Author Picks

Using the Web to Build the IoT

Chapters selected by
Dominique D. Guinard and Vlad M. Trifa

manning

Using the Web to Build the IoT

Selections by Dominique D. Guinard
and Vlad M. Trifa

Manning Author Picks

To pre-order or learn more about these books go to www.manning.com

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN 9781617294006
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

contents

introduction

This collection of chapters examines one of most important new waves in computing: the Internet of Things (IoT)! Capturing the essence of the IoT in one sentence is nearly impossible. It has become such a hot topic that there are no clear boundaries between what the IoT is and what it isn't. Broadly speaking, the IoT vision is of a world where the internet is much more than the bunch of multimedia content it is today—where it extends into the physical, real-time world using a myriad of tiny computers. The simplest definition we can offer is the following: The Internet of Things is a system of physical objects that can be discovered, monitored, controlled, or interacted with by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider internet.

The concept has been around since 1999, but the IoT has not yet truly materialized. Yes, we have smart devices in our homes that can be controlled via mobile phones. We have smart thermostats that are aware of our location. We have smart scales that can help us manage our weight and fitness trackers that motivate us to move. Yet, all these devices largely exist in isolation. To put it bluntly, the Internet of Things of today is essentially a growing collection of isolated Intranets of Things that can't be connected to each other. No need to worry too much—the internet itself went through a similar phase. It started as a network of computers that used multiple incompatible protocols to communicate with one another. It formed a network of connected computers, but without standard ways of building applications on top of this network the use of the internet was rather limited! Then came the web: a simple and universal application. The web allowed the internet to evolve from a network of computers exchanging bits of data to a world-wide service platform accessible through standard and universally understood protocols such as HTTP. Similarly, the Internet of Things desperately needs its own application layer to truly blossom. Just like the internet needed the web, the IoT needs a set of standards that applications can use to control, monitor, and aggregate the data of connected things; otherwise it is likely to remain an Intranet of isolated Things! We could reinvent the wheel once more, but because the web proved to be the most scalable, flexible, and versatile application

layer out there, why shouldn't we reuse it for the IoT? This is what we call the Web of Things!

The concept of the Web of Things (WoT) is fairly straightforward: it explores how we can re-use the goodness of the web to make these tiny computers talk together and push their data to places where it can be leveraged to build truly ground-breaking applications! To better grasp the different technologies that can be involved in making the IoT an integral part of the web, in our book *Building the Web of Things*, we create four layers for the WoT architecture: Access, Find, Share & Secure, and Compose (see figure 1). Each layer solves a set of problems using web technologies for the layer above it. For example, the Access layer is all about creating web APIs for Things, while the Find layer assumes these APIs exist and deals with making them discoverable and findable on the web.

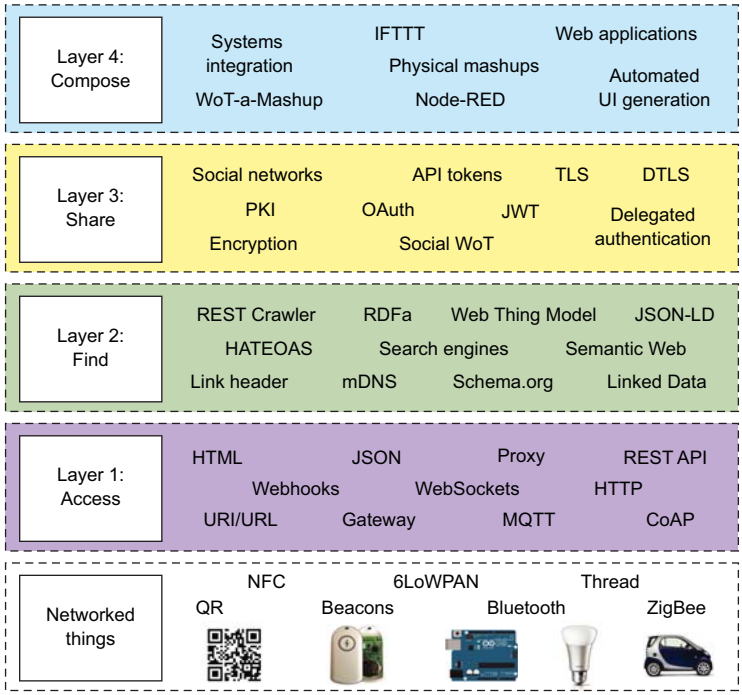


Figure 1 The Web of Things architecture stack with its 4 layers.

This architecture probably makes *Building the Web of Things* the first comprehensive toolbox for building the WoT. However, each layer could entail an entire book of its own! That's what this collection of chapters is all about. We've borrowed chapters from other great Manning books that are spot-on when it comes to illustrating our WoT architecture and building the application layer of the IoT!

A chapter from *Streaming Data: Designing the real-time pipeline* by Andrew G. Psaltis builds on the Access layer by looking into data collection, patterns, and protecting from data loss.

Linked Data: Structured data on the Web by David Wood, Marsha Zaidman, Luke Ruth, and Michael Hausenblas takes you into the Find layer with an in-depth look at Resource Description Framework in Attributes.

In the Share layer, we delve into keeping code bug-free, dealing with attacks, and auditing code with a chapter from *Express in Action: Node applications with Express and its companion tools* by Evan M. Hahn.

The final layer, Compose, is illustrated with case study-like examples and in-depth visualizations from *Real-World Machine Learning* by Henrik Brink, Joseph W. Richards, and Mark Fetherolf, and from *D3.js in Action* by Elijah Meeks.

The Access layer

The Access layer is the most fundamental because it looks into the way Things can be connected to the web by offering a web API. This layer is responsible for turning any Thing into a programmable web Thing that other devices and applications can easily talk to. The core idea of this level is simple: how can Things be smoothly integrated into the web by exposing their services through a RESTful API using HTTP, built on top of TCP/IP as well as the JSON data format. The Access layer also describes how to use WebSockets to accommodate the fact that a number of IoT use cases are real-time or event-driven. Because not all Things will be able to speak web protocols or even be connected to the internet, the Access layer also looks into the web integration of non-web and non-internet Things using several integration patterns such as gateways.

To better illustrate why bringing Things to the web is really powerful, we picked the chapter "Hello, World Wide Web of Things" from *Building the Web of Things*. In this chapter, you'll see how you can program applications using embedded devices with simple and powerful JavaScript code instead of having to use complex embedded programming!

Chapter 2 from *Building the Web of Things*
by Dominique D. Guinard and Vlad M. Trifa

Hello, World Wide Web of Things

This chapter covers

- A sneak peek at the different levels of the Web of Things architecture
- Accessing devices with HTTP, URLs, and browsers
- Working with REST APIs to expose JSON data
- Learning about the idea of semantics on the web
- Creating your first physical mashup

Before we dive head first into the Web of Things architecture and show how to build it from scratch, we want to give you a taste of what the Web of Things looks like. This chapter is structured as a set of exercises where you'll build tiny web applications that use data generated by a real device. Each exercise will be a smooth introduction to the many problems and technical issues that you'll face when building web-connected devices and the applications around them.

In this chapter, you'll have the opportunity to get your hands dirty and code some simple (and less simple) Web of Things applications. Oh, you don't have a device yet? No problem; just use ours! To make it possible for you to do those exercises

without having a real device nearby, we connected our own device to the web so you can connect to it from your computer. Of course, if you already have a device, you can also download the source code used in this chapter and run it on your own device. How to run the code on the device will be detailed later, in chapter 7.

2.1 Meet a Web of Things device

This chapter is organized as a series of short and sweet exercises that illustrate the various difficulties and problems you'll learn how to solve in the next chapters. Each exercise allows you to interact with an actual Web of Things device in our office that's live 24/7. This will allow you to do the exercises without having a real device next to you.

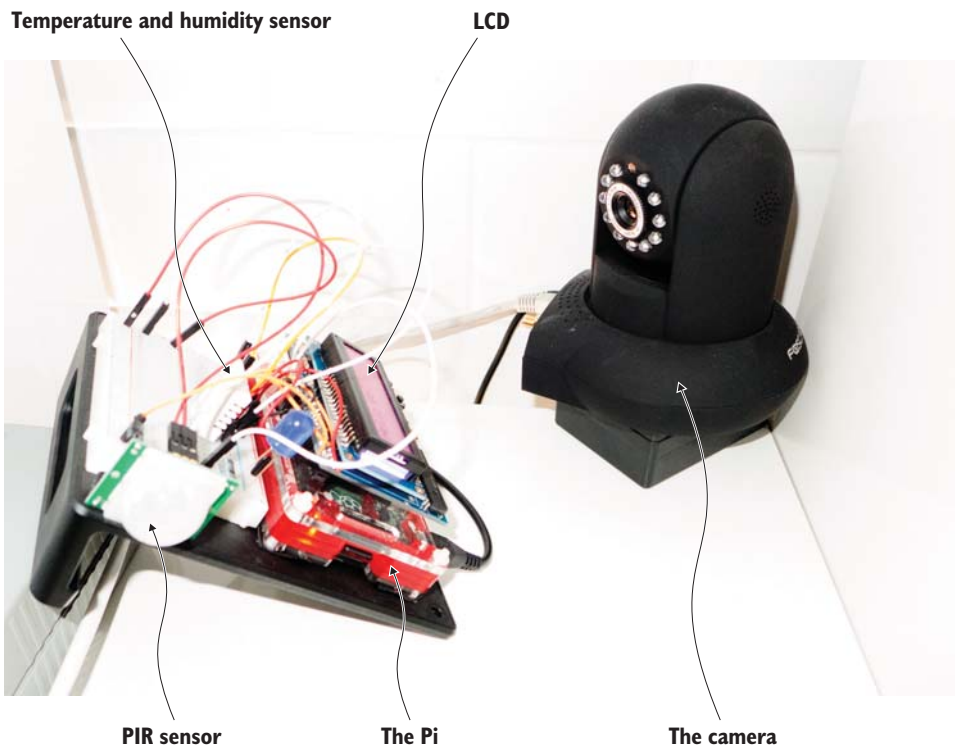


Figure 2.1 The Raspberry Pi and webcam you are accessing as they are set up in our London office

The device in our office is the Raspberry Pi 2 (or just Pi for friends and family) shown in figure 2.1, which we'll describe in detail in chapter 4. If you've never seen one, you can simply think of a credit card–sized computer board with a few sensors attached to it and connected to our local network and the web via an Ethernet cable. In chapter 7, we'll describe what gateways are in the Web of Things and help you build your own, but for now just imagine it's a somewhat intelligent proxy or, in more detail, a server that abstracts the access to other servers, hiding some of the complexity to the clients, as shown in figure 2.2.

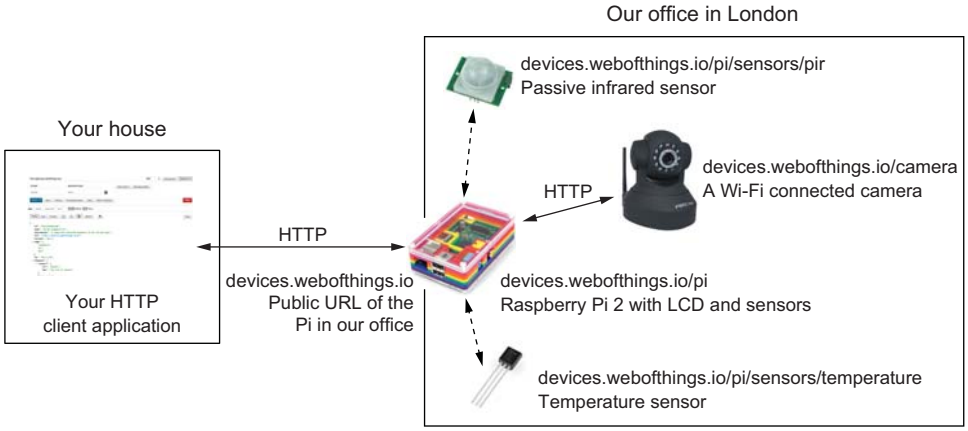


Figure 2.2 The setup of devices and sensors used in the examples of this chapter

At the time of writing, we have a liquid crystal display (LCD), a camera, a temperature sensor, and a PIR sensor connected to our Raspberry Pi. We'll keep adding various sensors and actuators to it over time, so you're welcome to experiment and go well beyond the examples we provide here. You'll soon realize that the various techniques and patterns described in this book will allow you to quickly extend and customize the examples we provide to any device, sensor, or object you can think of.

