu exemplo criam a solicitação e a enviam por meio do protocolo HTTP (Exemplo 6.7).

Exemplo 6.7 – A solicitação de JSON é criada e enviada

```
myXMLHttpRequest.open("GET", url, true);
myXMLHttpRequest.send();
```

Talvez você ache estranho que o código que trata a resposta JSON esteja antes do envio da solicitação. Até falei dele antes de ter mencionado a criação e o envio da solicitação. O valor de função da propriedade onreadystatechange é um EventHandler. A engine JavaScript subjacente (não é um código meu) tem uma lógica para acessar o valor das propriedades (minha função) sempre que o "ready state" (estado da solicitação) mudar. Os estados da solicitação variam de 0 a 4 e correspondem a:

0 para UNSENT
: É o estado antes de a função open() ter sido executada.

1 para OPENED
: É o estado após a função open() ter sido executada, mas antes de send() ter sido chamada.

2 para HEADERS_RECEIVED
: É o estado após a função send() ter sido executada e os cabeçalhos e o status estarem disponíveis.

3 para LOADING
: Os cabeçalhos foram recebidos, mas o texto da resposta ainda está sendo obtido.

4 para DONE

Concluído; a mensagem completa, com os cabeçalhos e o corpo, foi recebida.

Lidar com os códigos de status para os estados da solicitação pode parecer confuso. No capítulo 7, veremos como XMLHttpRequest pode ser simplificado com a jQuery. Com a jQuery podemos obter os dados JSON usando a função getJSON() simples e lidar com o status de uma maneira mais legível aos seres humanos: com done(), fail() e always().

Em meu exemplo, utilizei um URL que pertence à API OpenWeatherMap. Esse é um web service que disponibiliza dados de clima por meio de uma API. A API tem URLs para recursos tanto XML quanto JSON para uma variedade de informações sobre o clima. Meu exemplo mostrou como solicitei e recebi dados JSON dessa API no lado cliente usando JavaScript. No entanto, os navegadores de Internet têm regras sobre compartilhamento que visam à segurança. A política de mesma origem (same-origin policy) define que um site poderá fazer esse tipo de solicitação em background somente para sites que estejam no mesmo domínio. Isso serve para proteger os usuários, pois, a menos que eles saibam o que são as ferramentas de desenvolvedor dos navegadores de Internet, eles não poderão *ver* essas solicitações ocorrendo.

O código que compartilhei em meu exemplo poderia ter sido executado em um script em um site cujo URL é *http://www.mycoolsite.com*. Esse é o domínio de origem, pois é o URL da solicitação original. O URL para a API é:

http://api.openweathermap.org/data/2.5/weather?lat=35&lon=139

O domínio *http://api.openweathermap.org* não corresponde ao domínio de origem, que é *http://www.mycoolsite.com*. Isso viola claramente a política de mesma origem. Essa solicitação e a resposta em background é possível somente por causa de alguns obstáculos superados pelos criadores da API OpenWeatherMap. Vamos dar uma olhada nesses obstáculos e nas brechas existentes para ver como as solicitações feitas nos bastidores para as APIs públicas são possíveis.

Problemas de relacionamento e regras sobre

compartilhamento

Sei que deixei você com algumas dúvidas lá atrás. Simplesmente mostrei um código que funciona, porém viola as regras de mesma origem dos navegadores de Internet. Esse código não deveria funcionar, porém funciona, somente por causa de uma brecha utilizada pelos criadores da API OpenWeatherMap. Parece uma novela em que duas pessoas não são capazes de entender o seu relacionamento. O navegador permitirá que a solicitação seja feita à API web pública? Fique ligado para saber.

Cross-Origin Resource Sharing (CORS)

Alguns desenvolvedores web podem passar anos escrevendo código JavaScript para AJAX que faça solicitações às APIs web públicas sem jamais se deparar com as políticas de mesma origem (same-origin policies). Isso ocorre porque a maioria das APIs públicas implementam o CORS em seus próprios servidores. O servidor disponibiliza algumas propriedades extras prefixadas com Access-Control-Allow no cabeçalho da resposta (Exemplo 6.8).

Exemplo 6.8 – Os cabeçalhos Access-Control-Allow da resposta da API OpenWeatherMap para o recurso em http://api.openweathermap.org/data/2.5/weather?lat=35&lon=139

```
Access-Control-Allow-Credentials:true
Access-Control-Allow-Methods:GET, POST
Access-Control-Allow-Origin:*
```

Esses cabeçalhos definem se as credenciais são permitidas, quais métodos HTTP são permitidos (GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE, CONNECT) e, acima de tudo, quais domínios de origem são permitidos. No exemplo, asterisco (*) foi fornecido. Esse é um caractere-curinga que informa que qualquer domínio de origem será permitido.

O CORS permite que sites como o OpenWeatherMap compartilhem seus dados de uma maneira que permita que os usuários os incluam em interações AJAX do lado cliente. Além do mais, o CORS pode ser usado para impedir que os sites executem tarefas prejudiciais com uma API, por exemplo, usar o CSRF. Você deve se lembrar do exemplo do capítulo 5 em que uma pessoa

mal-intencionada podia fazer um link com o JSON em uma tag <script> e explorar a confiança de um site bancário. Uma medida de segurança que o banco poderia ter usado é implementar o CORS e incluir os cabeçalhos mostrados no exemplo 6.9 em sua resposta.

EXEMPLO 6.9 – CORS USADO COMO MEDIDA DE SEGURANÇA

```
Access-Control-Allow-Methods:POST
Access-Control-Allow-Origin:http://www.somebank.com
```

No exemplo, POST é o único método permitido. Se alguém tentar solicitar esse recurso a partir de um URL em uma tag <script> (usando o método GET), o navegador não permitirá isso. Além do mais, ao especificar o URL do site de origem para o banco, o navegador de Internet impedirá o acesso de qualquer site que não seja *http://www.somebank.com*.

Implementar o CORS às vezes não será uma opção e o JSON nem sempre será hospedado por uma API web completa. Talvez você tenha dois sites com dois domínios diferentes (*http://domainone.com* e *http://domaintwo.com*) e queira compartilhar um arquivo JSON de *http://domainone.com* para *http://domaintwo.com*. É nesse caso que o JSON-P entra em cena.

JSON-P

JSON-P quer dizer *JSON with padding* (JSON com preenchimento). No capítulo 5, mencionei que as tags <script> são isentas das políticas de mesma origem. O JSON-P utiliza essa isenção para solicitar dados JSON dos servidores sem a mesma origem.

O JSON-P não é tão ideal quanto o CORS, pois é classificado como uma solução alternativa (o CORS é o método preferível, pois é um padrão). Entretanto, quando a situação exigir, uma solução alternativa como essa às vezes é necessária. É preciso ter um relacionamento entre *http://domainone.com* e *http://domaintwo.com*, e as regras do navegador sobre compartilhamento podem tornar isso frustrante quando o CORS não é uma opção.

O “padding” (preenchimento) em JSON-P é bem simples. É um JavaScript adicionado ao documento JSON. Veja o exemplo 6.10.

EXEMPLO 6.10 – JSON COM PADDING (JSON-P)

```
getTheAnimal(
   {
      "animal": "cat"
   }
);
```

O “padding” JavaScript nesse documento JSON corresponde a uma chamada de função, com o JSON fornecido como parâmetro. Os parâmetros de função proporcionam uma maneira de passar dados para a função. Por exemplo, se eu quiser ter uma função que some dois números, deverei ter uma maneira de passar esses dois números para a função.

A função a seguir é definida no código do lado cliente, em nosso JavaScript (Exemplo 6.11).

EXEMPLO 6.11 – A FUNÇÃO DECLARADA EM NOSSO JAVASCRIPT

```
function getTheAnimal(data) {
   var myAnimal = data.animal; // será "cat"
}
```

Depois que a função for declarada em JavaScript, algumas configurações serão necessárias. Essa é a parte do JSON-P que tira proveito da isenção da tag <script> em relação à política de mesma origem. Veja o exemplo 6.12.

EXEMPLO 6.12 – A TAG <SCRIPT> É CRIADA E CONCATENADA DINAMICAMENTE AO DOCUMENTO HTML APÓS A TAG <HEAD>

```
var script = document.createElement("script");
script.type = "text/javascript";
script.src = "http://notarealdomain.com/animal.json";
document.getElementsByTagName('head')[0].appendChild(script);
```

Para os servidores que implementam JSON-P, também é comum permitir que o usuário decida qual será o nome da função. Normalmente, ela é passada por meio do URL como um parâmetro de string de query. Veja o exemplo 6.13.

EXEMPLO 6.13 – UM NOME PARA A FUNÇÃO, INFORMADO AO SERVIDOR POR MEIO DA STRING DE QUERY

```
script.src = "http://notarealdomain.com/animal.json?callback=getThing";
```

O servidor receberá o nome de minha função na callback e incluirá o padding dinamicamente no arquivo JSON. Veja o exemplo 6.14.

EXEMPLO 6.14 – UMA FUNÇÃO NOMEADA DINAMICAMENTE NO JSON COM PADDING

```
getThing(
   {
      "animal": "cat"
   }
);
```

O JSON-P exige esforços do lado do servidor, pois o recurso JSON deve conter o padding JavaScript. Tanto o CORS quanto o JSON-P exigem um esforço do lado do servidor. Desse modo, um XMLHttpRequest entre domínios feito por um cliente dependerá do servidor do recurso JSON para que seja bem-sucedido.

Termos e conceitos essenciais

Este capítulo discutiu os seguintes termos essenciais:

API web
: É um conjunto de instruções e de padrões para interagir com um serviço por meio de HTTP.

XMLHttpRequest
: É um objeto JavaScript usado principalmente em programação AJAX que obtém dados de um URL sem que a página seja recarregada.

HyperText Transfer Protocol, ou Protocolo de transferência de hipertexto (HTTP)
: É o protocolo que constitui a base da comunicação de dados na World Wide Web.

Serialização
: É o ato de converter o objeto em texto.

Desserialização
: É o ato de converter o texto serializado de volta em um objeto.

Política de mesma origem (Same-origin policy)

O navegador de Internet exige que os scripts sejam provenientes do mesmo domínio por razões de segurança.

Cross-origin resource sharing (CORS)

Permite que recursos (por exemplo, um documento JSON) sejam solicitados de outro domínio que não seja o domínio de quem fez a solicitação, por meio da definição de permissões nos cabeçalhos da resposta.

JSON with padding, ou JSON com preenchimento (JSON-P)

Utiliza a isenção da tag <script> no que diz respeito à política de mesma origem para solicitar JSON de servidores sem a mesma origem.

Também discutimos os seguintes conceitos essenciais:

- O relacionamento entre o XMLHttpRequest do JavaScript e uma API web é um relacionamento cliente-servidor.
- XMLHttpRequest não serve apenas para XML. Ele é usado para solicitar recursos JSON também.
- Um site visa os seres humanos, enquanto uma API web visa o código. Ambos servem recursos por meio de HTTP.
- A política de mesma origem pode causar problemas de relacionamento nas interações cliente-servidor, entre o JavaScript e um recurso JSON.
- Um XMLHttpRequest entre domínios diferentes feito por um cliente dependerá do servidor do recurso JSON para que seja bem-sucedido.

CAPÍTULO 7

JSON e os frameworks do lado cliente

No capítulo 6, vimos um relacionamento cliente-servidor entre o navegador web e uma API web. Neste capítulo, daremos um zoom na lente para observar o lado cliente desse relacionamento e veremos como os frameworks do lado cliente aceitam ou tiram proveito do JSON.

Vamos ver rapidamente o que é um framework para aqueles que não tenham familiaridade com esse conceito. No mundo físico, um “framework” (estrutura ou armação) pode ser usado para descrever uma estrutura essencial de suporte, ou seja, uma estrutura subjacente. Em computação, um framework não é uma estrutura subjacente. É uma estrutura que está em uma camada acima do software ou de uma linguagem de programação para prover suporte aos desenvolvedores.

Um framework em computação é uma estrutura de suporte, porém não de uma maneira que possa ser relacionada às vigas de um edifício. Se a linguagem JavaScript fosse uma casa, um framework JavaScript não seria a estrutura de suporte dessa construção. Com efeito, poderíamos criar uma casa bem resistente sem um framework JavaScript.

Se estivéssemos no negócio de construção de casas com JavaScript, um framework JavaScript seria como pedir uma casa pré-fabricada, que já contivesse o encanamento e a fiação para a eletricidade. Com nossas casas pré-fabricadas, poderíamos nos focar na instalação de qualquer tipo de pia de cozinha que quiséssemos e prover a água seria simplesmente uma questão de conectar a pia ao encanamento. Poderíamos nos focar na instalação de belos armários e lindas bancadas de granito. Em sua essência, o nosso framework JavaScript nos permitiria economizar tempo e focar a criação das

funcionalidades.