2.1.1 The suspect: Raspberry Pi

We'll introduce the Raspberry Pi in greater detail in chapter 4, so all you need to understand for now is that a Pi is a small computer to which you can connect multiple sensors and accessories. It offers all the features you would expect from a desktop computer but with a lower power consumption and smaller form factor. Moreover, you can attach all sorts of digital and analog sensors or actuators to it using the input/output (I/O) pins. *Actuator* is an umbrella term for any element attached to a device that has an effect on the real world, for example, turning on/off some LEDs, displaying a text on an LCD panel, rotating an electric motor, unlocking a door, playing some music, and so on. In the Web of Things, just as you send write requests to a web API using HTTP, you do the same to activate an actuator. Now back to our exercises. The first thing you need to do is to download the examples used in these pages from our repository here: http://book.webofthings.io.

You can check out the repository on your own computer, and you'll find in it a few folders—one for each chapter. The exercises in this chapter are located in the folder chapter2-hello-wot/client. If you wonder about the code for the server, worry not! This is what you'll learn how to build in the rest of the book.

How to get the code examples in this chapter

We use the GitHub[a] service as a syncing server between our Pi and your computer. As an alternative, the Bitbucket[b] service works and is configured in a similar manner. Both services are based on the Git source version control system, and the source code for all the chapters is available from GitHub (here's the link: http://book.webofthings.io). The examples for this chapter are located in the chapter2-hello-wot folder.

If you're unfamiliar with Git and its commands, don't worry: there are plenty of short descriptions on the web, but here are the most vital commands to work with it:

- `git clone`—Fetches a version of a repository locally. For the book code you need to use the `recursive` option that will clone all the sub-projects as well: `git clone https://github.com/webofthings/wot-book --recursive`.
- `git commit -a -m "your message"`—Commits code changes locally.
- `git push origin master`—Pushes the last commits to the remote repository (origin) on the master branch. can

a GitHub is a widely popular, web-based, source code management system. Many open source projects are hosted on GitHub, because, well, it's pretty awesome. Here's an excellent intro to GitHub: http://bit.ly/intro-git.

b https://bitbucket.com

2.2 Exercise 1—Browse a device on the Web of Things

We'll start our exploration of the Web of Things with a simple exercise where you have almost nothing to do but click a few links. The first point we want to illustrate is that on the Web of Things, devices can easily offer simultaneously a visual user interface (web pages) to allow humans to control and interact with them and an application programming interface (API) to allow machines or applications to do the same.

2.2.1 Part 1—The web as user interface

In this first exercise, you'll simply use your browser to interact with some of the real Web of Things devices connected in our office. First, have a glimpse of what the setup in our office looks like through a webcam; see figure 2.3. Open the following link in your favorite browser to access the latest image taken by the web cam: http://devices.webofthings.io/camera/sensors/picture. This link will always return the latest screenshot taken by our camera so you can see the devices you will play with (try it at night—at night it's even more fun!). You won't be seeing the camera itself though.

You probably noticed that the URL you typed had a certain path structure. Let's play a bit with this structure and go back to the root of this URL, where you'll see the homepage of the gateway that allows you to browse through the devices in our office (figure 2.4). Simply enter the following URL in your browser: http://devices.webofthings.io.

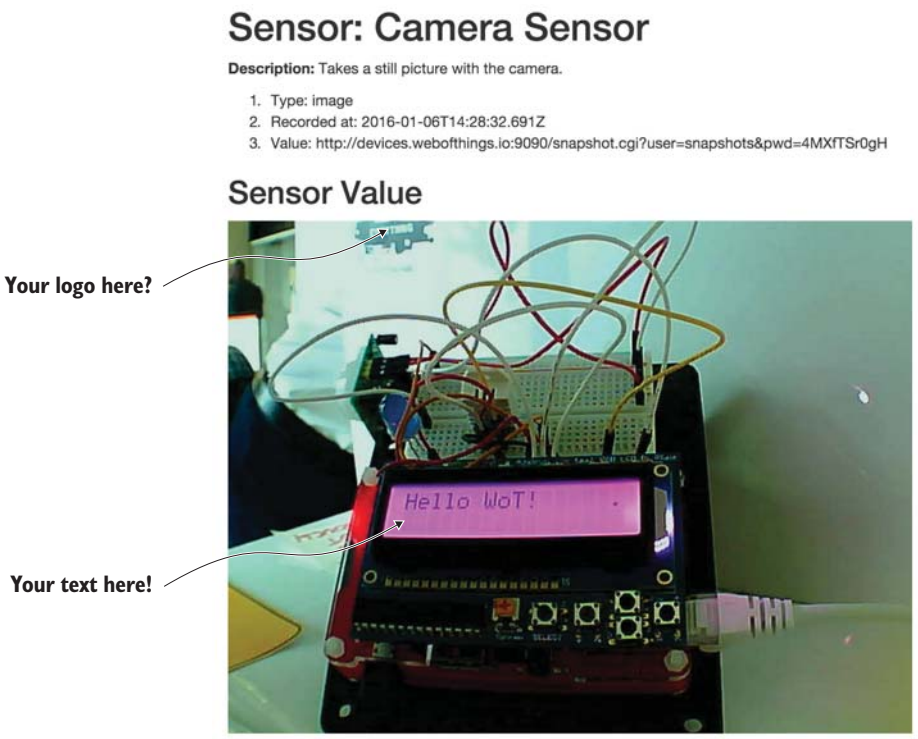


Figure 2.3 The web page of the camera used in our setup. The image is a live screenshot taken by the camera.

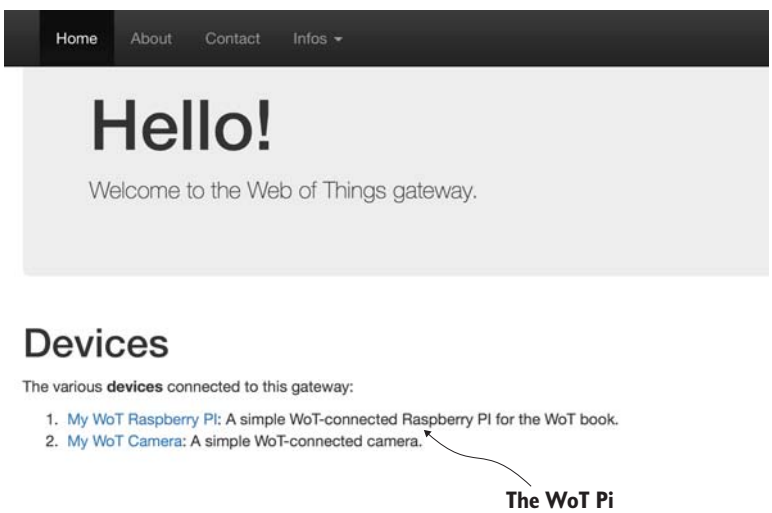


Figure 2.4 The HTML homepage of the gateway of our WoT device. The two hyperlinks at the bottom of the page allow you to access the pages of the devices connected to the gateway.

This URL will always redirect you to the *root page* of the gateway running in our office, which shows the list of devices attached to it. Here, you can see that two devices are attached to the gateway:

- A Raspberry Pi with various sensors and actuators
- A webcam (the one you accessed earlier)

Note that this page is automatically generated based on which physical devices we have attached to it, so you might see a few more devices or sensors as we attach them. Yes, although it looks just like any other web page, it's actually *real* data served in *real time* from *real* devices that are in a *real* office!

Now, click the My WoT Raspberry Pi link to access the root page of the device itself. Because you followed a link in your browser, you'll see that the URL has changed to http://devices.webofthings.io/pi, as shown in figure 2.5.

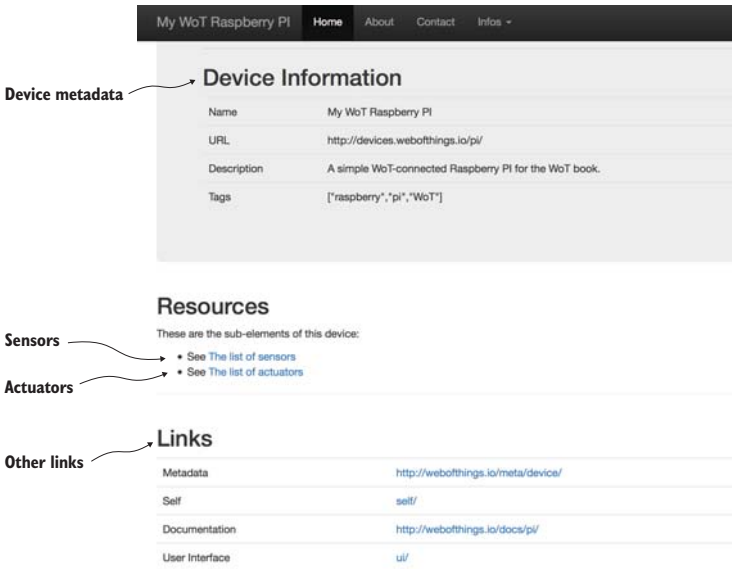


Figure 2.5 The homepage of the Raspberry Pi. Here as well, you can use the links at the bottom to browse and explore the various resources offered by this device, for example, its sensors and actuators.

This is another root page—the one of the device this time. In this case, we just appended /pi to the root URL of the gateway because this page is hosted on the gateway. But it would have been equally simple to serve the root page directly from the Pi and allow you to access the Pi directly (say, using its public IP address). In this case, it would have been both impractical (because we want to make sure we scale and support many concurrent users) and insecure (after all, the device is connected to the LAN of our company office). Using a gateway (a simple software application) outside our private network solves both those problems without changing the experience from your point of view, because you're still sending HTTP requests to a URL. Being

able to do this sort of thing is exactly the point of the Web of Things: leverage the tools and abstractions that work on the web and use them for physical objects!

Coming back to our device root page, hover with your mouse above the various links to see their structure, and then click The List of Sensors link. You'll see the URL change again to this (figure 2.6): http://devices.webofthings.io/pi/sensors.

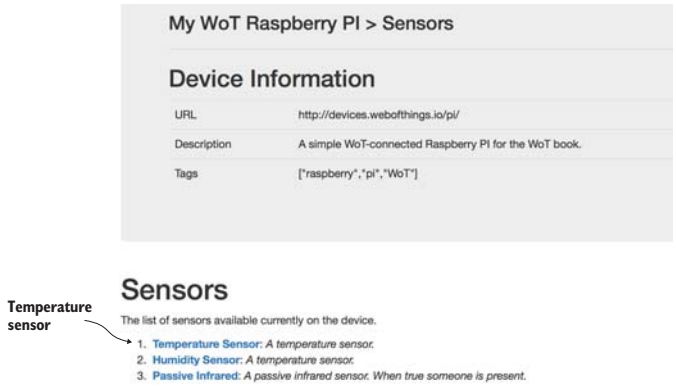


Figure 2.6 The list of sensors on the Pi. You can click each of them and see the latest known value for each.

So far, it's pretty straightforward: your browser is asking for an HTML page that shows the list of /sensors of the device /pi connected to the devices.webofthings.io gateway. Remember that there's also a camera connected to this, so in your browser address bar replace *pi* with *camera* and you'll be taken directly to the Sensors page of the camera: http://devices.webofthings.io/camera/sensors; see figure 2.7.

Now, go back to the list of sensors on your Pi and see the various sensors attached to the device. Currently, you can access three sensors: temperature, humidity, and passive infrared. Open the Temperature Sensor link and you'll see the temperature sensor page with the current value of the sensor. Finally, just like you did for the sensor, go to the actuators list of the Pi and open the Actuator Details page (screenshot in figure 2.13), at the following URL: http://devices.webofthings.io/pi/actuators/display.

The display is a simple LCD screen attached to the Pi that can display some text, which you'll use in exercise 2.4. You can see the information about this actuator, in particular the current value being displayed, the API description to send data to it, along with a form to display new data. You won't use this form for now, but this is coming in section 2.4.

2.2.2 Part 2—The web as an API

In part 1, you started to interact with the Web of Things from your browser. You've seen how a human user can explore the various content offered by a device (sensors, actuators, and so on) and how to control it from a web page. All of that is done by browsing the resources of a physical device, just as you'd browse the various pages of a

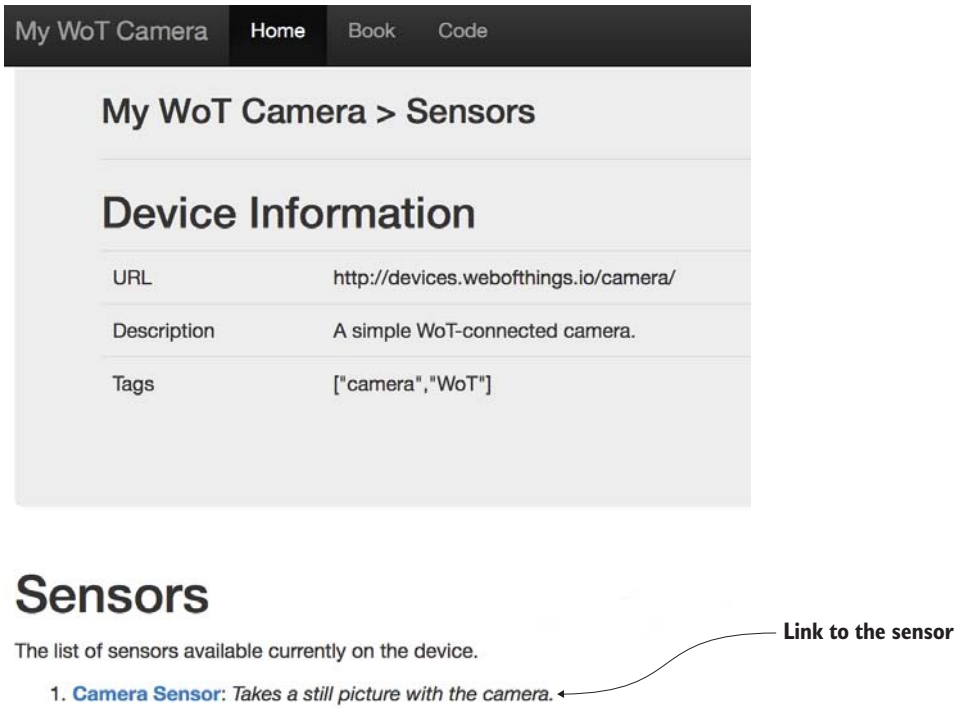


Figure 2.7 The sensors on the camera. There's only one sensor here, which is the current image.

website. But what if instead of a human user, you want a software application or another device to do the same thing, without having a human in the loop? How can you make it easy for any web client to find a device, understand what it does, see what its API looks like, determine what commands it can send, and so on?

Later in the book, we'll show you in detail how to do this. For now, we'll simply illustrate how the web makes it easy to support both humans and applications *by showing you what a client application sees* when it browses your device.

For this exercise, you'll need to have Chrome installed and install one of our favorite browser extensions called Postman[1] or use cURL[2] if you'd rather use the command line. Postman is a handy little app that will help you a lot when working with a web API, because it allows you to easily send HTTP requests and customize the various options of these requests, such as the headers, the payload, and much more. Postman will make your life easier throughout this book, so just go ahead and install it.

[1] Get it here: http://www.getpostman.com/

[2] cURL is a command-line tool that allows you to transfer data using various protocols, among which is HTTP. If it's not preinstalled on your machine, you can easily install it on Mac, Linux, or Windows. Website: http://curl.haxx.se/

In part 1, your browser is simply a web client requesting content from the server. The browser automatically asks for the content to be in HTML format, which is returned by the server and then displayed by the browser.

In part 2, you'll do almost the same exercise as in part 1 but this time by requesting the server to return JSON documents instead of an HTML page. JSON is pretty much the most successful data interchange format used on the internet. It has an easy-to-understand syntax and is lightweight, which makes it much more efficient to transmit when compared to its old parent, XML. In addition, JSON is easy for humans to read and write and also for machines to parse and generate, which makes it particularly suited to be *the* data exchange format of the Web of Things. The process of asking for a specific encoding is called *content negotiation* in the HTTP 1.1 specification and will be covered in detail in chapter 6.

STEP 1—GETTING THE LIST OF DEVICES FROM THE GATEWAY

Just as you did before, you'll send a GET request to the root page of the gateway to get the list of devices. For this just enter the URL of the gateway in Postman and click Send, as shown in figure 2.8.

Because most web servers return HTML by default, you'll see in the body area the HTML page content returned by the server (4). This is basically what happens behind the scenes each time you access a website from your browser. Now to get JSON instead

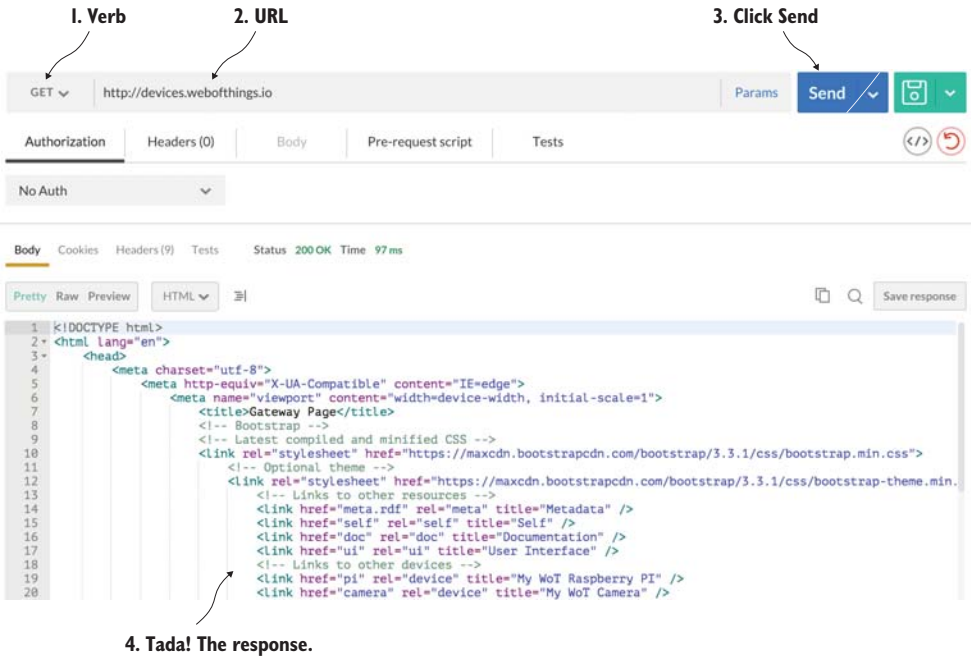


Figure 2.8 Getting the root page of the gateway using the Postman web client. The request is an HTTP GET (1) on the URL of the gateway (2). The response body will contain an HTML document (4).

of HTML, click the Headers button and add a header named `Accept` with `application/json` in the value, and click Send again, as shown in figure 2.9. Adding this header to your request is simply telling the HTTP server, "Hey, if you can, please return me the results encoded in JSON." Because this is supported by the gateway, you'll now see the same content in JSON, which is the machine equivalent of that page with only the content and no visual elements (that is, the HTML code).

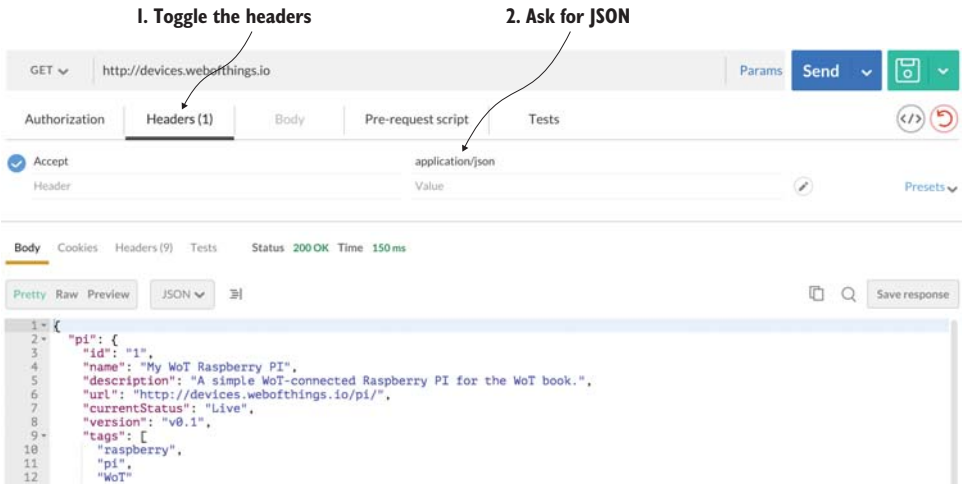


Figure 2.9 Getting the list of devices connected to the gateway via Postman. The `Accept` header is now set to `application/json` to ask for the results to be returned in JSON.

The JSON body returned contains a machine-readable description of the devices attached to the gateway and looks like this:

```
{
  "id": "1",
  "name": "My WoT Raspberry Pi",
  "description": "A simple WoT-connected Raspberry Pi for the WoT book.",
  "url": "http://devices.webofthings.io/pi/",
  "currentStatus": "Live",
  "version": "v0.1",
  "tags": [
    "raspberry",
    "pi",
    "WoT"
  ],
  "resources": {
    "sensors": {
      "url": "sensors/",
      "name": "The list of sensors"
    },
    "actuators": {
      "url": "actuators/",
      "name": "The list of actuators"
```

```
    }
  },
  "links": {
    "meta": {
      "rel": "http://book.webofthings.io",
      "title": "Metadata"
    },
    "doc": {
      "rel":
     "https://www.raspberrypi.org/products/raspberry-pi-2-model-b/",
      "title": "Documentation"
    },
    "ui": {
      "rel": ".",
      "title": "User Interface"
    }
  }
}
```

In this JSON document, you can see there are two first-level elements (`pi` and `camera`) that represent the two devices attached to the gateway and a few details about them, such as their URL, name, ID, or description. Don't worry for now if you don't understand everything here; all of this will become crystal clear to you in a few chapters.

STEP 2—GETTING A SINGLE DEVICE

Now change the URL of the request in Postman to point to the Pi device (which is exactly the same as the one you typed in your browser in part 1), and click Send again, as shown in figure 2.10.

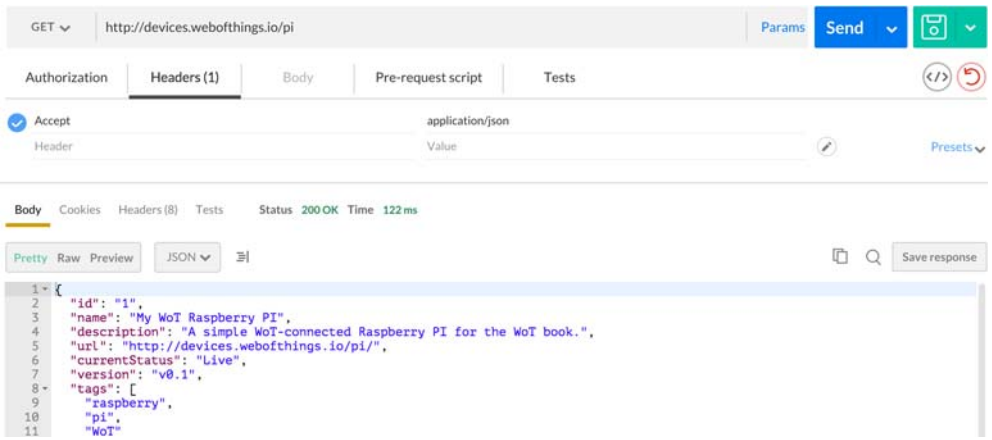


Figure 2.10 Getting the JSON representation of the Raspberry Pi. The JSON payload contains metadata about the device as well as links to its sub-resources.