Esse framework que possibilita economizar tempo e focar outros aspectos é conhecido em ciência da computação como uma ferramenta de abstração. Para as pessoas que não tenham familiaridade com o conceito de abstração, essa palavra poderá suscitar imagens da arte de Picasso, com rostos estranhos compostos de formas. A abstração não é uma arte complexa, mas uma técnica para lidar com sistemas complexos.

Se você recebesse a tarefa de construir uma espaçonave e não tivesse nenhuma experiência com ciência espacial, por onde você começaria? A maioria de nós diria “É evidente que não sou a pessoa mais adequada para essa tarefa”, mas e se você não tivesse outra opção? Você poderia encarar de frente essa tarefa complexa e achar que ela é impossível. Poderia entrar em pânico e trancar-se em um quarto. Ou você poderia usar a abstração.

Com a abstração, você pode focar uma parte do quebra-cabeça todo de cada vez. Dividimos a tarefa de construir uma nave espacial em todas as suas partes essenciais, por exemplo, prover suporte à vida humana na nave ou lançá-la para o espaço sideral. Focamos cada parte do sistema complexo até termos o sistema como um todo.

Uma ferramenta de abstração é qualquer ferramenta que facilite a abstração. Um framework JavaScript facilita a abstração por meio de bibliotecas previamente criadas que já lidem com as complexidades do sistema. Isso não quer dizer que possamos construir uma verdadeira nave espacial com um framework JavaScript, porém, se pudéssemos, esse framework poderia conter uma nave que já fosse capaz de ser lançada ao espaço. Isso permitiria que os construtores da nave focassem como eles proveriam suporte à vida humana na nave.

Há vários frameworks JavaScript (mais de cinquenta) disponíveis atualmente. Geralmente, eles são chamados de bibliotecas ou kits de ferramenta JavaScript. Cada um deles cria uma camada entre os sistemas complexos e a tarefa que você tem à mão. Muitos deles permitem uma fácil manipulação do DOM (Document Object Model, ou Modelo de objetos de documentos) do HTML, enquanto outros estão focados na criação de aplicações web completas do lado cliente.

Vamos dar uma olhada em alguns frameworks JavaScript e em seus relacionamentos com o JSON.

jQuery e JSON

A jQuery (*http://jquery.com/*) é uma ferramenta de abstração que permite aos desenvolvedores focarem-se na criação de funcionalidades ao cuidar da manipulação do DOM (Document Object Model, ou Modelo de objetos de documentos) do HTML. O DOM é uma convenção para interagir com a página HTML. Nesse modelo, o HTML subjacente é tratado como um objeto ou um conjunto de nós que pode ser enumerado, acessado e manipulado.

A manipulação do DOM em JavaScript é possível sem o framework jQuery. Entretanto, há algumas tarefas que os desenvolvedores devem executar rotineiramente com o DOM e que exigem diversas linhas de código. Por exemplo, uma tarefa comum a ser feita com o DOM é ocultar um elemento HTML. Vamos supor que tenho um botão que eu queira ocultar (Exemplo 7.1).

Exemplo 7.1 – Um botão HTML exibido em um navegador web com o texto "My Button"

```
<button id="myButton">My Button</button>
```

Conforme mostra o exemplo 7.2, para atingir o objetivo de ocultar o botão em JavaScript, devo chamar a função getElementById(*id*) no objeto do documento HTML e, em seguida, definir a propriedade style.display com "none".

Exemplo 7.2 – Esta linha de JavaScript oculta o botão HTML cujo id é "myButton"; esse botão não será exibido no navegador

```
document.getElementById("myButton").style.display = "none";
```

A jQuery alcança esse mesmo objetivo com menos da metade dos caracteres de código (Exemplo 7.3).

Exemplo 7.3 – Esta linha de jQuery oculta o botão HTML cujo id é "myButton"; esse botão não será exibido no navegador

```
$("#myButton").hide();
```

Ter de digitar menos da metade dos caracteres para atingir o mesmo objetivo permite economizar tempo de desenvolvimento. Além de economizar tempo, a jQuery também lida com problemas de compatibilidade entre os navegadores de Internet. Por exemplo, no capítulo 5, apresentei o JSON.parse() como uma alternativa mais segura a eval() para fazer parsing de JSON. Em versões mais antigas de Internet Explorer, Firefox e Chrome, o JSON.parse() não é tratado. Isso quer dizer que, qualquer que tenha sido a funcionalidade desenvolvida com o JSON.parse(), ela não poderá ser usada por um pequeno percentual de seus usuários que deixaram de atualizar os seus navegadores.

A jQuery trata muitos problemas comuns de compatibilidade de versão para você, incluindo o problema mencionado sobre o JSON.parse(). A jQuery tem uma função para parsing de JSON: jQuery.parseJSON. No exemplo 7.4, o parse do JSON é feito com o JSON.parse() do JavaScript. No exemplo 7.5, esse parse é feito com a função interna da jQuery.

Exemplo 7.4 – Parsing de JSON em JavaScript com JSON.parse()

```
var myAnimal = JSON.parse('{ "animal" : "cat" }');
```

Exemplo 7.5 – Parsing de JSON com a jQuery usando jQuery.parseJSON

```
var myAnimal = jQuery.parseJSON('{ "animal" : "cat" }');
```

Nos exemplos, podemos ver que o exemplo com a jQuery não parece estar proporcionando a economia de nenhum caractere na digitação. Entretanto, a função jQuery.parseJSON() está executando muito mais que uma linha de código. Se você observar o código dessa função no código-fonte da biblioteca jQuery, verá que ela tenta usar a função JSON.parse() nativa. Se ela não estiver disponível no navegador (nas versões mais antigas), new Function() será usada como alternativa, o que será equivalente a usar eval(). Além do mais, há uma lógica para verificar a existência de caracteres inválidos, que estariam presentes em um script em uma tentativa de ataque de injeção de código; nesse caso, um erro será gerado.

Além de jQuery.parseJSON, há uma função para fazer uma solicitação HTTP para obter dados JSON. Se dermos uma olhada em nosso exemplo do capítulo 6, veremos que solicitar um recurso com o JavaScript por meio de HTTP pode exigir uma porção grande de código (Exemplo 7.6).

Exemplo 7.6 – O código deste exemplo cria um novo objeto XMLHttpRequest e obtém dados JSON da API OpenWeatherMap

```
var myXMLHttpRequest = new XMLHttpRequest();
var url = "http://api.openweathermap.org/data/2.5/weather?lat=35&lon=139";

myXMLHttpRequest.onreadystatechange = function() {
   if (myXMLHttpRequest.readyState === 4
         && myXMLHttpRequest.status === 200) {
      var myObject = JSON.parse(myXMLHttpRequest.responseText);
      var myJSON = JSON.stringify(myObject);
   }
}
myXMLHttpRequest.open("GET", url, true);
myXMLHttpRequest.send();
```

Com a jQuery, podemos solicitar o recurso JSON ao escrever apenas algumas linhas de código (Exemplo 7.7).

Exemplo 7.7 – Este código jQuery cria um novo XMLHttpRequest e obtém o mesmo recurso JSON do exemplo anterior; a desserialização do JSON em um objeto JavaScript está incluída

```
var url = "http://api.openweathermap.org/data/2.5/weather?lat=35&lon=139";
$.getJSON(url, function(data) {
   // faz algo com os dados de clima
});
```

O framework jQuery para JavaScript trata o JSON de uma forma que permite economizar tempo de desenvolvimento na solicitação e no parsing do JSON. Isso inclui o parsing do JSON levando em consideração as versões mais antigas de navegadores de Internet. Podemos dizer que a jQuery *suporta* a interação com o JSON.

Agora vamos dar uma olhada no AngularJS – uma biblioteca que *tira proveito* de objetos JavaScript e do JSON

AngularJS

O framework jQuery é uma ferramenta de abstração que visa à manipulação do DOM (Document Object Model, ou Modelo de objetos de documentos). O

framework AngularJS (*https://angularjs.org/*) é uma ferramenta de abstração que visa às aplicações single-page (uma só página). As *aplicações web single-page* são páginas web que romperam com a abordagem multipage (múltiplas páginas) tradicional a fim de criar uma experiência mais natural para a aplicação.

A experiência tradicional com aplicações web multipage está intensamente associada à interação humana entre cliente e servidor. O ser humano digita ou clica em um URL para fazer a solicitação de um recurso usando HTTP. Essa dança entre o cliente, o servidor e o ser humano em uma aplicação web tradicional implica “mover-se” para uma nova página web a cada passo.

Antes de a Internet ter se transformado na fera capaz de lidar com volumes enormes de multimídia que é atualmente, a maioria das aplicações para as massas eram aplicações desktop. Se você quisesse ter uma enciclopédia digital, era necessário instalá-la em seu computador. A aplicação com a qual você interagia oferecia uma experiência razoavelmente natural. Ao pesquisar algo em sua enciclopédia, não havia nenhuma barra de URL nem uma página que mudava de endereço.

As aplicações web single-page procuram retornar a essa experiência natural com a aplicação, diretamente em seu navegador de Internet. Boa parte disso é conseguido por meio do JavaScript e do nosso bom e velho amigo XMLHttpRequest. Em vez de um ser humano pular de um recurso para outro por meio de URLs e de links para URLs, o código em background trata as solicitações de recursos, enquanto a pessoa permanece em uma página.

O framework AngularJS é uma ferramenta de abstração que evita que o desenvolvedor precise criar a aplicação single-page desde o princípio. A aplicação web single-page é um sistema complexo e, de forma alguma, o AngularJS cria essa aplicação para o desenvolvedor.

Em vez disso, ele disponibiliza um framework baseado no conceito MVC (Model-View-Controller, ou Modelo-Visão-Controlador) de arquitetura.

O AngularJS implementa o conceito de MVC da seguinte maneira:

Modelo (Model)

Os objetos JavaScript representam o modelo de dados.

Visão (View)

É o HTML [uma sintaxe para binding (associação) de dados com o modelo é usada].

Controlador (Controller)

Os arquivos JavaScript que usam a sintaxe do AngularJS para definir e tratar as interações com o modelo e a visão.

O AngularJS pode ter uma curva de aprendizado longa, especialmente para aqueles que trabalham com JavaScript e com manipulações de DOM (Document Object Model, ou Modelo de objetos de documentos) do HTML há anos. Ele exige uma mudança de raciocínio na forma de interagir com o DOM. O AngularJS procura desacoplar a manipulação do DOM da lógica da aplicação.

Esse desacoplamento da manipulação do DOM pode ser expresso ao compararmos o modo como atingiríamos o mesmo objetivo com e sem o framework AngularJS. Por exemplo, preciso alterar uma mensagem em uma página, do “Hello, stranger” genérico (Exemplo 7.8) para uma saudação personalizada quando um usuário fizer login (Exemplo 7.9). Em JavaScript, fazemos isso por meio da manipulação do DOM.

EXEMPLO 7.8 – A MENSAGEM HTML QUANDO O USUÁRIO NÃO ESTIVER LOGADO

```
<h1 id="message">Hello, stranger!</h1>
```

EXEMPLO 7.9 – O JAVASCRIPT PARA ALTERAR A MENSAGEM SE O USUÁRIO ESTIVER LOGADO

```
if (signedIn) {
   var message = "Hello, " + userName + "!";
   document.getElementById("message").innerHTML = message;
}
```

Em AngularJS, como o HTML é a nossa visão (view), ele será configurado para que esteja preparado para mudanças em nosso modelo de dados (Exemplo 7.10).

EXEMPLO 7.10 – A VISÃO HTML COM OS ATRIBUTOS DO ANGULARJS E A SINTAXE PARA BINDING DE DADOS

```
<body ng-app="myApp">
   <div id="wrapper" ng-controller="myAppController">
      <h1 id="message">Hello {{ userData.userName }}!</h1>
   </div>
</body>
```

Os atributos "ng-app" e "ng-controller" nas tags HTML (<body> e) fazem parte da sintaxe para configurar a visão para que ela trate o binding de dados. O binding de dados é exatamente o que a palavra quer dizer: fazemos o "binding" (associação) dos dados a uma página por meio de uma série de placeholders. A sintaxe do AngularJS utiliza "handlebars" para esses placeholders nos pontos em que os dados devem ser associados. Esses handlebars correspondem a duas chaves de abertura ({{) e duas chaves de fechamento (}}) em torno da sintaxe para o placeholder (userData.userName no exemplo 7.10).