The body now contains the JSON object of the Pi except with the same information as shown previously, and you can see that the `resources` object has `sensors`, `actuators`, and so on:

```
"resources": {
  "sensors": {
    "url": "sensors/",
    "name": "The list of sensors"
  },
  "actuators": {
    "url": "actuators/",
    "name": "The list of actuators"
  }
}
```

STEP 3—GETTING THE LIST OF SENSORS ON THE DEVICE

To get to the list of sensors available on the device, just as you did before, simply append `/sensors` to the URL of the Pi in Postman and send the request again. An HTTP GET there will return this JSON document in the response:

```
{
  "temperature": {
    "name": "Temperature Sensor",
    "description": "A temperature sensor.",
    "type": "float",
    "unit": "celsius",
    "value": 23.4,
    "timestamp": "2015-10-04T14:39:17.240Z",
    "frequency": 5000
  },
  "humidity": {
    "name": "Humidity Sensor",
    "description": "A temperature sensor.",
    "type": "float",
    "unit": "percent",
    "value": 38.9,
    "timestamp": "2015-10-04T14:39:17.240Z",
    "frequency": 5000
  },
  "pir": {
    "name": "Passive Infrared",
    "description": "A passive infrared sensor. True when someone present.",
    "type": "boolean",
    "value": true,
    "timestamp": "2015-10-04T14:39:17.240Z",
    "gpio": 20
  }
}
```

You can see that the Pi has three sensors attached to it (respectively, `temperature`, `humidity`, and `pir`), along with details about each sensor and its latest value.

STEP 4—GET DETAILS OF A SINGLE SENSOR

Finally, you'll get the details of a specific sensor, so simply append `/temperature` to the URL in Postman and click Send again. The URL should now be http://devices.webofthings.io/pi/sensors/temperature, as shown in figure 2.11.

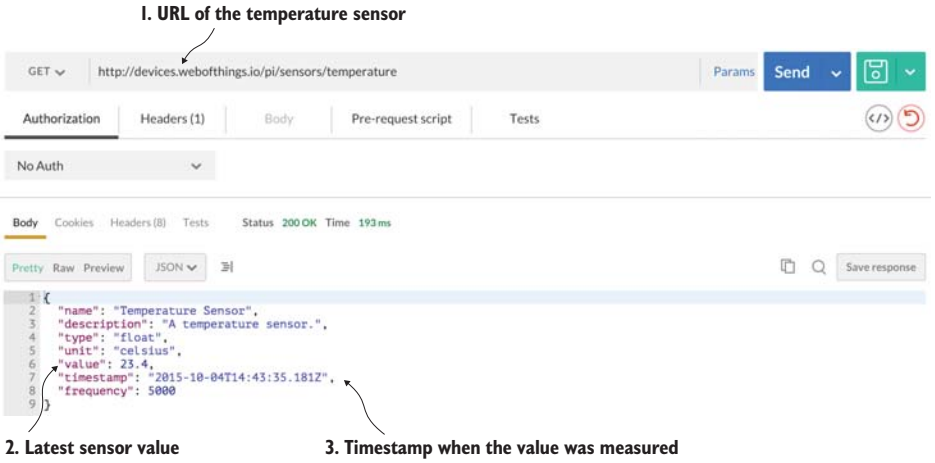


Figure 2.11 Retrieve the temperature sensor object from the Raspberry Pi. You can see the latest reading (23.4 degrees Celsius) and when it took place (at 14:43 on October 4, 2015).

You'll get more detailed information about that sensor, in particular the field `value`, which contains the latest value of the temperature sensor:

```
{
    "value":22.4
}
```

You'll now see additional details about this particular sensor, and among others you can see the latest value of the temperature sensor. If you only want this sensor value, you can append `/value` to the URL of the temperature sensor to retrieve it. This also works for the other sensors and actuators.

2.2.3 So what?

Now it's time for you to play around with the different URLs you've seen so far in this exercise. Look at how they differ and are structured, browse around the device, and try to understand what data each sensor has, its format, and so on. As an extension look at the electronic devices around you—the appliances in your kitchen or the TV or sound system in your living room, the ordering system in the café, or the train notification system, depending on where you're reading this book from. Now imagine how the services and data offered by all these devices could all have a similar structure: URLs, content, paths, and so on. Try to map this system using the same JSON structure you've just seen, and write the URLs and JSON object that would be returned.

What you have seen is that both humans and applications get data using exactly the same URL but using another encoding format (HTML for humans, JSON for applications). Obviously, the data in both cases is identical, which makes it easy for application developers to go back and forth from one format to the other. A lot of what you've seen in this first part is linked to using HTTP and URLs as technologies to offer web services. You'll explore and learn a lot more about how this can be used on devices in chapter 6 onward.

2.3 Exercise 2—Polling data from a WoT sensor

In the first exercise you learned about the structure of a WoT device and how it works. In particular, you saw that every element of the device is simply a resource with a unique URL that can be used by both people and applications to read and write data. Now you're going to put a developer hat on and start coding your first web application that interacts with this Web of Things device.

2.3.1 Part 1—Polling the current sensor value

For this exercise, go to the folder you checked out from GitHub into the chapter2-hello-wot/client folder. Double-click the ex-2.1-polling-temp.html file to open it in a modern browser.[1] This page simply displays the value of the temperature sensor on the Pi in our office and updates this value every five seconds by retrieving it in JSON, exactly as you saw in figure 2.11.

This file uses jQuery[2] to poll data from the temperature sensor on our Pi. Now open this file in your favorite code editor and look at the source code. You'll see two things there:

- An `<h2>` tag showing where the current sensor value will be written.
- A JavaScript function called `doPoll()` that reads the value from the Pi, displays it, and calls itself again five seconds later. This function is shown in the following listing.

Listing 2.1 Polling for the temperature sensor

Wait until the page is loaded and then call doPoll()

Use the AJAX helper to get the JSON payload from the temperature sensor

```
$(document).ready(
  function doPoll() {
    $.getJSON('http://devices.webofthings.io/pi/sensors/temperature',
      function (data) {
        console.log(data);
```

When the response arrives, this function is called

[1] We fully tested our examples on Firefox (>41) and Chrome (>46) and suggest you install the latest version of these. Safari (>9) should also work. If you really want to use Internet Explorer, please be aware that you'll need version 10 onward; older versions won't work.

[2] jQuery is a handy JavaScript library that makes it easier to do lots of things, such as talk to REST APIs, manipulate HTML elements, handle events, and so on. Learn more here: http://jquery.com/.

```
        $('#temp').html(data.value + ' ' + data.unit);
        setTimeout(doPoll, 5000);
    });
});
```

The doPoll() function sets a timer to call itself again in 5 seconds (5000 milliseconds)

Select the "temp" HTML element and update its content using the data.value (the value) and data.unit (the unit) returned in the JSON payload

When developing (and especially debugging!) web applications, it might be useful to display content from JavaScript outside the page; for this you have a JavaScript console. To access it in Chrome, right-click somewhere on the page and select Inspect Element; then see the console view that displays below. The `console.log(data)` statement displays the `data` JSON object received from the server in this console.

2.3.2 Part 2—Polling and graphing sensor values

This is great, but in some cases you'd like to display more than just the current value of the sensor, for example, a graph of all readings in the last hour or week. So open the second HTML file in the exercises (ex-2.2-polling-temp-chart.html). This is a slightly more complex example that keeps track of the last 10 values of the temperature sensor and displays them in a graph. When you open this second file in your browser, you'll see the graph being updated every two seconds, as shown in figure 2.12.

We built this graph using Google Charts,[1] a nice and lightweight JavaScript library for displaying all sorts of charts and graphs. See our annotated code sample in the next listing.

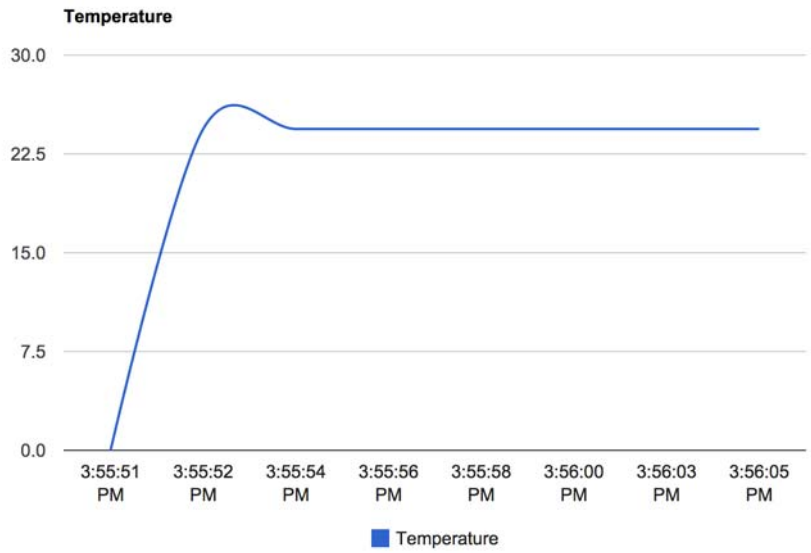


Figure 2.12 This graph gets a new value every few seconds from the device and is updated automatically.

[1] https://developers.google.com/chart/

Listing 2.2 Polling and displaying a sensor reading

```
$(document).ready(function () {
  var maxDataPoints = 10;
  var chart = new google.visualization.LineChart($('#chart')[0]);
  var data = google.visualization.arrayToDataTable([
    ['Time', 'Temperature'],
    [getTime(), 0]
  ]);

  var options = {
    title: 'Temperature',
    curveType: 'function',
    animation: {
      duration: 1000,
      easing: 'in'
    },
    legend: {position: 'bottom'}
  };

  function addDataPoint(dataPoint) {
    if (data.getNumberOfRows() > maxDataPoints) {
      data.removeRow(0);
    }
    data.addRow([getTime(), dataPoint.value]);
    chart.draw(data, options);
  }

  function getTime() {
    var d = new Date();
    return d.toLocaleTimeString();
  }

  function doPoll() {
    $.getJSON('http://devices.webofthings.io/pi/sensors/temperature/value',
      function (result) {
        addDataPoint(result);
        setTimeout(doPoll, 2000);
      });
    }

    doPoll();
});
```

Initialize the Google chart

Create an array that will contain the data points

Configure the parameters of the chart

Add a data point to the chart data and remove the oldest one if needed (if there are already 10 points available)

Redraw the chart with the new data

Poll the temperature sensor just like before

When the new readings are returned, use them to call the addDataPoint() function

2.3.3 Part 3—Real-time data updates

In the previous exercises, polling the temperature sensor of the Pi worked just fine. But this seems somewhat inefficient, doesn't it? Instead of having to fetch the temperature from the device every two seconds or so, wouldn't it be better if our script was *informed* of any change of temperature when it happens, and only if the value changes?

As we'll explore to a greater extent in chapter 6, this has been one of the major impedance mismatches between the model of the web and the event-driven model of wireless sensor applications. For now, we'll just look at one way of resolving the problem using a relatively recent add-on to the web: WebSockets. In a nutshell, WebSockets

are simple yet powerful mechanisms for web servers to push notifications to web clients introduced as part of the efforts around the HTML5 standards.

The WebSockets standard comprises two distinct parts: one for the server and one for the client. Since the server is already implemented for us, the only specification we'll use here is the client part. The client WebSockets API is based on JavaScript and is relatively simple and straightforward. The two lines of code in the following listing are all you need to connect to a WebSocket server and display in the console all messages received.