No exemplo 7.11, o controlador JavaScript com um objeto userData é adicionado ao escopo global; isso permite que a visão (o HTML) utilize o objeto para o binding de dados.

EXEMPLO 7.11 – OBJETO USERDATA ADICIONADO AO ESCOPO GLOBAL

```
angular.module('myApp', [])
   .controller('myAppController', function($scope) {
   $scope.userData = { userName : "Stranger" };
   if(signedIn)
   {
      $scope.userData = { userName : "Bob" };
   }
});
```

No arquivo de controlador JavaScript do AngularJS, em vez de manipular o DOM, o modelo de dados é manipulado. O modelo de dados sendo manipulado é o objeto userData. Tanto o exemplo com JavaScript/HTML puros quanto o exemplo com o AngularJS resultam na mesma funcionalidade. Em meu exemplo simples, a opção com o AngularJS parece exigir mais esforços, especialmente para uma ferramenta de abstração. No entanto, ao mergulhar em uma aplicação complexa, por exemplo, uma página que permita que os usuários executem as operações CRUD (Create, Read,

Update and Delete, ou Criar, ler, atualizar e apagar) em suas informações de usuário, a alternativa com o AngularJS simplifica o desenvolvimento.

O exemplo 7.11 mostrou um objeto JavaScript userData como um modelo de dados. O AngularJS tira proveito dos objetos JavaScript ao usá-los como modelo na arquitetura MVC. Como sabemos, o JSON utiliza a notação de objetos literais do JavaScript. Sendo assim, como o JSON se enquadra no AngularJS?

A pergunta a ser feita é: “De onde vem o nosso modelo de dados?” No código de exemplo, meu modelo de dados estava fixo no código do controlador. Se retornarmos ao exemplo da enciclopédia digital, sabemos que os dados devem estar em algum lugar. Quando os dados são armazenados, normalmente isso é feito em algum tipo de banco de dados.

Cabe aos desenvolvedores ler esses dados no modelo de dados de nossa aplicação web single-page. Em um modelo de dados do AngularJS, o veículo mais comum para obter os dados do banco de dados e lê-los em um modelo é o JSON. Geralmente, esses dados serão um recurso JSON solicitado por meio de HTTP, ou seja, um relacionamento cliente-servidor. O AngularJS tira proveito desse relacionamento para obter facilmente os modelos de dados usando um serviço básico do AngularJS: o $http.

No exemplo 7.12, o $http do AngularJS é usado para obter os dados de clima da API OpenWeatherMap; o JSON é acrescentado ao escopo global na forma de um objeto chamado weatherData. A resposta está sendo apresentada no exemplo 7.13.

EXEMPLO 7.12 – OBTENDO DADOS DE CLIMA DA API WEB OPENWEATHERMAP

```
angular.module('myApp', [])
   .controller('myAppController', function($scope, $http) {
      $http.get('http://api.openweathermap.org/data/2.5/weather?lat=35&lon=139').
         success(function(data, status, headers, config) {
            $scope.weatherData = data;
         });
});
```

EXEMPLO 7.13 – A RESPOSTA JSON DA API WEB OPENWEATHERMAP

```
{
   "coord": {
      "lon": 139,
      "lat": 35
   },
   "sys": {
      "message": 0.0099,
      "country": "JP",
      "sunrise": 1431805135,
      "sunset": 1431855710
   },
   "weather": [
      {
         "id": 800,
         "main": "Clear",
         "description": "sky is clear",
         "icon": "02n"
      }
   ],
   "base": "stations",
   "main": {
      "temp": 291.116,
      "temp_min": 291.116,
      "temp_max": 291.116,
      "pressure": 1020.61,
      "sea_level": 1028.58,
      "grnd_level": 1020.61,
      "humidity": 95
   },
   "wind": {
      "speed": 1.51,
      "deg": 339.501
   },
   "clouds": {
      "all": 8
   },
   "dt": 1431874405,
```

```
  "id": 1851632,
  "name": "Shuzenji",
  "cod": 200
}
```

O JSON da API OpenWeatherMap é desserializado pelo `$http` do AngularJS. Após ter sido adicionado ao escopo global como um objeto chamado `weatherData`, ele será o nosso modelo de dados, com o qual será possível fazer o binding com a nossa visão HTML usando a sintaxe de handlebars.

No exemplo 7.14, na sintaxe com os handlebars, os dados aos quais será feito o binding correspondem à descrição do clima. A descrição do clima está na propriedade `weather`, que contém um array de objetos com apenas um valor. Selecionei o valor no índice 0 com `weather[0]` e, em seguida, usei a propriedade `description`.

EXEMPLO 7.14 – BINDING DA DESCRIÇÃO DO CLIMA

```
<body ng-app="myApp">
  <div id="wrapper" ng-controller="myAppController">
    <div>
      {{ weatherData.weather[0].description }}
    </div>
  </div>
</body>
```

Na época em que escrevi este capítulo, os dados do clima informavam “sky is clear” (céu limpo). Eu poderia fazer essa aplicação single-page ser atualizada a cada 60 segundos para fornecer a descrição do clima nesse momento. A solicitação HTTP ocorreria nos bastidores e a atualização do modelo de dados alteraria automaticamente a visão HTML. O ser humano não precisaria iniciar essa solicitação; tudo isso pode ocorrer em segundo plano. Além disso, com o conceito de MVC, o código JavaScript que eu escrever não precisará fazer alterações no DOM. Atualizar o modelo de dados fará a visão ser automaticamente atualizada.

O AngularJS tira proveito dos objetos JavaScript e do JSON em sua arquitetura Modelo-Visão-Controlador. A jQuery trata o JSON com funções simples para solicitar e fazer parsing de JSON. Boa parte do sucesso de um

formato para intercâmbio de dados depende do tratamento disponibilizado por aqueles que fazem a troca de dados. No relacionamento cliente-servidor na Web, podemos ver que o JSON é aceito tanto pelo JavaScript quanto pelas suas ferramentas eficazes de abstração. Nos capítulos 8 e 9, veremos como o JSON é aceito e utilizado no lado servidor desse relacionamento.

Termos e conceitos essenciais

Este capítulo discutiu os seguintes termos essenciais:

Abstração
: É uma técnica para lidar com sistemas complexos que envolve focar as áreas menores desse sistema.

Framework
: É uma ferramenta de abstração que permite economizar tempo e focar o desenvolvimento das funcionalidades.

jQuery.parseJSON()
: É uma função que não só utiliza a função JSON.parse(), como também provê uma função alternativa para navegadores mais antigos que não tratam o JSON.parse(); além disso, essa função trata a brecha de segurança relacionada à avaliação de uma string para a validação de seus caracteres.

jQuery.getJSON()
: É um atalho para a função jquery.ajax() e faz parse do JSON, gerando um objeto JavaScript, tudo de uma só vez.

Aplicações web single-page
: Páginas web que romperam com a abordagem multipage (múltiplas páginas) tradicional a fim de criar uma experiência mais natural para a aplicação.

MVC (Model-View-Controller, ou Modelo-Visão-Controlador)
: Um padrão de arquitetura que divide uma aplicação em três componentes: o modelo (dados), a visão (apresentação) e o controlador (atualiza o modelo e a visão).

AngularJS

Um framework JavaScript MVC que utiliza objetos JavaScript como modelo de dados.

Também discutimos os seguintes conceitos essenciais:

- A jQuery é uma ferramenta de abstração que oferece *suporte* ao JSON ao reduzir o tempo de desenvolvimento para a solicitação e o parsing de dados. Além disso, ela trata problemas relacionados a versões diferentes de navegadores.
- No conceito de MVC do AngularJS:
  - o JSON é o modelo (ou o modelo de dados);
  - o HTML é a visão e utiliza a sintaxe de binding (associação) de dados com o modelo;
  - os controladores são os arquivos JavaScript do AngularJS e contêm a sintaxe para definição e tratamento das interações com o modelo e a visão.
- O AngularJS *tira proveito* dos objetos JavaScript e do JSON em sua arquitetura MVC.

CAPÍTULO 8

JSON e NoSQL

No capítulo 7, demos um zoom na lente para observar o lado cliente do relacionamento cliente-servidor na Web. Neste capítulo, daremos um zoom nessas lentes no lado servidor para observar um tipo de banco de dados que não só usa documentos JSON para armazenar dados, como também faz a interface com o mundo externo por meio de uma API web.

Vivemos na "era da informação", em que quantidades enormes de dados e de conhecimento são facilmente acessíveis. Podemos pesquisar informações sobre um assunto específico como "rato-toupeira pelado" em sites como a Wikipedia. As informações sobre o "rato-toupeira pelado" devem estar armazenadas em algum lugar, e esse lugar, em última instância, é um banco de dados.

Se você já trabalhou com um banco de dados antes e tem familiaridade com o SQL, é mais provável que você tenha trabalhado com um *banco de dados relacional*. Os bancos de dados relacionais são estruturados por meio de tabelas, colunas e linhas. Cada tabela representa uma informação, por exemplo, uma conta. A tabela que representa uma conta pode ter um relacionamento com uma tabela que representa os endereços associados a essas contas. Uma chave em cada tabela, por exemplo, uma coluna que contenha um identificador da conta, forma um relacionamento entre as duas tabelas.

Para criar, manipular e consultar esses bancos de dados relacionais, utilizamos o *SQL* (Structured Query Language, ou Linguagem de consulta estruturada). Com o SQL, podemos fazer query em colunas e em linhas de uma ou mais tabelas do banco de dados (Exemplo 8.1).

EXEMPLO 8.1 – UMA QUERY SQL SIMPLES QUE RETORNA COLUNAS E LINHAS PARA

accountID, firstName e lastName da tabela Account

```
SELECT accountId, firstName, lastName
FROM Account
```

Com os relacionamentos entre as tabelas em um banco de dados relacional, podemos também fazer query de colunas e de linhas de várias tabelas (Exemplo 8.2).

Exemplo 8.2 – Uma query que reúne (faz o join de) duas tabelas; ela retornará as colunas e as linhas para o primeiro nome, o sobrenome, o endereço residencial e o CEP de todas as contas e seus endereços relacionados

```
SELECT Account.firstName, Account.lastName, Address.street, Address.zip
FROM Account
JOIN Address
ON Account.accountId = Address.accountId
```

O título “NoSQL” dado a um banco de dados nos informa que o banco de dados não é relacional. Não podemos usar o SQL para pedir colunas e linhas de tabelas reunidas do banco de dados. O que esse título não informa é *o que ele é*. Ele simplesmente diz que “encontrou uma maneira alternativa de armazenar dados que não é a forma relacional”. Naturalmente, há mais de uma “maneira alternativa”.

Os bancos de dados NoSQL no mundo atualmente variam em natureza tanto quanto um tigre difere de um elefante. Esses tipos de bancos de dados variam quanto à natureza porque os dados têm diferentes tamanhos, formatos e propósitos. Os criadores desses bancos de dados NoSQL perceberam isso e, desse modo, criaram lares para os dados serem armazenados e lidos de modo a romperem com o modelo relacional tradicional.

Um exemplo de um banco de dados NoSQL é o *key-value store* (armazenamento chave-valor). O key-value store modela dados simples na forma de pares chave-valor. Se tomássemos o dicionário de inglês e criássemos um banco de dados a partir dele, esses dados se enquadrariam bem em um key-value store. Cada palavra poderia ser uma chave e cada definição, um valor. Esse tipo de banco de dados é uma ótima alternativa para o overhead causado em um banco de dados relacional por estruturas simples de dados.

O tipo do banco de dados NoSQL que vamos explorar neste capítulo é um *document store* (orientado a documentos). Essa alternativa a um banco de dados relacional estrutura os dados em torno do conceito de documento em vez de representar e relacionar porções de dados na forma de tabelas. Atualmente, há mais de vinte bancos de dados orientados a documentos sendo usados no mundo; alguns utilizam documentos XML, enquanto outros utilizam documentos JSON.

Neste capítulo, daremos uma olhada em um tipo de banco de dados orientado a documentos que utiliza documentos JSON para armazenar dados. Vamos dar uma olhada no CouchDB.

O banco de dados CouchDB

Pelo fato de ser um banco de dados NoSQL, o CouchDB (*http://couchdb.apache.org/*) tem uma maneira não relacional de armazenar dados com documentos JSON.

Vamos supor que precisamos armazenar dados de contas de clientes. Em um banco de dados relacional, as informações de contas poderiam espalhar-se rapidamente por diversas tabelas. Se houver necessidade de termos mais de um endereço para uma conta, uma tabela separada de endereços deverá ser usada para tratar esse relacionamento de um para muitos (one-to-many).