Listing 2.3 Connecting to a WebSocket and listening for messages

```
var socket = new WebSocket('ws://ws.webofthings.io');
socket.onmessage = function (event) {console.log(event);};
```

Let's get back to our examples. Go to the folder. Double-click the ex-2.3-websockets-temp-graph.html file to open it in your favorite browser. What you see on the page is exactly the same as in the previous exercise. But under the hood things are quite different. Indeed, have a look at the new code shown here.

Listing 2.4 Register to a WebSocket and get real-time temperature updates

```
var socket = new
     WebSocket('ws://devices.webofthings.io/pi/sensors/temperature');

socket.onmessage = function (event) {
  var result = JSON.parse(event.data);
  addDataPoint(result);
};

socket.onerror = function (error) {
  console.log('WebSocket error!');
  console.log(error);
};
```

Create a WebSocket subscription to the temperature sensor. Note that the URL uses the WebSockets protocol (ws://...).

Register this anonymous function to be called when a message arrives on the WebSocket.

Register this other anonymous function to be triggered when an error occurs on the WebSocket.

In this exercise, you don't poll periodically for new data but only register your interest in these updates, by subscribing to the `/sensors/temperature` endpoint via WebSockets. When the server has new temperature data available, it will send it to your client (your web browser). This event will be picked up by the anonymous function you registered and give it as a parameter the event object that contains the latest temperature value.

2.3.4 So what?

Let's take a step back and reflect about what you did with this exercise: you managed to communicate with an embedded device (the Raspberry Pi) that might be on the other side of the world (if you don't happen to be living in rainy and beautiful England). From a web page you managed to fetch, on a regular basis, data from a sensor

connected to the device and display it on a graph. Not bad for a simple web page of 60 lines of HTML, JavaScript, and CSS code. You didn't stop there: with fewer than 10 lines of JavaScript you also subscribed to notifications from our Pi using WebSockets and then displayed the temperature in our office in real time. As an extension of this exercise, you could write a simple page that automatically fetches the image from the camera (ideally, you'd avoid doing this 25 times per second!).

If this was your first encounter with the Web of Things, what should strike you at this stage is the simplicity of these examples. Let's imagine for a second our Pi wasn't actually providing its data through HTTP, JSON, or WebSockets but via a "vintage" XML-based machine-to-machine application stack such as DPWS (if you've never heard about it, don't worry; that's exactly our point!). Basically, you wouldn't be able to talk directly to the device from your browser without a lot more effort. You would have been forced to write your application using a lower-level and more complex language such as C or Java. You wouldn't have been able to use widespread concepts and languages such as URLs, HTML, CSS, and JavaScript. This is also what the Web of Things is about: creating APIs for things that are universally accessible and bringing them closer to the masses of web development where a lot of today's innovation and creative building happens.

As mentioned before, in this book you'll learn a lot more about the art of API crafting for physical things. In chapter 6 we'll look at HTTP, REST, and JSON as well as at the real-time web, but in chapter 7 we'll also look at how to build bridges to bring other protocols and systems closer to goodness of the web.

2.4 Exercise 3—Act on the real world

So far, you've seen various ways to read all sorts of sensor data from web devices. What about "writing" to a device? For example, you'd like to send a command to your device to change a configuration parameter. In other cases, you might want to control an actuator (for example, open the garage door or turn off all lights).

2.4.1 Part 1—Use a form to update text to display

To illustrate how you can send commands to an actuator, this exercise will show you how to build a simple page that allows you to push a piece of text to the LCD connected to the Pi in our office. To test this functionality first, open the actuator page of the LCD: http://devices.webofthings.io/pi/actuators/display.

On this page (shown in figure 2.13), you now see the various *properties* of the LED actuator. First, you see `brightness`, which you could change (but can't, because we made it read-only). Then, you have `content`, which is the value you want to send, and finally there is the `duration`, which is how long the piece of text will be displayed on our LCD. Use Postman to get the JSON format of the display actuator by entering the URL just shown as you learned in the first exercise of this chapter:

```
{
  "name": "LCD Display screen",
  "description": "A simple display that can write commands.",
```

```
  "properties": {
    "brightness": {
      "name": "Brightness",
      "timestamp": "2015-02-01T21:06:02.913Z",
      "value": 80,
      "unit": "%",
      "type": "integer",
      "description": "Percentage of brightness of the display. Min is 0
       which is black, max is 100 which is white."
    },
    "content": {
      "name": "Content",
      "timestamp": "2015-02-01T21:06:32.933Z",
      "type": "string",
      "description": "The text to display on the LCD screen."
    },
    "duration": {
      "name": "Display Duration",
      "timestamp": "2015-02-01T21:06:02.913Z",
      "value": 5000,
      "unit": "milliseconds",
      "type": "integer",
      "read-only": true,
      "description": "The duration for how long text will be displayed
       on the LCD screen."
    }
  },
  "commands": [
    "write",
    "clear",
    "blink",
    "color",
    "brightness"
  ]
}
```


Obviously, it wouldn't be much fun to display something in our office if you couldn't see what is being displayed. For this reason, we've set up a webcam where you can see the LCD on our Pi, so you can always see what is displayed on it. Here's the URL: http://devices.webofthings.io/camera/sensors/picture. So go ahead, open this page, and you'll see the latest picture of the camera you saw earlier in figure 2.3 (to see the latest image, just refresh the page).

Now you'll send a new message to the Pi for it to be displayed by the LCD. The `content` property is always the current message displayed on the LCD, so to update it you simply POST a new value for that property with the message to be displayed (for example, `{"value": "Hello World!"}`) as a body. You can go ahead and try this in Postman, but the simplest way to do it is through the page of the display actuator in your browser: http://devices.webofthings.io/pi/actuators/display. See figure 2.13 for the details of the LCD actuator.

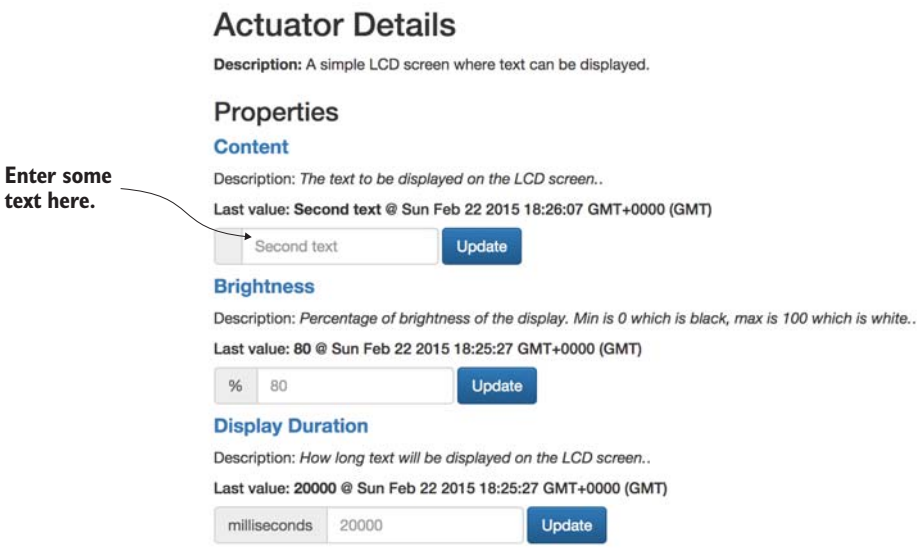


Figure 2.13 The details of the LCD actuator, with the various properties that you can set, for example, the text that should be displayed next on the device

On this page you can see the various properties of the LCD actuator. Some are editable, and some aren't. The `content` property is the one you want to edit, so enter the text you'd like to display and click Update. If all works fine, you'll see a JSON payload like this:

```
{
  "id":11,
"messageReceived":"Make WoT, not war!",
  "displayInSeconds":20
}
```

The returned payload contains the message that will be displayed, a unique ID for your message, and an estimated delay for when your text will appear on the LCD screen (in seconds), so you know when to look at the camera image to see your text.

2.4.2 Part 2—Create your own form to control devices

Now let's build a simple HTML page that allows you to send all sorts of commands to a web device using a simple form. From your browser, open the file ex-3.1-actuator-form.html in the exercises folder and you'll see the screen shown in figure 2.14.

Figure 2.14 This simple client-side form allows you to send new text to be displayed by the Pi.

This page has an input text field and a Send to Pi button, as shown in the following listing. Whatever text you enter will be displayed there. So yes, please keep it courteous, and because the API of our Pi is open to the public, we decline all responsibilities for what people write there.

Listing 2.5 Simple HTML form to send a command to an actuator

```
<form action="http://devices.webofthings.io/pi/actuators/display/content/"
 method="post">
  <label>Enter a message:</label>
  <input type="text" name="value" placeholder="Hello world!">
  <button type="submit">Send to Pi</button>
</form>
```

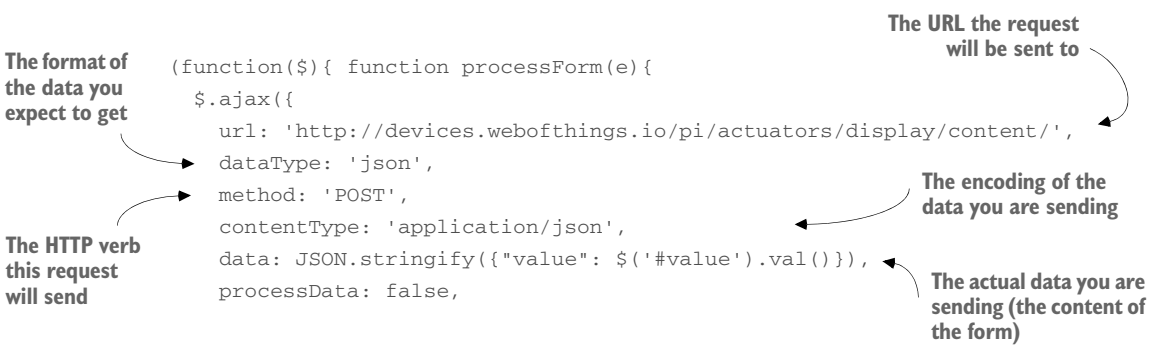
This is a simple HTML form that sends an HTTP POST (value of `method`) to the URL of the display (the value of `action`). The input text bar is called *value* (`name="value"`) so that the Pi knows where to find the text to be displayed. This method works well. Unfortunately, what you don't see behind the scenes is that web browsers do not submit (nor do they make it possible to submit) the server using a JSON payload body (as you could easily do with Postman for the previous) but instead use a format called `application/x-www-form-urlencoded`. The Pi needs to be able to understand this format in addition to `application/json` in order to handle data input from HTML forms.

HTML forms can use only the verbs POST or GET but not DELETE or PUT. It's rather unfortunate that even modern browsers don't send the content of HTML forms as JSON objects because of some obscure legacy reasons, but hey, *c'est la vie!*

As you'll see later in this book, the ability for all entities on the Web of Things to receive and transmit JSON content is essential to guarantee a truly open ecosystem. For this reason, we'll show you how to send actual JSON from an HTML form page (by using AJAX and JavaScript), because doing so is an essential part of communicating with web devices.

Open the ex-3.2-actuator-ajax-json.html file to see a similar form but this time with a large piece of JavaScript, as follows.

Listing 2.6 Send an HTTP POST with JSON payload from a form




```
(function($){ function processForm(e){
  $.ajax({
    url: 'http://devices.webofthings.io/pi/actuators/display/content/',
    dataType: 'json',
    method: 'POST',
    contentType: 'application/json',
    data: JSON.stringify({"value": $('#value').val()}),
    processData: false,
```

```
    success: function( data, textStatus, jQxhr ){
      $('#response pre').html( JSON.stringify( data ) );
    },

    error: function( jqXhr, textStatus, errorThrown ){
      console.log( errorThrown );
    }
  });
  e.preventDefault();
}
$('#message-form').submit(processForm);
})(jQuery);
```

The callback to invoke if the request was successful

The callback to invoke if the request failed

Attach the processForm() function to be called when someone clicks Submit

In this code, a function called `processForm()` is defined, which takes the data from the form, packs it into a JSON object, POSTs it to the Pi, and displays the result if successful (or displays an error in the console otherwise). The `url` parameter specifies the end-point URL (the Pi display), the `method` is the HTTP method to use, and the `contentType` is the format of the content sent to the server (in this case `application/json`). The last line attaches the event generated by a click of the Submit button of the form `#message-form` to call the `processForm()` function.

There is a variation of this code, ex-3.2b-actuator-ajax-form.html, which encodes the data in the `application/x-www-form-urlencoded` format in place of JSON, just as it's done with the simple form we showed in part 1 of exercise 3.

2.4.3 So what?

In this section you learned the basics of how to send write requests and commands to a device, both using a form on a web page and from an API. You had a crash course in the limitations, challenges, and problems of the modern web (don't worry; there are many more ahead!), in particular how different web browsers can interpret and implement the same web standards differently. Finally, you learned how to use AJAX to bypass these limitations and send JSON commands to a Raspberry Pi and control it remotely.

We hope that after doing this exercise you realize that it's straightforward to send actuator commands to all sorts of devices—as long as these are connected to the web and offer a simple HTTP/JSON interface. But the last problem is how to find a device nearby, understand its API, determine what functions are offered by the device, and know what parameters you need to include in your command, along with their type, unit, limitations, and the like. The next section will show you how to solve this problem, so keep reading.

2.5 Exercise 4—Tell the world about your device

In the previous exercises you learned how devices can be easily exposed over the web and then explored and used by other client applications. But those examples assumed

that you (as a human developer or as the application you wrote) *know* what the fields of the JSON objects (for example, sensor or actuator) mean and how to use them. But how is this possible? What if the only thing you know about a device is its URL and nothing else?

Imagine you'd like to build a web application that can control home automation devices present in your local network. How can you ensure this application will always work, even if you're in someone else's network and you don't know anything about devices there?

First, you need to find the devices at a network level (the *device discovery* problem). In other words, how can your web application discover the root URL of all the devices around you?

Second, even if you happened to know (by some magic trick) the root URL of all Web of Things–compatible devices around you, how could your application "understand" what sensors or actuators these devices offer, what formats they use, and the meaning of those devices, properties, fields, and so on?

As you saw in exercise 2 (section 2.3.2), if you know the root URL of a device, then you can easily browse the device and find data about it and its sensors, services, and more. This is easy because you're a human, but imagine if you just had a JSON document with unintelligible words or characters and no documentation that explain what those words mean—how would you know what the device does? And how would you know it's a device, for that matter?

Open ex-4-parse-device.html in your browser, and you'll see a form prepopulated with the URL of the Pi (figure 2.15), so simply click Browse This Device.

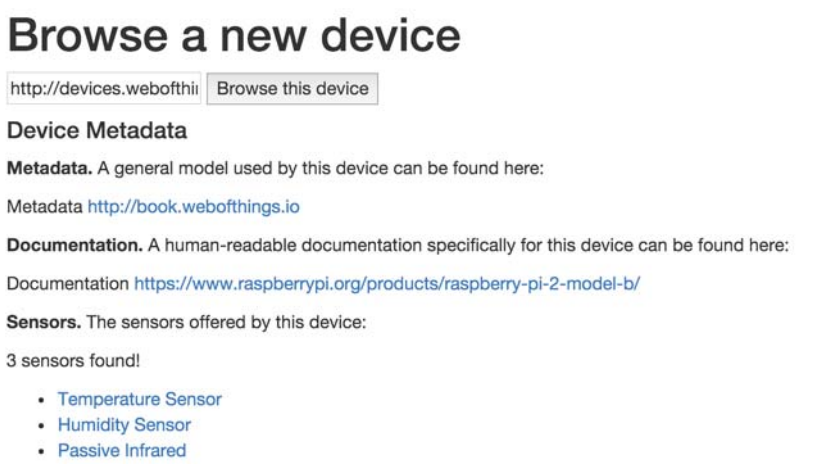


Figure 2.15 A mini-browser that parses your device metadata and displays the results

This JavaScript code of ex-4-parse-device.html will read the root document of the Raspberry Pi (as JSON) and generate a simple report about the device and its sensors, along with link to the documentation for this device. First, let's look at the HTML code to display the report.

Listing 2.7 A basic browser

```
<form id="message-form">
    <input type="text" id="host" name="host"
     value="http://devices.webofthings.io/pi"
placeholder="The URL of a WoT device" />
    <button type="submit">Browse this device</button>
</form>

<h4>Device Metadata</h4>
<p><b>Metadata.</b> A general model used by this device can be found here:
<div id="meta"></div></p>
<p><b>Documentation.</b> A human-readable documentation specifically for
this device can be found here: <div id="doc"></div></p>
<p><b>Sensors.</b> The sensors offered by this device:
     <div id="sensors"></div></p>
<ul id="sensors-list">
</ul>
```

The first thing you can see is a form where you can enter the root URL of a device with a Browse button. Then, there are some HTML text elements that will act as placeholders (`meta`, `doc`, and so on). Now let's look at the AJAX calls.

Listing 2.8 Retrieve and parse device metadata using AJAX JSON calls

```
(function ($) {
  function processForm(e) {

    var sensorsPath = '';

    $.ajax({
      url: $('#host').val(),
      method: 'GET',
      dataType: 'json',
      success: function (data) {
        $('#meta').html(data.links.meta.title + " <a href=\"" +
        data.links.meta.rel + "\">" + data.links.meta.rel + "</a>");
        $('#doc').html(data.links.doc.title + " <a href=\"" +
        data.links.doc.rel + "\">" + data.links.doc.rel + "</a>");

        sensorsPath = data.url + data.resources.sensors.url;

        $.ajax({
          url: sensorsPath,
          method: 'GET',
          dataType: 'json',
          success: function (data) {
            var sensorList = "";

            $('#sensors').html(Object.keys(data).length + " sensors
            found!");
```

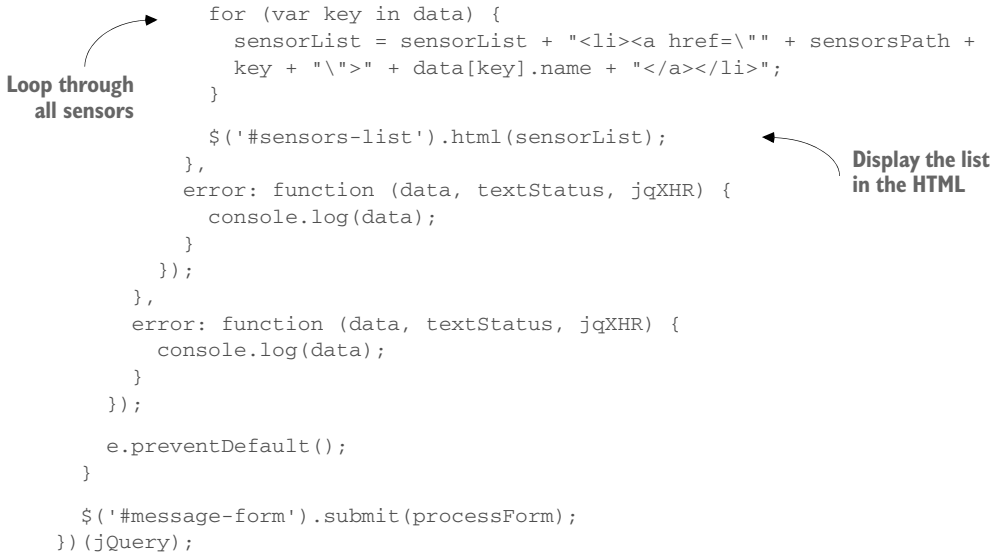
GET the ROOT JSON of the device and extract data from it

Update the "meta" and "doc" elements with the link found in the root JSON document

Store the URL of the sensors resource

GET the list of all sensors on the device

Callback function that processes the sensors JSON document; 'data' contains the JSON object of the sensors




```
            for (var key in data) {
              sensorList = sensorList + "<li><a href=\"" + sensorsPath +
              key + "\">" + data[key].name + "</a></li>";
            }

            $('#sensors-list').html(sensorList);
          },
          error: function (data, textStatus, jqXHR) {
            console.log(data);
          }
        });
      },
      error: function (data, textStatus, jqXHR) {
        console.log(data);
      }
    });

    e.preventDefault();
  }

  $('#message-form').submit(processForm);
})(jQuery);
```

Looking at this code, you can see that you first set the root JSON document of the device using the URL entered in the form (`$('#host').val()`). If the JSON file has been successfully retrieved, the `success` callback function will be triggered with the `data` variable containing the root JSON document of the device (which was shown in step 2 of section 2.2.2). Then you parse this JSON to extract the elements you're looking for; in this case the code is looking for a `links` element in the returned JSON object (hence the `data.links`), which contains various links to get more information about this device, which looks like the following:

```
"links": {
    "meta": {
      "rel": "http://book.webofthings.io",
      "title": "Metadata"
    },
    "doc": {
      "rel":
"https://www.raspberrypi.org/products/raspberry-pi-2-model-b/",
      "title": "Documentation"
    },
    "ui": {
      "rel": ".",
      "title": "User Interface"
    }
}
```

In particular, the `meta` element contains a link (value of `rel`) to the general model used by this device (which describes the grammar used to describe the elements of this device) and then a `doc` that links to a human-readable documentation that

describes the meaning (the semantics) and specific details of this particular device (that is, which sensors are present and what they measure).

The metadata document linked in the previous code is nothing more than a machine-readable JSON document model that allows users to describe WoT devices in a structured manner, along with a definition of the logic elements all WoT devices must have. If hundreds of device manufacturers would use this same data model to expose the services of their devices, it would mean that any application that can read and parse this file will be able to read the JSON file returned by the device and understand the components of the devices (how many sensors it has, their names or limitations, their type, and so on).

Now, what about the sensors or actuators themselves? The `links` element only defined metadata (documentation and such) about the device, not the device contents itself. To find the sensors contained in the device, you'll have to parse the `sensors` field of the `resources` element, which is what happens in the second AJAX call where you do a GET on the sensors resource of the device. Once you get the sensors JSON document, you iterate over each sensor and create a link to it using this pattern:

```
<li><a href=\""+sensorsPath+key+"\">"+data[key].name+"</a></li>
```

Here `sensorsPath` is the URL of the sensors resource (in this case http://devices.webofthings.io/pi/sensors) to which you add the sensor ID of each sensor (`key`), along with the name of the respective sensor (`data[key].name`).

2.5.1 So what?

If you didn't understand all the details of the previous exercises, it's perfectly fine—there's nothing wrong with you! What happened is that you got your first hands-on crash course on the Semantic Web or rather on the hard problems it tries to solve. The reason you've heard a lot about it yet never seen or used it (or understood it, for that matter) is that it's a complex problem for computers and people who program them: how the hell do you explain the real world—and its existential questions—to a computer? Well, it turns out you can't really teach philosophy to your machine yet. But, as we've shown here and will detail in chapter 8, there are quite a few small tricks that you can apply successfully that make the web—and computers—just a little smarter.

You've seen how web devices can advertise their basic capabilities, data, and services in a machine-readable manner. The fact that we used well-known web patterns made it easy to build a web app interacting with our Things. Unfortunately, there's no single standard to define this information universally, and the JSON model we use is something born out of trial and error over the years. In order to reach more of the Web of Things potential, we need to have the ability to define this information in a universally accepted manner using a well-defined namespace with a clear semantic definition. We'll explore how to get there using web and lightweight Semantic Web technologies in much more detail in chapter 8.