Em um banco de dados relacional, em um relacionamento de um para muitos – por exemplo, uma conta com vários endereços –, a query deve ser feita com um join para que os dados sejam reunidos novamente. Além disso, se fizermos uma query no banco de dados para consultar uma única conta e todos os registros de endereço reunidos, o resultado conterá diversas linhas. Cada linha terá campos repetidos para a mesma conta, com um endereço diferente em cada linha. Desse modo, a estrutura de dados está no banco de dados, porém será transformada em colunas e linhas quando uma query for feita.

Com o CouchDB, os relacionamentos entre os dados não exigem que esses dados estejam separados para armazenamento e sejam reunidos posteriormente para leitura. Os relacionamentos são expressos nos dados e a estrutura é mantida à medida que os dados forem incluídos ou excluídos do

banco de dados.

No exemplo 8.3, o relacionamento de um para muitos dos endereços das contas é representado na forma de objetos em um array.

EXEMPLO 8.3 – UM DOCUMENTO JSON QUE REPRESENTA UMA CONTA

```
{
  "firstName": "Bob",
  "lastName": "Barker",
  "age": 91,
  "addresses": [
    {
      "street": "123 fake st",
      "city": "Somewhere",
      "state": "OR",
      "zip": 97520
    },
    {
      "street": "456 fakey ln",
      "city": "Some Place",
      "state": "CA",
      "zip": 96026
    }
  ]
}
```

Com o banco de dados CouchDB orientado a documentos, quando pedirmos uma conta ao banco de dados, teremos um documento estruturado como resposta. Nenhuma composição de dados será necessária. Isso proporciona conveniência e rapidez.

No entanto, um banco de dados orientado a documentos não constitui uma solução única para qualquer problema de armazenamento de dados. Esse modelo pode rapidamente se tornar problemático caso várias associações sejam necessárias. O que ocorreria se quiséssemos ter relacionamentos entre cidades, estados e CEPs para um endereço? Assim que esses relacionamentos se tornarem necessários, o modelo relacional será mais adequado para os dados, pois será difícil expressar relacionamentos complexos em um único

documento.

Outra tarefa que o CouchDB executa muito bem é facilitar a evolução dos dados. Alguns dados evoluem com o tempo. Por exemplo, em 1990, é provável que os campos a seguir para números de telefone fossem adequados em um registro para uma conta:

- Telefone residencial
- Telefone comercial
- Fax

Agora que não estamos mais em 1990, precisamos de um campo para um telefone celular. Os dados devem evoluir.

Em um banco de dados relacional, seria necessário modificar o esquema da tabela de endereços para aceitar esse campo de telefone celular. Também poderíamos decidir que nos livraríamos dos campos da tabela de contas e transferiríamos os números de telefone para a sua própria tabela.

Com o CouchDB, nossos dados podem evoluir sem a necessidade de modificar os esquemas. Posso facilmente representar os números de telefone em JSON como um array de objetos (Exemplo 8.4) e permitir que uma conta tenha tantos registros de números de telefone quantos realmente existirem.

EXEMPLO 8.4 – UM ARRAY DE OBJETOS PARA NÚMEROS DE TELEFONE

```
{
  "phoneNumbers": [
    {
      "description": "home phone",
      "number": "555-555-5555"
    },
    {
      "description": "cell phone",
      "number": "123-456-7890"
    },
    {
      "description": "fax",
      "number": "456-789-1011"
    }
```

```
  ]
}
```

No futuro, se uma conta exigir um novo número, poderemos simplesmente adicioná-lo ao array, conforme mostrado no exemplo 8.5.

EXEMPLO 8.5 – O ARRAY DE OBJETOS PARA NÚMEROS DE TELEFONE ADQUIRE UM NOVO "SPACE PHONE" (TELEFONE ESPACIAL)

```
{
  "phoneNumbers": [
    {
      "description": "home phone",
      "number": "555-555-5555"
    },
    {
      "description": "cell phone",
      "number": "123-456-7890"
    },
    {
      "description": "fax",
      "number": "456-789-1011"
    },
    {
      "description": "space phone",
      "number": "932-932-932"
    }
  ]
}
```

Além disso, se precisarmos de um novo campo no registro da conta, por exemplo, "galaxy" (galáxia), simplesmente o adicionaremos a todos os futuros registros (Exemplo 8.6). Nenhuma modificação de esquema será necessária.

EXEMPLO 8.6 – A ADIÇÃO DE UM CAMPO "GALAXY" EM UM REGISTRO DE CONTA

```
{
  "firstName": "Octavia",
  "lastName": "Wilson",
```

```
  "age": 26,
  "addresses": [
    {
      "street": "123 fake st",
      "city": "Somewhere",
      "state": "OR",
      "zip": 97520
    }
  ],
  "galaxy": "Milky Way"
}
```

Os dados são inseridos e excluídos do banco de dados CouchDB como documentos JSON. A pergunta agora é: como inserimos os dados e como nos livramos deles? O CouchDB utiliza o HTTP para uma API. Vamos dar uma olhada na API do CouchDB.

API do CouchDB

Com o CouchDB, a maneira de pedir dados ao banco de dados ocorre por meio de uma solicitação de recursos com o HTTP. Com o HTTP, fazemos solicitações de recursos usando um URL. O recurso solicitado à API do CouchDB é um documento JSON (application/json).

Se eu tiver um banco de dados chamado “accounts” em meu CouchDB instalado localmente, poderei usar o URL *http://localhost:5984/accounts/* para obter informações sobre o banco de dados. No exemplo 8.7, as informações sobre o banco de dados de contas serão retornadas em formato JSON.

EXEMPLO 8.7 – RESPOSTA DE HTTP://LOCALHOST:5984/ACCOUNTS/

```
{
  "db_name": "accounts",
  "doc_count": 3,
  "doc_del_count": 0,
  "update_seq": 7,
  "purge_seq": 0,
  "compact_running": false,
```

```
    "disk_size": 28773,
    "data_size": 1248,
    "instance_start_time": "1432493477586600",
    "disk_format_version": 6,
    "committed_update_seq": 7
}
```

Observe o par nome-valor que contém "doc_count". Ele especifica quantos documentos existem no meu banco de dados e, nesse momento, há três. Posso fazer a query de cada documento do banco de dados por meio do identificador único de cada documento usando o URL *http://localhost:5984/accounts/<identificador_único>*. Se eu não souber quais são os identificadores únicos de meus documentos, poderei usar o URL *http://localhost:5984/accounts/_all_docs* para obter um array com os identificadores das linhas.

No exemplo 8.8, o recurso JSON no URL *http://localhost:5984/accounts/_all_docs* contém um array com os identificadores de documentos para cada documento de meu banco de dados.

EXEMPLO 8.8 – ARRAY DE IDENTIFICADORES DE DOCUMENTO

```
{
    "total_rows": 3,
    "offset": 0,
    "rows": [
        {
            "id": "3636fa3c716f9dd4f7407bd6f7000552",
            "key": "3636fa3c716f9dd4f7407bd6f7000552",
            "value": {
                "rev": "1-8a9527cbfc22e28984dfd3a3e6032635"
            }
        },
        {
            "id": "ddc14efcf71396463f53c0f880001538",
            "key": "ddc14efcf71396463f53c0f880001538",
            "value": {
                "rev": "1-3aef6f6ae7fff90dac3ff5d6c4460ceb"
            }
```

```
    },
    {
      "id": "ddc14efcf71396463f53c0f8800019ea",
      "key": "ddc14efcf71396463f53c0f8800019ea",
      "value": {
        "rev": "5-c38761b818edaf9842a63574927b7d38"
      }
    }
  ]
}
```

Se eu usar o primeiro identificador (3636fa3c716f9dd4f7407bd6f7000552) desse array, poderei estruturar um URL para fazer uma solicitação desse documento (Exemplo 8.9).

EXEMPLO 8.9 – RECURSO JSON QUE REPRESENTA UMA CONTA NO URL HTTP://LOCALHOST:5984/ACCOUNTS/3636FA3C716F9DD4F7407BD6F7000552

```
{
  "_id": "3636fa3c716f9dd4f7407bd6f7000552",
  "_rev": "1-8a9527cbfc22e28984dfd3a3e6032635",
  "firstName": "Billy",
  "lastName": "Bob",
  "address": {
    "street": "123 another st",
    "city": "Somewhere",
    "state": "OR",
    "zip": "97501"
  },
  "age": 54,
  "gender": "male",
  "famous": false
}
```

Vimos como solicitar dados de um banco de dados CouchDB. Agora vamos dar uma olhada em como fornecer dados a ele.

Se eu quiser adicionar outro documento de conta em meu banco de dados de contas, farei isso com um post para o URL *http://localhost:5984/accounts/* (veja os exemplos 8.10 e 8.11).

EXEMPLO 8.10 – CABEÇALHOS HTTP PARA UMA SOLICITAÇÃO (HTTP://LOCALHOST:5984/ACCOUNTS/) USANDO O MÉTODO POST

```
POST /accounts/ HTTP/1.1
Host: localhost:5984
Content-Type: application/json
Cache-Control: no-cache
```

EXEMPLO 8.11 – CORPO DO HTTP PARA UMA SOLICITAÇÃO (HTTP://LOCALHOST:5984/ACCOUNTS/) USANDO O MÉTODO POST

```
{
   "firstName": "Janet",
   "lastName": "Jackson",
   "address": {
      "street": "456 Fakey Fake st",
      "city": "Somewhere",
      "state": "CA",
      "zip": "96520"
   },
   "age": 54,
   "gender": "female",
   "famous": true
}
```

Se você usar o Chrome como o seu navegador de Internet, dê uma olhada no “Postman” no Chrome Web Store. O Postman oferece uma interface simples para criar e testar solicitações HTTP para APIs com todos os métodos HTTP (GET, POST, PUT, DELETE etc.). Além disso, o Postman salva um histórico de suas solicitações, de modo que você poderá testar as mesmas solicitações repetidamente sem a necessidade de recriá-las.

Após a solicitação HTTP ter sido bem-sucedida, a API do CouchDB retornará uma mensagem de resposta em formato JSON que incluirá o identificador recém-gerado para o documento (Exemplo 8.12).

EXEMPLO 8.12 – A RESPOSTA JSON DE HTTP://LOCALHOST:5984/ACCOUNTS/

```
{
   "ok": true,
   "id": "3636fa3c716f9dd4f7407bd6f700076c",
   "rev": "1-363f3b4bf90183781d08fe22487f3c90"
```

```
}
```

Agora poderei compor um URL usando o novo identificador único para solicitar o documento JSON do banco de dados de contas (Exemplo 8.13).

Exemplo 8.13 – Recurso no URL http://localhost:5984/accounts/3636fa3c716f9dd4f7407bd6f700076c

```
{
   "_id": "3636fa3c716f9dd4f7407bd6f700076c",
   "_rev": "1-363f3b4bf90183781d08fe22487f3c90",
   "firstName": "Janet",
   "lastName": "Jackson",
   "address": {
      "street": "456 Fakey Fake st",
      "city": "Somewhere",
      "state": "CA",
      "zip": "96520"
   },
   "age": 54,
   "gender": "female",
   "famous": true
}
```

Se eu quiser atualizar o meu novo documento JSON, poderei simplesmente incluir os pares nome-valor `"_id"` e `"_rev"` no corpo de minha solicitação e fazer um POST para o URL do recurso. Por exemplo, se eu quiser atualizar a idade de Janet Jackson para 55, enviarei o mesmo documento do exemplo anterior com a idade atualizada. A API então responderá com um documento com o status JSON que incluirá o par nome-valor `"rev"` atualizado (veja o exemplo 8.14).

Exemplo 8.14 – Resposta após fazer o POST do documento JSON atualizado para http://localhost:5984/accounts/3636fa3c716f9dd4f7407bd6f700076c

```
{
   "ok": true,
   "id": "3636fa3c716f9dd4f7407bd6f700076c",
   "rev": "3-29ede949ceed9df62bd7caecb095bffe"
}
```

Se eu quiser apagar um documento, usarei o método HTTP DELETE e passarei o identificador de revisão no URL como um parâmetro da string de query (Exemplo 8.15).

EXEMPLO 8.15 – "REV" É INFORMADO COMO UM PARÂMETRO DA STRING DE QUERY

```
http://localhost:5984/accounts/3636fa3c716f9dd4f7407bd6f700076c?
    rev=3-29ede949ceed9df62bd7caecb095bffe
```

Se eu solicitar o recurso no URL *http://localhost:5984/accounts/* mais uma vez, verei agora que há quatro documentos (Exemplo 8.16).