2.6 Exercise 5—Create your first physical mashup

In the previous exercises, you learned how to access a web device, understand the service and data it offers, and read and write data from devices. In this exercise, we'll show you how to build your first mashup. The concept of mashups originates from the hip-hop scene to describe a song composed by taking samples of other songs. Similarly, a web mashup is a web application that gets data from various sources, processes it, and combines it to create a new application.

Here, you'll create not only a web mashup but a *physical mashup*—a web application that uses data from a real sensor connected to the web. Indeed, in this exercise you're going to take local temperature data from the Yahoo! Weather service, compare it with the temperature sensor attached to the Pi in our office, and publish your results to the LCD screen attached to the Pi in London. Finally, to get a visual feedback of what your message looks like, you'll use the web API of the webcam to take a picture and display it on our web page! See figure 2.16 for an illustration.

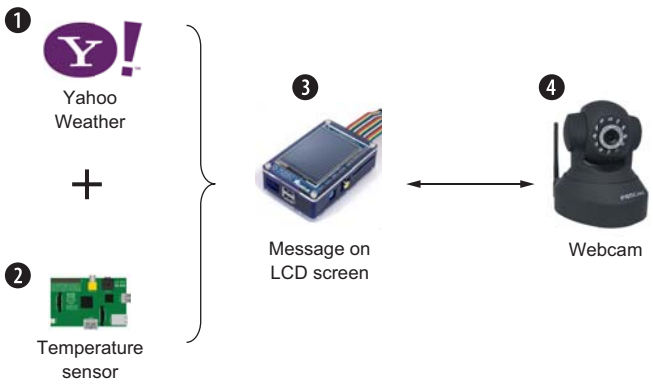


Figure 2.16 A physical mashup application. First (1), you retrieve the local temperature from Yahoo Weather and then the remote temperature from the sensor attached to our Pi (2). You compare it with the temperature in London and send the results to an LCD screen (3). When the screen displays the text you've sent, you retrieve a picture of the screen form the webcam (4) and display it on the mashup.

Go ahead and open the file ex-5-mashup.html in both your editor and your browser. This code is a little longer than what you've seen so far but not much more complicated, as shown in the following listing.

Listing 2.9 Listing 2.9 Mashup function

```
$(document).ready(function () {
  var rootUrl = 'http://devices.webofthings.io';

  function mashup(name, location) {
    var yahooUrl = "https://query.yahooapis.com/v1/public/yql?q=select item
    from weather.forecast where woeid in (select woeid from geo.places(1)
```

```
    where text='" + location + "') and u='c'&format=json";
    $.getJSON(yahooUrl, function (yahooResult) {
      var localTemp =
      yahooResult.query.results.channel.item.condition.temp;
      console.log('Local @ ' + location + ': ' + localTemp);
      $.getJSON(rootUrl + '/pi/sensors/temperature', function (piResult) {
        console.log('Pi @ London: ' + piResult.value);
        publishMessage(prepareMessage(name, location, localTemp,
        piResult.value));
      });
    });
  }

  function publishMessage(message) {
    $.ajax(rootUrl + '/pi/actuators/display/content', {
      data: JSON.stringify({"value": message}),
      contentType: 'application/json',
      type: 'POST',
      success: function (data) {
        $('#message').html('Published to LCD: ' + message);
        $('#wait').html('The Webcam image with your message will appear
        below in : ' + (data.displayInSeconds+2) + ' seconds.');
        console.log('We will take a picture in ' +
        (data.displayInSeconds+2) + ' seconds...');
        setTimeout(takePicture, (data.displayInSeconds+2) * 1000);
      }
    });
  }

  function prepareMessage(name, location, localTemp, piTemp) {
    return name + '@' + location + ((localTemp < piTemp) ? ' < ' : ' > ')
    + piTemp;
  }

  function takePicture() {
    $.ajax({
      type: 'GET',
      url: rootUrl + '/camera/sensors/picture/',
      dataType: 'json',
      success: function (data) {
        console.log(data);
        $('#camImg').attr('src', data.value);
      },
      error: function (err) {
        console.log(err);
      }
    });
  }

  mashup('Rachel', 'Zurich, CH');
});
```

Get the temperature in the user location from Yahoo

Get the temperature from the WoT Pi in London

Prepare the text to publish and use it to update the content of the LCD screen

POST the message to the LCD actuator

Set a timer that will call the takePicture() function in N seconds (after the LCD content has been updated)

Generate the text to display with the user name, location, and Pi temperature

Retrieve the current image from the webcam in our office

Update the HTML <img> tag with the image URL

The `mashup()` function is responsible for running the different bits of the mashup. It takes two parameters: the first parameter is your name; the second one is the name of the city where you live formatted as `city, country code` (for example, Zurich, CH; London, UK; or New York, US). It's then essentially composed of two HTTP GET calls over AJAX requesting a response as `application/json` representations. The first call is to the Yahoo Weather Service API, which given a location returns its current weather and temperature.

Once this call has returned (that is, the anonymous callback function has been invoked), the second function is called to fetch the latest value from the Pi temperature sensor, just as you already did in section Part 1—Polling the current sensor value.

Next, you call `prepareMessage()`, which formats your message and passes the result to `publishMessage()`. This last function runs an HTTP POST call over AJAX with a JSON payload containing the message to push to the LCD screen, as done in section Exercise 3—Act on the real world.

Because you need to wait in the queue for your message to be displayed, you set a timer that will trigger the `takePicture()` function. This last function runs a final HTTP GET request to fetch a picture of what the LCD screen shows, via the web-enabled camera. You then dynamically add the returned picture to the image container of your HTML page.

To start this chain of real-world and virtual-world events, all you need to do is edit the source code so it invokes the `mashup(x,y)` function using your own name and city. For example, Rachel from Zurich in Switzerland needs to call this function as follows:

```
mashup('Rachel', 'Zurich, CH')
```

Then open the file in your browser, and voilà! Within a few seconds, you'll see a live image from the webcam with your message appearing on the screen of the Pi in our office.

2.6.1 So what?

You've built your first web-based physical mashup using data from various sources, both physical and real-time, and run a simple algorithm to decide whether your weather is better than ours (although competing against London on the weather is somewhat unfair). Think about it for a second. This mashup involves a temperature sensor connected to an embedded device, a video camera, an LCD screen, and a virtual weather service, and yet you were able to create a whole new application that fits into 80 lines of HTML and JavaScript, UI included! Isn't that nice? All this thanks to the fact that all the actors (devices and other services) expose their APIs on the web and therefore are directly accessible using JavaScript! You'll learn much more about physical mashups throughout the book and especially in chapter 10, where we'll survey the various tools and techniques available.

2.7 Summary

- You experienced your first hands-on encounter with web-connected devices across the world and could browse their metadata, content, sensors, actuators, and so on.
- Web-connected devices can be surfed just like any other website. Real-time data from sensors can be consumed via an HTTP API just like other content on the web.
- It's much easier and faster to understand the basics of HTTP APIs than the various and complex protocols commonly used in the IoT.
- In only a few minutes you were able to read and write data to a device across the world by sending HTTP requests with Postman.
- Connecting the physical world to the web enables rapid prototyping of interactive applications that require only a few lines of HTML/JavaScript code.
- As data and services from various devices are made available as web resources, it becomes easy to build physical mashups that integrate content from all sorts of sources with minimal integration effort.

The Internet of Things (IoT) is a hot conversation topic. Analysts call it a disruptive technology. Competing standards and technologies are appearing daily, and there are no tangible signs of a single protocol that will enable all devices, services, and applications to talk to each other seamlessly. Fortunately, there's a great universal IoT application platform available now: the World Wide Web. Web standards and tools provide the ideal substrate for connected devices and applications to exchange data, and this vision is called the Web of Things.

Building the Web of Things is a hands-on guide that will teach how to design and implement scalable, flexible, and open IoT solutions using Web technologies. This book focuses on providing the right balance of theory, code samples and practical examples, to enable you how to successfully connect all sorts of devices to the Web and how to expose their services and data over REST APIs. After you build a simple proof of concept app, you'll learn a systematic methodology and system architecture for connecting things to the Web, finding other things, sharing data, and combining these components to rapidly build distributed applications and physical mashups. Gain the knowledge and skills you'll need to fully take advantage of a new generation of real-time, web-connected devices and services and to be able to build scalable applications that merge the physical and digital worlds.

What's inside

- Sense and connect the real world
- Build a Web interface to control your Smart Home using a Raspberry Pi
- Create a Web API for any device
- Build real-time physical mashups with JavaScript and node.js
- Integrate other protocols such as MQTT, CoAP or Bluetooth to the Web
- WoT and IoT platforms, tools, and protocols

Whether you're a seasoned developer, a system architect, or a curious amateur with basic programming skills, this book will provide you with a complete toolbox to become an active participant in the Web of Things revolution.

With the previous chapter about the Access layer, you learned how blending embedded devices into the web makes them so much more easily programmable. The request/response pattern of REST combined with the real-time power of WebSockets is ideal to offer simple web APIs for Things, but other patterns exist on the web. These patterns become especially valuable when considering use cases beyond your control. In the next chapter, "Getting data from clients: data ingestion" from *Streaming Data*, you'll discover a number of other web patterns for your Things, such as Publish/Subscribe, One-Way, and Request/Acknowledge. Finally, you'll explore the Stream pattern, which is especially useful when continuous flows of data need to be transmitted from Things to web platforms.

Chapter 2 from *Streaming Data*
by Andrew G. Psaltis

Getting data from clients: data ingestion

This chapter covers

- Learning about the collection tier
- Understanding the data collection patterns
- Taking the collection tier to the next level
- Protecting from data loss

Now on to our first tier: the collection tier is our entry point for bringing data into our streaming system. Figure 2.1 shows a slightly modified version of our blueprint, with focus put on the collection tier.

This tier is where data comes into the system and starts its journey; from here it will progress through the rest of the system. In the coming chapters we'll follow the flow of data through each of the tiers. Your goal for this chapter is to learn about the collection tier. When you finish this chapter you will have learned about the collection patterns, how to scale, and how to improve the dependability of the tier via the application of fault-tolerance techniques.

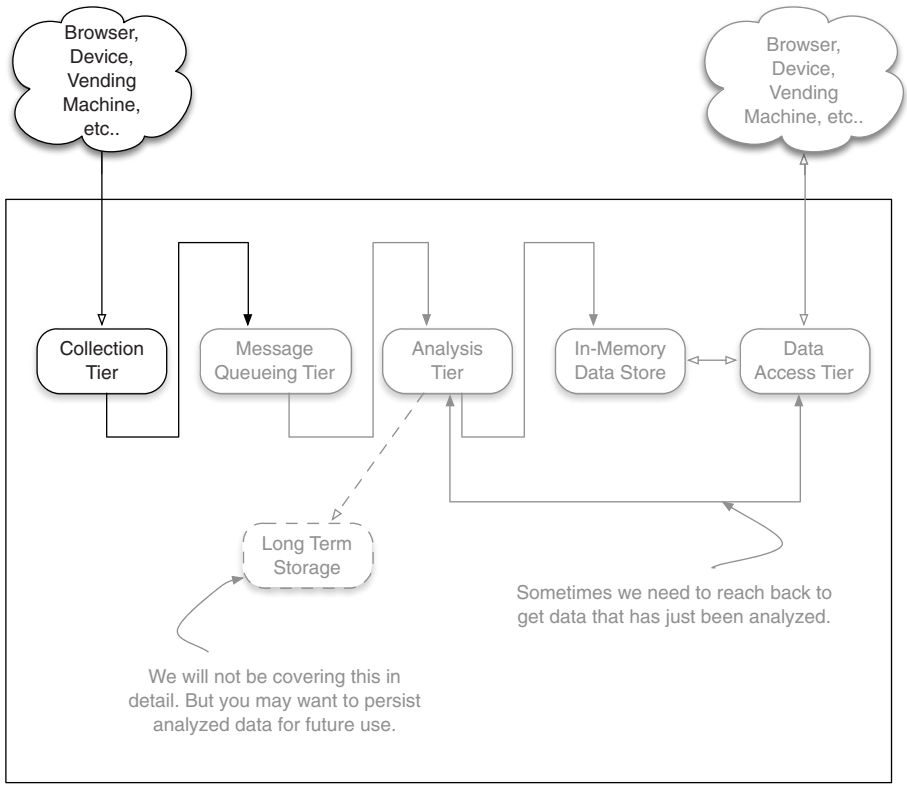


Figure 2.1 Architectural blueprint with emphasis on the collection tier

2.1 Common interaction patterns

Regardless of the protocol used by a client to send data to the collection tier—or in certain cases the collection tier reaching out and pulling in the data—a limited number of interaction patterns are in use today. Even considering the protocols driving the emergence of the *Internet of Everything*, the interaction patterns fall into one of the following categories:

- Request/response
- Publish/subscribe
- One-way
- Request/acknowledge
- Stream

At a high level we can group these interactions into two main categories:

- Request/response optional (request/response, publish/subscribe, one-way, request/acknowledge)
- Stream

Let's take a moment and explore each of these patterns and discuss how you might collect data using them.