EXEMPLO 8.16 – DOCUMENTO JSON NO URL HTTP://LOCALHOST:5984/ACCOUNTS/; O PAR NOME-VALOR COM "DOC_COUNT" AGORA TEM UM VALOR IGUAL A 4

```
{
   "db_name": "accounts",
   "doc_count": 4,
   "doc_del_count": 0,
   "update_seq": 8,
   "purge_seq": 0,
   "compact_running": false,
   "disk_size": 32869,
   "data_size": 1560,
   "instance_start_time": "1432493477586600",
   "disk_format_version": 6,
   "committed_update_seq": 8
}
```

Esses exemplos mostraram como podemos visualizar e criar documentos JSON armazenados em um banco de dados CouchDB. Se isso fosse tudo o que pudéssemos fazer, iríamos nos deparar rapidamente com problemas. Talvez eu precise de uma lista dos sobrenomes que inclua somente as pessoas famosas. No CouchDB, podemos fazer isso com "views" (visualizações). Veja o exemplo 8.17.

As views são o modo como podemos reestruturar e fazer query de dados de um banco de dados CouchDB. Elas são armazenadas em um documento JSON chamado de documento de design. O documento de design especifica a linguagem e pode incluir várias views.

EXEMPLO 8.17 – UM DOCUMENTO DE DESIGN DO COUCHDB PARA O BANCO DE DADOS DE CONTAS QUE INCLUI UMA VIEW PARA PESSOAS FAMOSAS

```
{
   "_id": "_design/lists",
   "_rev": "8-4124de7756277c6a937004a763d6247d",
   "language": "javascript",
   "views": {
      "famous": {
         "map": "function(doc){if(doc.lastName!==null&&doc.famous){emit
         (doc.lastName,null)}}"
      }
   }
}
```

Cada view é um objeto que pode conter uma função map e uma função reduce. No exemplo 8.17, há apenas uma função map. A função map aceita cada um dos documentos como parâmetro e, em seguida, a função emit() é chamada. Isso essencialmente transforma os dados (Exemplo 8.18).

EXEMPLO 8.18 – UMA VISÃO MAIS DETALHADA DA FUNÇÃO "MAP" PARA A VIEW "FAMOUS"

```
function(doc) {
   if (doc.lastName !== null && doc.famous) {
      emit(doc.lastName, null)
   }
}
```

No exemplo 8.18, a função verifica cada documento JSON do banco de dados para garantir que temos um sobrenome e que o par nome-valor com "famous" seja verdadeiro. Os parâmetros de emit() são key e value, e eles serão exibidos como pares nome-valor no documento transformado resultante.

Poderei então solicitar essa view como um recurso por meio do URL *http://localhost:5984/accounts/_design/lists/_view/famous* (veja o exemplo 8.19). A estrutura é */<nome_do_banco_de_dados>/_design/<nome_do_documento_de_design>/_*

EXEMPLO 8.19 – O RECURSO NO URL

HTTP://LOCALHOST:5984/ACCOUNTS/_DESIGN/LISTS/_VIEW/FAMOUS


```
{
   "total_rows": 2,
   "offset": 0,
   "rows": [
      {
         "id": "ddc14efcf71396463f53c0f880001538",
         "key": "Barker",
         "value": null
      },
      {
         "id": "3636fa3c716f9dd4f7407bd6f700076c",
         "key": "Jackson",
         "value": null
      }
   ]
}
```


Além do passo relacionado a map, você poderá, opcionalmente, fazer uma redução (reduce) do resultado. O CouchDB tem três funções de redução incluídas: _count, _sum e _stats. Elas são úteis após a transformação dos dados com o passo map a fim de obter informações estatísticas sobre o seu conjunto de dados.

Por exemplo, posso querer informações sobre quantas contas há, de acordo com o sexo (gender). Posso acrescentar uma nova view ao meu documento de design (veja o exemplo 8.20).

EXEMPLO 8.20 – O DOCUMENTO DE DESIGN COM O ACRÉSCIMO DE UMA VIEW "GENDER_COUNT"

```
{
   "famous": {
      "map": "function(doc){if(doc.lastName!==null&&doc.famous){emit
      (doc.lastName,null)}}"
   },
   "gender_count": {
      "map": "function(doc){if(doc.gender!==null) emit(doc.gender);}",
      "reduce": "_count"
```

```
    }
}
```

No passo relacionado a map para a view "gender_count", a função está "emitindo" o valor do par nome-valor com "gender". Sem o passo relacionado a reduce, isso geraria um array de objetos que tenha um valor de sexo (gender) para "key" (veja o exemplo 8.21).

EXEMPLO 8.21 – A VIEW "GENDER_COUNT" APÓS O PASSO RELACIONADO A MAP


```
{
   "total_rows": 4,
   "offset": 0,
   "rows": [
      {
         "id": "3636fa3c716f9dd4f7407bd6f700076c",
         "key": "female",
         "value": null
      },
      {
         "id": "ddc14efcf71396463f53c0f8800019ea",
         "key": "female",
         "value": null
      },
      {
         "id": "3636fa3c716f9dd4f7407bd6f7000552",
         "key": "male",
         "value": null
      },
      {
         "id": "ddc14efcf71396463f53c0f880001538",
         "key": "male",
         "value": null
      }
   ]
}
```


Após o passo referente a reduce com a função interna "_count", teremos um resultado que fornece um contador de todos os registros (veja o exemplo 8.22).

EXEMPLO 8.22 – RECURSO NO URL HTTP://LOCALHOST:5984/ACCOUNTS/_DESIGN/LISTS/_VIEW/GENDER_COUNT

```
{
   "rows": [
      {
         "key": null,
         "value": 4
      }
   ]
}
```

O que estamos procurando é um contador para cada sexo único, portanto devemos passar a flag *group* para que o resultado seja agrupado de acordo com a chave. Para isso, acrescentamos um parâmetro de string de query *?group=true* ao URL (veja o exemplo 8.23).

EXEMPLO 8.23 – RECURSO NO URL HTTP://LOCALHOST:5984/ACCOUNTS/_DESIGN/LISTS/_VIEW/GENDER_COUNT?GROUP=TRUE

```
{
   "rows": [
      {
         "key": "female",
         "value": 2
      },
      {
         "key": "male",
         "value": 2
      }
   ]
}
```

Com a API do CouchDB podemos criar qualquer quantidade de documentos de design e views para um banco de dados. Cada uma das views nos documentos de design podem conter opcionalmente uma função map e uma função reduce para transformar os dados. Podemos então criar o nosso conjunto personalizado de recursos de URL para conjuntos de dados

transformados a partir dos documentos JSON armazenados. Essencialmente, podemos criar a nossa própria API personalizada para os nossos dados com a API do CouchDB.

Termos e conceitos essenciais

Este capítulo discutiu os seguintes termos essenciais:

Banco de dados relacional
: É um tipo de banco de dados estruturado para reconhecer relacionamentos entre os dados armazenados.

Banco de dados NoSQL
: É um tipo de banco de dados que *não* é estruturado com base nos relacionamentos entre os dados armazenados.

CouchDB
: É um banco de dados NoSQL orientado a documentos que armazena dados na forma de documentos JSON.

Também discutimos os seguintes conceitos essenciais:

- Em um banco de dados relacional, normalmente há tabelas, colunas e linhas, além de relacionamentos entre essas tabelas, colunas e linhas. Há chaves primárias (primary keys) e chaves estrangeiras (foreign keys).
- Há vários tipos de bancos de dados NoSQL que romperam com o modelo relacional.

Também discutimos o CouchDB. Eis alguns pontos essenciais a serem lembrados:

- É um tipo de banco de dados NoSQL orientado a documentos.
- Esse banco de dados armazena e administra documentos JSON.
- A estrutura dos dados é mantida conforme esses dados são armazenados e lidos.
- Uma API HTTP é usada para acessar dados na forma de recursos correspondentes a documentos JSON.

- Utiliza JavaScript como sua linguagem de query, com funções de map e reduce para views que podem ser acessadas por meio da API HTTP.

CAPÍTULO 9

JSON no lado servidor

Quando falamos sobre relacionamentos cliente-servidor na Web, podemos classificar as tecnologias em lado cliente e lado servidor:

- No lado cliente, temos o HTML, o CSS e o JavaScript.
- No lado servidor, temos PHP, ASP.NET, Node.js, Ruby, Ruby on Rails, Java, Go e muito mais.

Como um cliente web, enviamos solicitações de recursos a um servidor usando HTTP. O servidor responde com um documento, por exemplo, HTML ou JSON. Quando esse documento for um documento JSON, o código do lado do servidor deverá cuidar de sua criação.

Além de *servir* um documento JSON, um servidor poderá *receber* um documento desse tipo. Vimos isso no capítulo 8, quando usamos o HTTP para fazer post de um documento JSON à API web do CouchDB. Quando um documento é recebido, o código do lado do servidor deve cuidar do parsing desse documento.

Nos capítulos anteriores, discutimos como o JavaScript pode fazer solicitações HTTP nos bastidores, solicitando recursos JSON às APIs web. É fácil pensar no JavaScript desempenhando essa função, pois ele é classificado como um recurso do “lado cliente”. Entretanto, o JavaScript não é a única linguagem que pode fazer solicitações HTTP de recursos JSON. As solicitações HTTP também podem ser feitas por um framework web do lado do servidor.

No cenário em que o JSON é solicitado por uma tecnologia do lado do servidor, seu papel será o de ser cliente. Eis alguns cenários de exemplo em que as solicitações HTTP são feitas pelo código de um framework web do lado do servidor:

- Usar uma API web para dados que preencherão páginas dinâmicas de um site. Pode ser uma API web pública de terceiros ou a sua própria API web privada, como o CouchDB.
- Integrar um gateway de pagamento no código de checkout em um site de e-commerce ao comunicar-se com uma API web.
- Usar uma API web para obter custos de frete em tempo real para entregas em um endereço específico.

Neste capítulo, analisaremos o papel do JSON do lado do servidor. Um framework web do lado do servidor ou uma linguagem de scripting gera páginas web dinâmicas. Quando uma solicitação de um recurso é feita, o servidor cria o recurso usando alguma lógica de programação.

Serializando, desserializando e solicitando JSON

O sucesso de um formato para intercâmbio de dados na Web exige suporte tanto do lado cliente quanto do lado servidor. Se tivéssemos suporte do lado cliente, mas não do lado servidor, o JSON não serviria para nada. Felizmente, o JSON é tratado pela maioria dos frameworks web do lado do servidor e das linguagens de scripting. Caso eles não incluam suporte para a serialização e a desserialização de JSON, é provável que haja uma biblioteca ou uma extensão que faça isso.

Como você deve se lembrar do que vimos no capítulo 6, a serialização é o ato de converter o objeto em texto. A desserialização é o ato de converter o texto de volta em um objeto.

Vamos dar uma olhada no modo como o JSON pode ser serializado, desserializado e solicitado do lado do servidor.

ASP.NET

O ASP.NET é um framework do lado do servidor, desenvolvido pela Microsoft. Originalmente, o desenvolvimento web com esse framework era feito somente com Web Forms, um modelo de GUI orientado a estados. No entanto, atualmente, ele inclui diversas extensões que permitem fazer o desenvolvimento com modelos diferentes, por exemplo, o MVC (Model-

View-Controller, ou Modelo-Visão-Controlador) do ASP.NET e a API web do ASP.NET (uma arquitetura para criar APIs web HTTP para Web Services).

Além disso, o ASP.NET utiliza o CLR (Common Language Runtime), que permite que os desenvolvedores escrevam seus códigos em qualquer linguagem que trate CLR, como C#, F#, Visual Basic .NET (VB.NET) e várias outras. Nesta seção, criaremos todos os exemplos com C#.

De modo diferente do JavaScript, o ASP.NET não faz parse facilmente de JSON. Para fazer parse de JSON, devemos escolher uma biblioteca de terceiros para o ASP.NET e incluí-la em nosso projeto. Na época em que este livro foi escrito, a biblioteca mais popular era a Json.NET – um framework JSON de código aberto da Newtonsoft.

Vamos dar uma olhada no modo como podemos serializar e desserializar dados JSON com a biblioteca Json.NET.

Serialização de JSON

Com o ASP.NET e a Json.NET, podemos serializar rapidamente nossos objetos ASP.NET para JSON. Inicialmente, precisaremos de um objeto com o qual vamos trabalhar. Neste exemplo, criaremos um objeto que representa uma conta, com a mesma estrutura que usamos no capítulo 8 (veja o exemplo 9.1).

EXEMPLO 9.1 – UM OBJETO CUSTOMERACCOUNT EM C# NO ASP.NET

```
public class CustomerAccount
{
   public string firstName { get; set; }
   public string lastName { get; set; }
   public string phone { get; set; }
   public Address[] addresses { get; set; }
   public bool famous { get; set; }
}

public class Address
{
   public string street { get; set; }
   public string city { get; set; }
```

```
    public string state { get; set; }
    public int zip { get; set; }
}
```

Agora que temos uma classe para representar o nosso objeto, vamos criar um novo objeto para a conta de Bob Barker. Na última linha, vamos serializar esse objeto para gerar JSON usando a biblioteca Json.NET (veja os exemplos 9.2 e 9.3).

Exemplo 9.2 – Um novo objeto em C# para a conta de Bob Barker; na última linha, serializaremos este objeto para gerar JSON

```
CustomerAccount bobsAccount = new CustomerAccount();
bobsAccount.firstName = "Bob";
bobsAccount.lastName = "Barker";
bobsAccount.phone = "555-555-5555";

Address[] addresses;
addresses = new Address[2];

Address bobsAddress1 = new Address();
bobsAddress1.state = "123 fakey st";
bobsAddress1.city = "Somewhere";
bobsAddress1.state = "CA";
bobsAddress1.zip = 96520;

addresses[0] = bobsAddress1;

Address bobsAddress2 = new Address();
bobsAddress2.state = "456 fake dr";
bobsAddress2.city = "Some Place";
bobsAddress2.state = "CA";
bobsAddress2.zip = 96538;

addresses[1] = bobsAddress2;

bobsAccount.addresses = addresses;
bobsAccount.famous = true;

string json = JsonConvert.SerializeObject(bobsAccount);
```

Exemplo 9.3 – O JSON resultante da serialização do objeto CustomerAccount no ASP.NET

```
{
```

```
  "firstName": "Bob",
  "lastName": "Barker",
  "phone": "555-555-5555",
  "addresses": [
    {
      "street": null,
      "city": "Somewhere",
      "state": "CA",
      "zip": 96520
    },
    {
      "street": null,
      "city": "Some Place",
      "state": "CA",
      "zip": 96538
    }
  ],
  "famous": true
}
```

Desserialização de JSON

Podemos desserializar o JSON de volta para um objeto ASP.NET com uma linha de código (Exemplo 9.4).

EXEMPLO 9.4 – A DESSERIALIZAÇÃO DA STRING JSON DO EXEMPLO ANTERIOR; ELA SERÁ DESSERIALIZADA DE ACORDO COM O TIPO ESPECIFICADO, QUE É "CUSTOMERACCOUNT"

```
CustomerAccount customerAccount =
  JsonConvert.DeserializeObject<CustomerAccount>(json);
```

A Json.NET provê a capacidade de desserializar o nosso JSON em um tipo específico de objeto, por exemplo, o objeto CustomerAccount do exemplo. Com isso, nossos dados poderão preservar a sua forma à medida que entrarem e saírem do código do lado do servidor. Entretanto, é importante que os nomes das propriedades de nosso objeto coincidam com os nomes dos pares nome-valor de nosso documento JSON. Caso contrário, a desserialização falhará.

Solicitação de JSON

Para fazer uma solicitação HTTP de um recurso, posso usar a classe System.Net.WebClient interna do ASP.NET. Com essa classe, posso chamar o método DownloadString para fazer download do recurso solicitado a partir de um URL especificado como uma string (veja os exemplos 9.5 e 9.6).

Exemplo 9.5 – Fazendo download de um documento JSON de endereços na forma de uma string a partir de meu banco de dados CounchDB local de endereços

```
using(var webClient = new WebClient())
{
   string json = webClient.DownloadString("3636fa3c716f9dd4f7407bd6f700076c");
}
```

Exemplo 9.6 – O recurso JSON no URL http://localhost:5984/accounts/3636fa3c716f9dd4f7407bd6f700076c

```
{
   "_id": "3636fa3c716f9dd4f7407bd6f700076c",
   "_rev": "2-e04207c11104e06b3a8f030f14c35580",
   "address": {
      "street": "456 Fakey Fake st",
      "city": "Somewhere",
      "state": "CA",
      "zip": "96520"
   },
   "age": 54,
   "gender": "female",
   "famous": true,
   "firstName": "Janet",
   "lastName": "Jackson"
}
```

Depois que tivermos o nosso recurso JSON na forma de string, podemos desserializá-lo e gerar um objeto ASP.NET desejado usando o método DeserializeObject de Json.NET (Exemplo 9.7).

Exemplo 9.7 – A desserialização do recurso JSON como uma string; a variável fullName na última linha de código será avaliada como "Janet Jackson"

```
CustomerAccount customerAccount;
```

```
using (var webClient = new WebClient())
{
   string json = webClient.DownloadString(
      "http://localhost:5984/accounts/3636fa3c716f9dd4f7407bd6f700076c");
   customerAccount = JsonConvert.DeserializeObject<CustomerAccount>(json);
}
string fullName = customerAccount.firstName + " " + customerAccount.lastName;
```

Nesses exemplos, podemos ver que a estrutura do objeto JSON o torna compatível com os objetos ASP.NET. Desde que as propriedades do objeto ASP.NET e os nomes dos pares nome-valor do JSON coincidam, serializar e desserializar JSON nesse framework web é simples.

PHP

O PHP é uma linguagem de scripting do lado do servidor e é usado para criar páginas web dinâmicas. O código PHP pode ser diretamente incluído nos documentos HTML (veja o exemplo 9.8).

EXEMPLO 9.8 – ESTA PÁGINA HTML EXIBE UMA TAG DE CABEÇALHO (<H1>) COM O TEXTO "HELLO, WORLD"

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8">
      <title>Hello, World</title>
   </head>
   <body>
      <h1>
      <?php
         echo "Hello, World";
      ?>
      </h1>
   </body>
</html>
```

Além disso, o PHP inclui o tipo de dado objeto. Os objetos são definidos com uma classe (veja o exemplo 9.9).

Exemplo 9.9 – Uma classe PHP que representa um gato

```
class Cat
{
   public $name;
   public $breed;
   public $age;
   public $declawed;
}
```

Quando uma classe é instanciada, um objeto é criado. O objeto pode então ser usado na lógica de programação (Exemplo 9.10).

Exemplo 9.10 – A classe é instanciada e as propriedades são definidas. Um objeto é criado. A última linha de código exibirá "Fluffy Boo".

```
$cat = new Cat();
$cat->name = "Fluffy Boo";
$cat->breed = "Maine Coon";
$cat->age = 2.5;
$cat->declawed = false;

echo $cat->name;
```

O PHP também inclui suporte interno para a serialização e a desserialização de JSON e refere-se a essas funções como codificação (encoding) e decodificação (decoding) de JSON. Quando codificamos algo, fazemos uma conversão para um formato codificado (ilegível). Ao decodificar algo, fazemos a conversão de volta para um formato legível. Do ponto de vista do PHP, o JSON está em um formato codificado. Sendo assim, para serializar dados JSON, a função json_encode deve ser chamada; para desserializar dados JSON, a função json_decode deverá ser chamada.

Serialização de JSON

Em PHP, podemos serializar objetos PHP rapidamente com o suporte pronto fornecido para JSON. Nesta seção, criaremos um objeto PHP que representa um endereço e faremos a serialização para gerar dados JSON.

No exemplo 9.11, inicialmente criamos uma classe que representa uma conta. Em seguida, criamos uma nova instância da classe para a conta de Bob

Barker. Um objeto Account é criado. Por fim, na última linha, chamamos a função interna json_encode para serializar o objeto Account (o resultado está sendo mostrado no exemplo 9.12).

EXEMPLO 9.11 – CRIANDO E SERIALIZANDO UMA CONTA

```
<?php
class Account {
   public $firstName;
   public $lastName;
   public $phone;
   public $gender;
   public $addresses;
   public $famous;
}

class Address {
   public $street;
   public $city;
   public $state;
   public $zip;
}

$address1 = new Address();
$address1->street = "123 fakey st";
$address1->city = "Somewhere";
$address1->state = "CA";
$address1->zip = 96027;

$address2 = new Address();
$address2->street = "456 fake dr";
$address2->city = "Some Place";
$address2->state = "CA";
$address2->zip = 96345;

$account = new Account();
$account->firstName = "Bob";
$account->lastName = "Barker";
$account->gender = "male";
$account->phone = "555-555-5555";
$account->famous = true;
```

```
$account->addresses = array ($address1, $address2);
$json = json_encode($account);
?>
```

EXEMPLO 9.12 – O RESULTADO DE JSON_ENCODE($ACCOUNT) NO FORMATO JSON

```
{
   "firstName": "Bob",
   "lastName": "Barker",
   "phone": "555-555-5555",
   "gender": "male",
   "addresses": [
      {
         "street": "123 fakey st",
         "city": "Somewhere",
         "state": "CA",
         "zip": 96027
      },
      {
         "street": "456 fake dr",
         "city": "Some Place",
         "state": "CA",
         "zip": 96345
      }
   ],
   "famous": true
}
```

Desserialização de JSON

Para serializar dados JSON em PHP, usamos a função interna json_encode. Para desserializar dados JSON, usamos a função json_decode. Infelizmente, não há suporte pronto para desserialização de JSON em um objeto PHP especificado, por exemplo, a classe Account. Desse modo, devemos realizar um pequeno processamento para reformatar nossos dados de volta no objeto PHP.

Vamos adicionar uma nova função ao objeto Account para carregar suas propriedades a partir de uma string JSON.

No exemplo 9.13, a função loadFromJSON aceita uma string JSON como parâmetro, chama a função interna json_decode para desserializar e gerar um objeto PHP genérico e mapeia os pares nome-valor às propriedades de Account de acordo com o nome em um loop foreach.

EXEMPLO 9.13 – ACRESCENTANDO UMA FUNÇÃO AO OBJETO ACCOUNT

```
class Account {
   public $firstName;
   public $lastName;
   public $phone;
   public $gender;
   public $addresses;
   public $famous;

   public function loadFromJSON($json)
   {
      $object = json_decode($json);
      foreach ($object AS $name => $value)
      {
         $this->{$name} = $value;
      }
   }
}
```

Em seguida, podemos criar um novo objeto Account e chamar a nova função loadFromJSON (Exemplo 9.14).

EXEMPLO 9.14 – CHAMANDO NOSSA NOVA FUNÇÃO LOADFROMJSON PARA DESSERIALIZAR A CONTA JSON DE VOLTA EM UM OBJETO ACCOUNT; A ÚLTIMA LINHA EXIBIRÁ "BOB BARKER"

```
$json = json_encode($account);

$deserializedAccount = new Account();
$deserializedAccount->loadFromJSON($json);

echo $deserializedAccount->firstName . " " . $deserializedAccount->lastName;
```

Solicitação de JSON

Para fazer uma solicitação HTTP de um recurso com o PHP, podemos usar a função interna file_get_contents. Essa função retorna o corpo do recurso como

uma string. Podemos então desserializar a string e gerar um objeto PHP (veja os exemplos 9.15 e 9.16).

Exemplo 9.15 – Recurso no URL http://localhost:5984/accounts/ddc14efcf71396463f53c0f8800019ea da API de meu CouchDB local

```
{
   "_id": "ddc14efcf71396463f53c0f8800019ea",
   "_rev": "6-69fd853972074668f99b88a86aa6a083",
   "address": {
      "street": "123 fakey ln",
      "city": "Some Place",
      "state": "CA",
      "zip": "96037"
   },
   "gender": "female",
   "famous": false,
   "age": 28,
   "firstName": "Mary",
   "lastName": "Thomas"
}
```

Exemplo 9.16 – Chamando a função interna file_get_contents para obter o recurso JSON com a conta a partir da API do CouchDB. Em seguida, um novo objeto Account é criado e a nossa função loadFromJSON é chamada para fazer a desserialização. A última linha exibirá "Mary Thomas".

```
$url = "http://localhost:5984/accounts/3636fa3c716f9dd4f7407bd6f700076c";
$json = file_get_contents($url);

$deserializedAccount = new Account();
$deserializedAccount->loadFromJSON($json);

echo $deserializedAccount->firstName . " " . $deserializedAccount->lastName;
```

Uma infinidade de solicitações HTTP para JSON

Tanto no exemplo com PHP quanto no exemplo com ASP.NET, podemos ver que, embora o JSON seja baseado na notação de objetos literais da linguagem JavaScript, a notação é devidamente traduzida para objetos em outras

linguagens de programação. Como as linguagens de programação do lado do servidor são muitas e bastante diversificadas, várias delas aceitam objetos e os tipos de dados do JSON. Isso é ideal, pois permite que a estrutura ou a "forma" dos dados seja preservada à medida que esses dados entram e saem de um sistema como dados JSON.

Além disso, o suporte ao JSON do lado servidor é amplo. A maioria das linguagens (ou frameworks) do lado do servidor oferece suporte interno para serialização/desserialização de JSON ou tem bibliotecas ou módulos disponíveis para isso. Esse suporte ao JSON do lado do servidor contribui para o seu sucesso como um formato para intercâmbio de dados.

Vamos dar uma olhada em alguns exemplos pequenos de solicitações HTTP de JSON do lado do servidor. Em cada um desses exemplos, veremos uma solicitação HTTP de um recurso JSON, o parsing desse recurso para um objeto e a apresentação de um valor de um documento JSON.

O documento JSON em todos os exemplos será obtido da API OpenWeatherMap. Toda a apresentação dos valores será baseada em um subconjunto do recurso JSON em *http://api.openweathermap.org/data/2.5/weather?q=London,uk* (veja o exemplo 9.17).

EXEMPLO 9.17 – O OBJETO DE COORDENADAS DO RECURSO JSON PARA OS DADOS DE CLIMA DA API OPENWEATHERMAP; ESSE OBJETO REPRESENTA A LATITUDE E A LONGITUDE DE LONDRES NA INGLATERRA

```
{
   "coord": {
      "lon": -0.13,
      "lat": 51.51
   }
}
```

Ruby on Rails

O Ruby on Rails é um framework de aplicação web para o lado servidor. É escrito na linguagem de programação Ruby e baseado na arquitetura MVC (Model-View-Controller, ou Modelo-Visão-Controlador).

Em Ruby, uma gem é um pacote para instalar um programa ou uma biblioteca. Para serializar e desserializar dados JSON no Ruby on Rails, a gem JSON será necessária. Após ter a gem JSON instalada, fazer parsing de JSON é simples: basta usar JSON.parse() (Exemplo 9.18).

No exemplo 9.18, criamos uma solicitação HTTP para a API OpenWeatherMap e desserializamos o JSON para gerar um objeto Ruby. Por fim, na última linha de código, apresentamos a longitude (lon) do objeto de coordenadas (coord) contido nos dados.

EXEMPLO 9.18 – SOLICITAÇÃO HTTP À API OPENWEATHERMAP

```
require 'net/http'
require 'json'

url = URI.parse('http://api.openweathermap.org/data/2.5/weather?q=London,uk')
request = Net::HTTP::Get.new(url.to_s)
response = Net::HTTP.start(url.host, url.port) {|http|
http.request(request)
}

weatherData = JSON.parse(response.body)

render text: weatherData["coord"]["lon"]
```

Node.js

O Node.js é um JavaScript do lado do servidor (sem um navegador de Internet), que é possível por causa da engine JavaScript V8 de código aberto do Google. Com o Node.js, podemos criar aplicações do lado do servidor com o JavaScript.

Anteriormente neste livro, discutimos o JSON com o JavaScript no lado cliente e vimos que o JSON é desserializado em um objeto JavaScript com uma simples chamada a JSON.parse(). O mesmo vale para o Node.js, pois ele *é* JavaScript.

No entanto, não usamos o objeto XMLHttpRequest no Node.js, pois ele é um objeto JavaScript específico para os navegadores de Internet. No Node.js, podemos solicitar o nosso JSON (e outros tipos de recurso) com uma função cujo nome é muito mais adequado: get().

No exemplo 9.19, criamos uma solicitação HTTP para a API OpenWeatherMap e desserializamos o JSON para gerar um objeto JavaScript. Em seguida, exibimos a longitude (lon) do objeto de coordenadas (coord) contido nos dados usando console.log().

EXEMPLO 9.19 – FAZENDO UMA SOLICITAÇÃO HTTP À API E DESSERIALIZANDO PARA GERAR UM OBJETO JAVASCRIPT

```
var http = require('http');
http.get({
   host: 'api.openweathermap.org',
   path: '/data/2.5/weather?q=London,uk'
}, function(response) {
   var body = '';
   response.on('data', function(data) {
      body += data;
   });
   response.on('end', function() {
      var weatherData = JSON.parse(body);
      console.log(weatherData.coord.lon);
   });
});
```

Java

O Java é uma linguagem de programação orientada a objetos. Essa linguagem pode ser executada no navegador web como applets Java ou de forma independente em um computador com JRE (Java Runtime Environment).

Há várias bibliotecas disponíveis que oferecem suporte a JSON em Java. Nesta seção, usarei as bibliotecas a seguir para obter o recurso JSON de um URL e desserializá-lo para gerar um objeto em Java:

- Apache Commons IO (*https://commons.apache.org/proper/commons-io/*)
- JSON in Java (*http://www.json.org/java/*)

No exemplo 9.20, recebemos o recurso JSON da API OpenWeatherMap na forma de string. Depois disso, desserializamos a string JSON ao instanciar um JSONObject. Então exibimos a longitude (lon) do objeto de coordenadas

(coord) contido nos dados usando System.out.println().

EXEMPLO 9.20 – DESSERIALIZANDO UMA STRING JSON COM UM JSONOBJECT

```
import java.io.IOException;
import java.net.URL;

import org.apache.commons.io.IOUtils;
import org.json.JSONException;
import org.json.JSONObject;

public class HelloWorldApp {

    public static void main(String[] args) throws IOException, JSONException {
        String url = "http://api.openweathermap.org/data/2.5/weather?q=London,uk";
        String json = IOUtils.toString(new URL(url));
        JSONObject weatherData = new JSONObject(json);
        JSONObject coordinates = weatherData.getJSONObject("coord");
        System.out.println(coordinates.get("lon"));
    }
}
```

Termos e conceitos essenciais

Este capítulo discutiu os seguintes termos essenciais:

ASP.NET
: É um framework web do lado do servidor, desenvolvido pela Microsoft.

PHP
: É uma linguagem de scripting do lado do servidor, usada para criar páginas web dinâmicas.

Ruby on Rails
: É um framework de aplicações web do lado do servidor que executa em Ruby.

Node.js
: É um JavaScript do lado do servidor, executado na engine JavaScript V8 do Google.

Java

É uma linguagem de programação orientada a objetos.

Também discutimos os seguintes conceitos essenciais:

- Do lado do servidor, o JSON pode ser desserializado, gerando um objeto que poderá ser usado na lógica de programação, e serializado no formato JSON a partir de um objeto.
- O JSON tem um amplo suporte *tanto* do lado cliente *quanto* do lado servidor, o que o torna um formato para intercâmbio de dados muito bem-sucedido na Web.

CAPÍTULO 10

Conclusão

De uma altura de aproximadamente dois mil metros, podemos ver o JSON circulando pelo mundo, transportando dados para dentro e para fora dos sistemas. O JSON é muito bom para realizar o seu trabalho como formato para intercâmbio de dados, e vimos isso ser expresso neste livro. No entanto, se você observar atentamente com seus binóculos, verá que o JSON permanece imóvel em alguns lugares.

Apesar dos fenômenos relacionados ao fato de o JSON permanecer imóvel em alguns locais, este livro, em sua maior parte, ignorou esse assunto. Focamos no intercâmbio de dados entre duas partes. Essencialmente, focamos nos dados sendo transportados.

O JSON pode ser tanto um receptáculo quanto um veículo para os dados e não está restrito a um único propósito. Vimos isso ilustrado por meio da função do JSON no NoSQL. Cada vez mais, o JSON está conquistando um espaço em projetos e em empreendimentos como uma ferramenta bastante útil. Para ilustrar melhor, vamos dar uma última olhada em uma função desempenhada pelo JSON no mundo atual: permanecer imóvel em um local para ser usado como um arquivo de configuração.

JSON como um arquivo de configuração

No capítulo 9, exploramos duas tecnologias web do lado do servidor que tratam o parsing de JSON. Conforme discutido naquele capítulo, o JSON tem um amplo suporte por parte da maioria das linguagens web (ou dos frameworks) do lado do servidor. Usar o JSON para armazenar dados de configuração tornou-se popular devido ao amplo suporte que ele tem, à facilidade de parsing no servidor e à legibilidade por parte dos seres

humanos.

Um arquivo de configuração ou de parâmetros geralmente é usado pelo software de modo que as configurações possam ser alteradas sem que uma recompilação seja necessária. Há diversos formatos de arquivos de configuração, incluindo INI e XML.

Vamos dar uma olhada no modo como podemos armazenar as mesmas informações de configuração nos formatos INI, XML e JSON (nos exemplos 10.1, 10.2 e 10.3, respectivamente).

EXEMPLO 10.1 – EXEMPLO DE UM ARQUIVO DE CONFIGURAÇÃO DE UM JOGO EM FORMATO INI (SETTINGS.INI)

```
[general]
playIntro=false
mouseSensitivity=0.54

[display]
complexTextures=true
brightness=4.2
widgetsPerFrame=326
mode=windowed

[sound]
volume=1
effects=0.68
```

EXEMPLO 10.2 – EXEMPLO DE UM ARQUIVO DE CONFIGURAÇÃO DE UM JOGO EM FORMATO XML (SETTINGS.XML)

```
<?xml version="1.0" encoding="UTF-8" ?>
<settings>
   <general>
      <playIntro>false</playIntro>
      <mouseSensitivity>0.54</mouseSensitivity>
   </general>
   <display>
      <complexTextures>true</complexTextures>
      <brightness>4.2</brightness>
      <widgetsPerFrame>326</widgetsPerFrame>
   </display>
```

```
    <sound>
        <volume>1</volume>
        <effects>0.68</effects>
    </sound>
</settings>
```

EXEMPLO 10.3 – EXEMPLO DE UM ARQUIVO DE CONFIGURAÇÃO DE UM JOGO EM FORMATO JSON (SETTINGS.JSON)

```
{
    "general": {
        "playIntro": false,
        "mouseSensitivity": 0.54
    },
    "display": {
        "complexTextures": true,
        "brightness": 4.2,
        "widgetsPerFrame": 326,
        "mode": "windowed"
    },
    "sound": {
        "volume": 1,
        "effects": 0.68
    }
}
```

Todos esses três formatos são legíveis aos seres humanos. Se eu quiser alterar as configurações de som, poderei rapidamente localizar essa informação em cada um dos arquivos e mudar os números. É isso que torna esses arquivos ótimos para configuração.

Cada um desses formatos tem seus prós e contras. O arquivo INI é mais fácil de ser lido por um ser humano, pois não contém os símbolos de menor e de maior do XML nem as chaves e as aspas duplas do JSON. O INI não é tão bom assim quando se trata de dados mais complexos, como informações aninhadas ou listas complicadas. O XML pode acomodar dados mais complexos, porém não tem tipos de dados como o JSON.

Além dos prós e dos contras de cada formato de dados, a facilidade de fazer

parsing pela linguagem ou framework usado também é um aspecto importante a ser considerado. Se um parser JSON já estiver sendo intensamente utilizado pela sua aplicação, o JSON poderá ser a melhor opção para o seu arquivo de configuração.

No capítulo 9, demos uma olhada rapidamente no Node.js com o JSON do lado do servidor. Um bom exemplo de JSON como arquivo de configuração no mundo real está no gerenciador de pacotes JavaScript que é default para o Node.js: o npm. Ele também é usado por outros frameworks, como o AngularJS e a jQuery.

O gerenciador de pacotes npm utiliza o JSON em um arquivo de configuração chamado *package.json*. O arquivo *package.json* contém informações específicas de cada pacote, por exemplo, o nome, a versão, o autor, os colaboradores, as dependências, os scripts e a licença (veja o exemplo 10.4).

EXEMPLO 10.4 – UM ARQUIVO PACKAGE.JSON DE EXEMPLO PARA O NPM

```
{
  "name": "bobatron",
  "version": "1.0.0",
  "description": "The extraordinary library of Bob.",
  "main": "bob.js",
  "scripts": {
    "prepublish": "coffee -o lib/ -c src/bob.coffee"
  },
  "repository": {
    "type": "git",
    "url": "git://github.com/bobatron/bob.git"
  },
  "keywords": [
    "bob",
    "tron"
  ],
  "author": "Bob Barker <bob.barker@fakemail.com> (http://bob.com/)",
  "license": "BSD-3-Clause",
  "bugs": {
    "url": "https://github.com/bobatron/issues"
```

```
    }
}
```

Embora o JSON permaneça imóvel em um só local como um arquivo de configuração, ele continua desempenhando a função de formato para intercâmbio de dados. No caso de um arquivo de configuração que esteja em um diretório em algum lugar, o intercâmbio ocorre entre o ser humano e o computador. Os dados são comunicados.

Quadro geral

Independentemente de ser um arquivo de configuração que permaneça em um servidor ou um recurso solicitado por um URL, o JSON faz o seu trabalho como um formato para intercâmbio de dados. Exploramos essas tarefas (ou funções) nos capítulos deste livro. Vamos fazer uma recapitulação do quadro geral para ver como o JSON está sendo usado no mundo atualmente.

Do lado do servidor, os objetos podem ser serializados para o formato-texto JSON e desserializados de volta em um objeto. O JSON também pode ser solicitado pelo código do lado do servidor. Vimos isso em ação nos exemplos tanto com o ASP.NET quanto com o PHP discutidos no capítulo 9.

Além disso, o JSON é um formato-texto que também pode ser usado como um documento em um banco de dados orientado a documentos. Vimos isso no capítulo 8 com o CouchDB. Esse banco de dados também disponibiliza uma interface por meio de uma API web HTTP.

Um API web HTTP oferece um sistema de solicitação e de resposta para recursos como HTML e documentos JSON. Esses documentos são solicitados com um URL por meio de HTTP. Também vimos isso no capítulo 6 com a API OpenWeatherMap fornecendo dados de clima na forma de recursos JSON.

O `XmlHttpRequest` do JavaScript pode solicitar um recurso JSON a partir de um URL. Para usar o JSON no código JavaScript, devemos desserializá-lo para gerar um objeto JavaScript. Isso é feito rapidamente com a função interna `JSON.parse()` do JavaScript.

De um objeto JavaScript ao formato JSON e para um objeto no servidor podemos ver os dados moverem-se em um ciclo completo. Todos os dias, os dados entram e saem dos sistemas pelo mundo, e esses dados estão em um formato que permite fazer um intercâmbio.

Se dermos uma olhada novamente de uma altura de aproximadamente dois mil metros, veremos que o JSON não é o único formato para intercâmbio de dados. Alguns dados são transferidos em um formato CSV (Comma-Separated Values, ou Valores separados por vírgula) ou em um formato Excel. Ambos são formatos tabulares. Outros dados são transferidos em formato XML, que aceita dados aninhados.

Os dados têm muitos formatos e formas. Os sistemas que enviam e recebem dados nesses formatos são diversificados. Tanto a forma dos dados quanto os sistemas que trocam esses dados devem ser considerados ao escolher um formato para intercâmbio de dados. Apesar da paixão deste livro pelo JSON e de vê-lo como um formato incrível para os dados, o JSON nem sempre é a resposta.

Por exemplo, vamos supor que dois sistemas devam transmitir dados de estoque e que esses dois sistemas armazenem essas informações em um formato tabular. Faria sentido converter esses dados tabulares em um formato de objeto, serializar para JSON, converter de volta em um formato de objeto e então gerar novamente os dados tabulares? Não; nesse caso, estaríamos acrescentando passos extras somente por causa do JSON. Um formato tabular será mais apropriado para esses dados, por exemplo, o CSV ou um formato delimitado por tabs.

O JSON é um formato popular para intercâmbio de dados porque muitos sistemas modelam seus dados na forma de objetos. Os dados geralmente são armazenados em um banco de dados relacional. Embora as tabelas dos bancos de dados relacionais sejam tabulares, os relacionamentos entre os dados estão separados em entidades que podem ser vistas como objetos. Cada entidade, por exemplo, um endereço, tem “campos” que, essencialmente, são pares nome-valor.

Em sistemas como um navegador de Internet, em que o JavaScript oferece suporte para a programação orientada a objetos, o JSON será ideal para o

intercâmbio de dados. O JSON é ótimo para ser usado com o JavaScript a fim de fazer a comunicação com APIs web e criar interações AJAX.

Em sistemas como um framework web do lado do servidor que seja orientado a objetos, o JSON será ideal para a troca de dados. O JSON se destaca ao manter a estrutura dos dados como um objeto literal à medida que esses dados entram e saem dos sistemas.

Como no exemplo da seção “JSON como um arquivo de configuração”, os prós e os contras devem ser analisados para selecionar um formato. Tudo se reduz a usar a ferramenta certa para o trabalho correto.

Sobre a autora

Lindsay Bassett tem paixão tanto por escrever sobre tecnologia quanto por ensiná-la. Seus cursos online de tecnologia e os livros compartilham o mesmo estilo “direto ao ponto”, que agrada os profissionais ocupados de TI ou os estudantes. Ela destila cada assunto, extraindo seus fundamentos, e tem um talento especial para transformar assuntos técnicos monótonos em uma leitura leve e agradável.

Colofão

O animal na capa de *Introdução ao JavaScript Object Notation* é uma foca comum (*Phoca vitulina*), um mamífero marinho normalmente encontrado em águas temperadas e árticas ao longo da costa da América do Norte e da Europa. Elas são “focas de verdade”, o que significa que têm nadadeiras dianteiras curtas, corpos robustos em forma de linguiça e não têm orelhas externas.

Cada foca pode ser identificada por um padrão único de manchas e a sua coloração varia: elas podem ser prateadas, cinzas, marrons, cinza-azuladas ou castanhas, com manchas escuras ou claras, conforme a cor principal de sua pelagem. O comprimento médio do corpo é de aproximadamente 1,8 metros e o peso pode variar de 55 a 170 quilogramas; as fêmeas são um pouco menores que os machos.

Embora as focas possam parecer desajeitadas ao se arrastarem pelo chão, seu corpo liso e as nadadeiras traseiras robustas as tornam ágeis e formidáveis na água. Sua dieta varia de acordo com a estação à medida que elas mudam de ambientes para se alimentar, porém as focas comuns geralmente consomem uma variedade de peixes, lulas, crustáceos e moluscos. Elas podem mergulhar até aproximadamente 450 metros e podem permanecer até 40 minutos embaixo da água, porém mergulhos mais rasos, com duração de 3 a 7 minutos, são mais regulares.

As focas comuns passam aproximadamente metade do tempo em ambiente terrestre – principalmente para descansar e regular a sua temperatura, mas também para promover interação social e parir seus filhotes. As focas geralmente têm locais favoritos para descanso e para a criação dos filhotes, porém abandonarão esse lugar se houver distúrbios frequentes por parte dos seres humanos. Por esse motivo, o desenvolvimento de áreas costeiras próximas às praias conhecidas como locais de permanência das focas deve ser cuidadosamente considerado.

Os filhotes das focas comuns sabem nadar desde que nascem e, em poucos dias, podem permanecer embaixo da água por até dois minutos. Eles são desmamados após quatro a seis semanas, porém quase dobram de peso nesse período devido ao alto teor de gordura do leite materno.

Muitos dos animais nas capas dos livros da O’Reilly estão ameaçados; todos eles são importantes para o mundo. Para saber como você pode ajudar, acesse *animals.oreilly.com*. A imagem da capa foi retirada do livro *Royal Natural History* de *Lydekker*.

O'REILLY

Engenharia de Confiabilidade do Google

COMO O GOOGLE ADMINISTRA SEUS SISTEMAS DE PRODUÇÃO

Site Reliability Engineering (SRE)

Editado por Betsy Beyer, Chris Jones, Jennifer Petoff e Niall Richard Murphy

novatec

Engenharia de Confiabilidade do Google

Beyer, Betsy

9788575227527

632 páginas

A maior parte do tempo de vida de um sistema de software se dá em seu uso, e não no design ou na implementação. Então, por que a sabedoria convencional insiste que os engenheiros de software devam se concentrar principalmente nas fases de design e de desenvolvimento dos sistemas computacionais de larga escala? Nesta coletânea de dissertações e artigos, membros essenciais da equipe de SRE (Site Reliability Engineering – Engenharia de Confiabilidade) do Google explicam como e por que seu comprometimento com todo o ciclo de vida tem permitido que a empresa desenvolva, implante, monitore e mantenha alguns dos maiores sistemas de software do mundo com sucesso. Você conhecerá os princípios e as práticas que possibilitam aos engenheiros do Google deixar os sistemas mais escaláveis, confiáveis e eficientes – lições que podem ser diretamente aplicáveis à sua empresa. Este livro está dividido em quatro partes: •Introdução – Saiba o que é SRE e por que ela difere das práticas convencionais do mercado de TI. •Princípios – Analise os padrões, os comportamentos e as áreas de preocupação que influenciam o trabalho de um SRE (Site Reliability Engineer – Engenheiro de Confiabilidade). •Práticas – Entenda a teoria e a prática do trabalho cotidiano de um SRE: desenvolver e operar sistemas computacionais distribuídos de grande porte. •Gerenciamento

– Explore as melhores práticas do Google para treinamento, comunicação e reuniões, que poderão ser usadas pela sua empresa.

Compre agora e leia

O'REILLY

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações nativas de nuvem

Bilgin Ibryam
Roland Huß

novatec

Padrões para Kubernetes

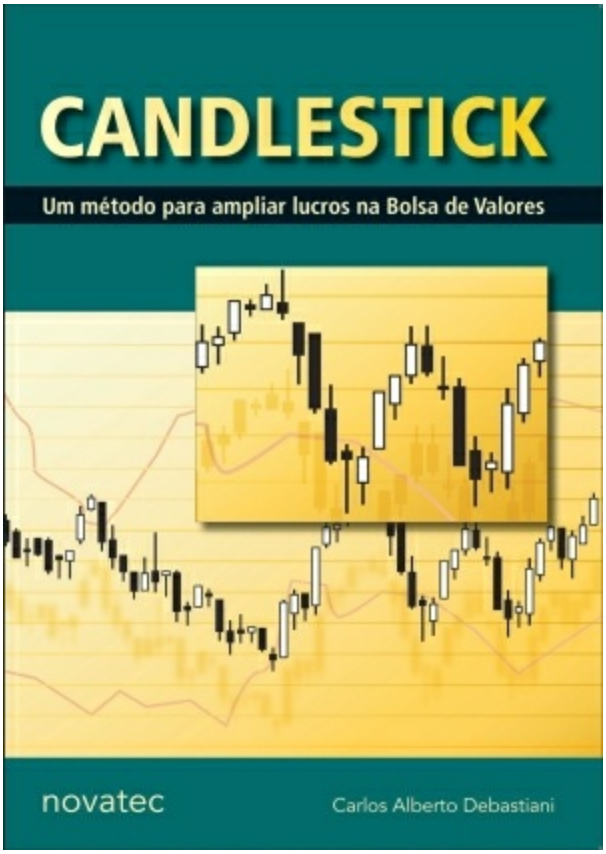
Ibryam, Bilgin

9788575228159

272 páginas

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões: • Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres. • Padrões comportamentais, que exploram conceitos mais específicos para administrar contêineres e interações com a plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de

configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).


CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

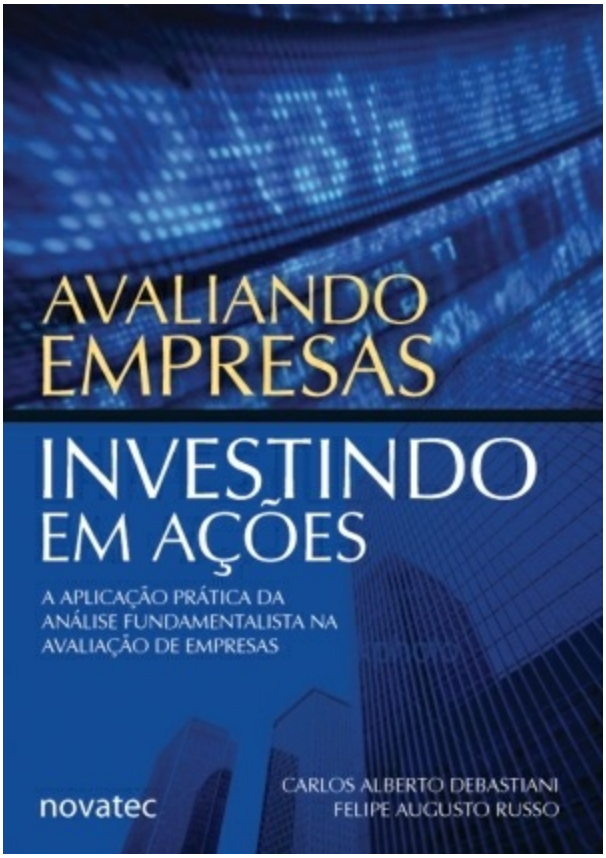
Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

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.


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

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

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.