2.1.1 Request/response

This is the simplest pattern and is used when the client must have an immediate response or wants the service to complete a task without delay. Every day you experience this pattern while browsing the web, searching for information online, and using your mobile device. This pattern works as follows: First, a client makes a request to a service; this may be to take an action (such as send a text message, apply for a job, or buy an airline ticket) or to request data (such as perform a search on Google or find the current weather in their city). Second, the service sends a response to the client. Figure 2.2 illustrates this pattern.

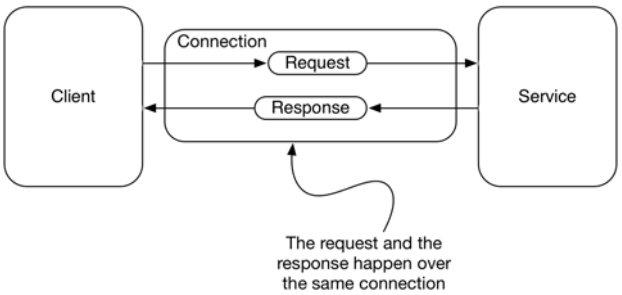


Figure 2.2 Basic request/response pattern

When you look at figure 5.2 it's apparent how simple this pattern is. One caveat of this pattern is that the request from the client and response from the service happen over the same connection in a synchronous fashion. This pattern is still widely used today and still relevant. The simplicity of a synchronous request and response comes at the cost of the client having to wait for the response and the service having to respond in a timely fashion. With modern-day services this cost often results in an unacceptable experience for users. Imagine browsing to your favorite news or social site and your browser trying to request all the resources in a synchronous fashion. Outside of basic services such as requesting the current weather, the potential delay is no longer tolerable. In many cases this can translate into lost revenue for merchants, because users don't want to wait for the response. There are three common strategies used to overcome this limitation: one client side, one service side, and the last a combination of the two. Let's consider the client side first. A common strategy often taken by the client is to make the requests asynchronously; this approach is illustrated in figure 2.3.

With this adaptation the client makes the request of the service and then continues on with other processing while the service is processing the request. This is the pattern used by all modern web browsers; the browser makes many asynchronous requests for resources and then renders the image and/or content as it arrives. This type of processing allows the client to maximize the time normally spent waiting on the response; the end result is an overall increase in the work performed by the client

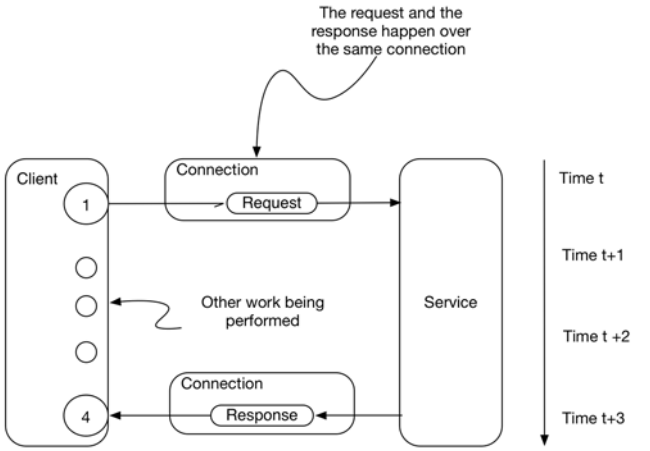


Figure 2.3 Client making asynchronous request to the service

over a period of time. Implementing this type of pattern is relatively easy today because all modern programming languages and many of the frameworks you may use natively support performing the request asynchronously. This pattern is often called *half-async* because one half of the request response is done asynchronously. Implementing this type of processing on the service end is also very common and is illustrated in figure 2.4.

With the service side half-async pattern the service receives a request from a client, delegates the work to be done, and when the work is finished responds to the client. This type of processing results in the development of more scalable services, which are able to handle requests from many more clients. This type is also very common in all server-side development frameworks found today for all popular programming languages.

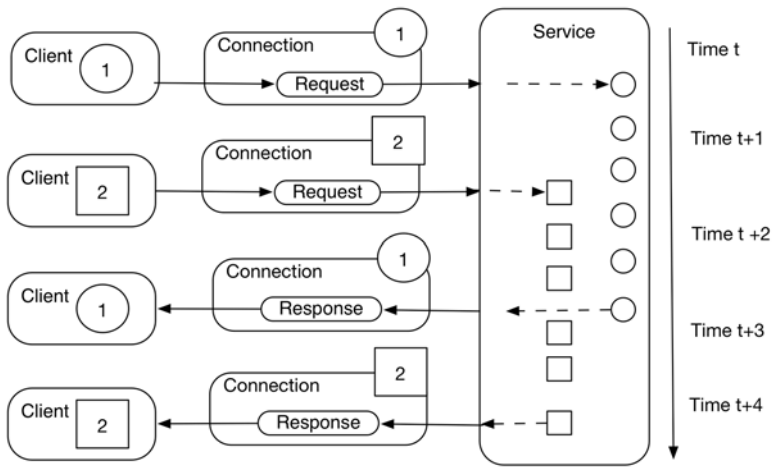


Figure 2.4 Service async request/response pattern

The last variation of this pattern occurs when both the client and the server perform their work asynchronously; the resulting flow is the same as that shown in figure 2.4. When both the client and the server are performing their work asynchronously, we call this pattern *full-async*. Today many modern clients and services operate in this fashion. Now let's walk through an example of how we can use this pattern in the design and the development of a streaming system.

Imagine for a moment that we work in the transportation industry and last week while enjoying coffee with our friend Eric, who works in the automotive industry, we came up with an idea to provide a real-time traffic and routing service for all vehicles on the road. Our company would build the service and Eric's company would build the streaming system that would reside inside the vehicles. We then sketched out what this solution would look like; starting with the vehicle part of it, figure 2.5 shows our high-level drawing of the vehicle side of things.

For the vehicle Eric is going to build an embedded streaming system to not just handle interacting with our traffic and navigation service but to also have the ability to interact with other services and perhaps vehicles.

The request/response pattern would work well for this scenario; in particular we'd want to choose the full-async variant. By choosing the full-async variant our traffic and

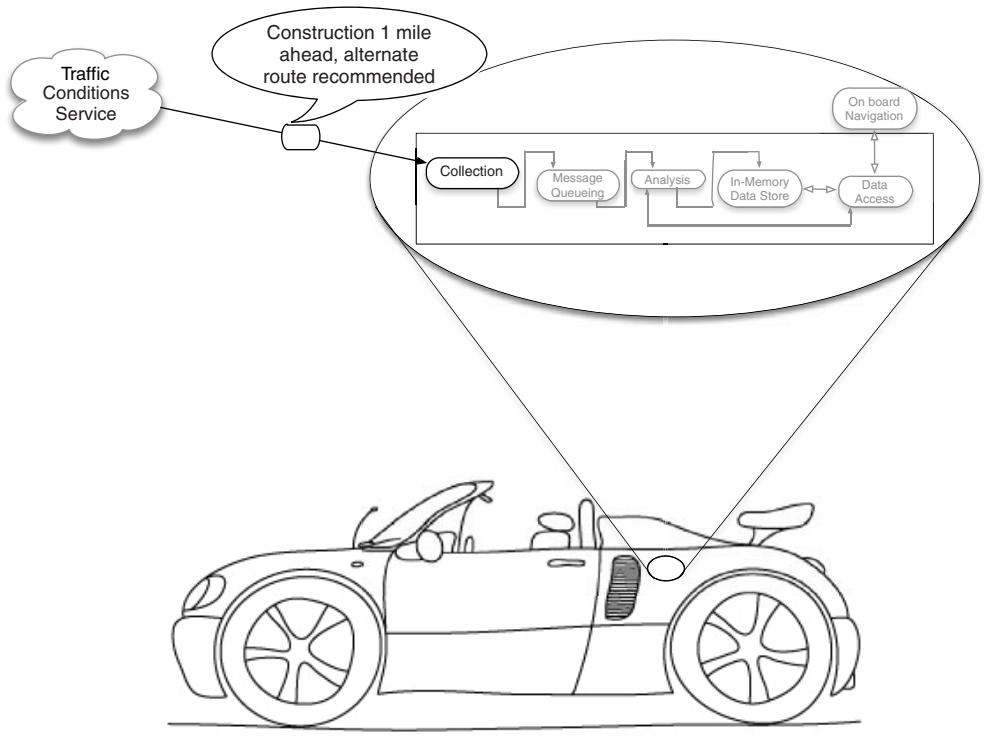


Figure 2.5 Receiving the response to the traffic conditions request with an on-board streaming system

navigation service would be better positioned to handle requests from a lot of cars on the road at a single time. For Eric's on-board streaming system, the ability to asynchronously request data and process it as it arrives would be essential. By following this pattern, the streaming system would not be blocked waiting for a response from our service and could handle other data analysis pertinent to the vehicle.

At this point we're ready for Eric's team to build the vehicle side of things and we're ready to build the traffic and routing service. If you are interested in learning more about this pattern, a good place to start is with Robert Daigneau's *Service Design Patterns* (Addison-Wesley, 2011).

2.1.2 Request/acknowledge

There are times when you need to use an interaction pattern with similar semantics to the request/response pattern but you don't need a response from the service. Instead, what you need is an acknowledgement that your request was received; the request/acknowledge pattern fits that need. Oftentimes the data sent back in the acknowledgement can be used to make subsequent requests, perhaps to check the status of the initial request or get a final response.

As an example, imagine we're working with the marketing department for our company to make sure that on our e-commerce site we provide the right offer to the right person at the right time, with the goal of increasing their likelihood of purchasing from us during their current visit. After further discussions with the marketing team we settled on a solution that would constantly update a visitor's propensity-to-buy score during their visit. With this dynamic score available, at any time our site can make the right offer to influence their decision to purchase. Figure 2.6 shows how this looks from a high level.

Let's walk through the flow of data illustrated in figure 2.6. As the visitor is browsing our site we're collecting data about each of the pages they're visiting and the links they're clicking. The unique thing we're doing that's particular to the request/acknowledge pattern occurs on the very first page they visit. On this page our collection tier returns an acknowledgment that can be used in future requests. Unlike the request/response pattern that may return as response success or failure, this pattern returns data that can be used in future requests. In this case the acknowledgement is nothing more than a unique identifier, but it plays an important role. The acknowledgement can be used on all subsequent pages the visitor visits. When we call the propensity service, we can pass the unique identifier we obtained on the very first visit. With the unique identifier, which identifies the visitor, our propensity service can determine and return the visitor's current propensity-to-purchase score. I realize that we're leaving out a lot of the details of how we go from collection to a propensity score, but don't worry; this will become clearer as we progress through the coming chapters. The key takeaway is that with the request/acknowledge pattern a client makes a request of a service, asking for an action to be taken, and in turn receives an acknowledgement token that can be used in future requests. If you think about it, you'll realize that we experience this pattern every day in real life. For example, when

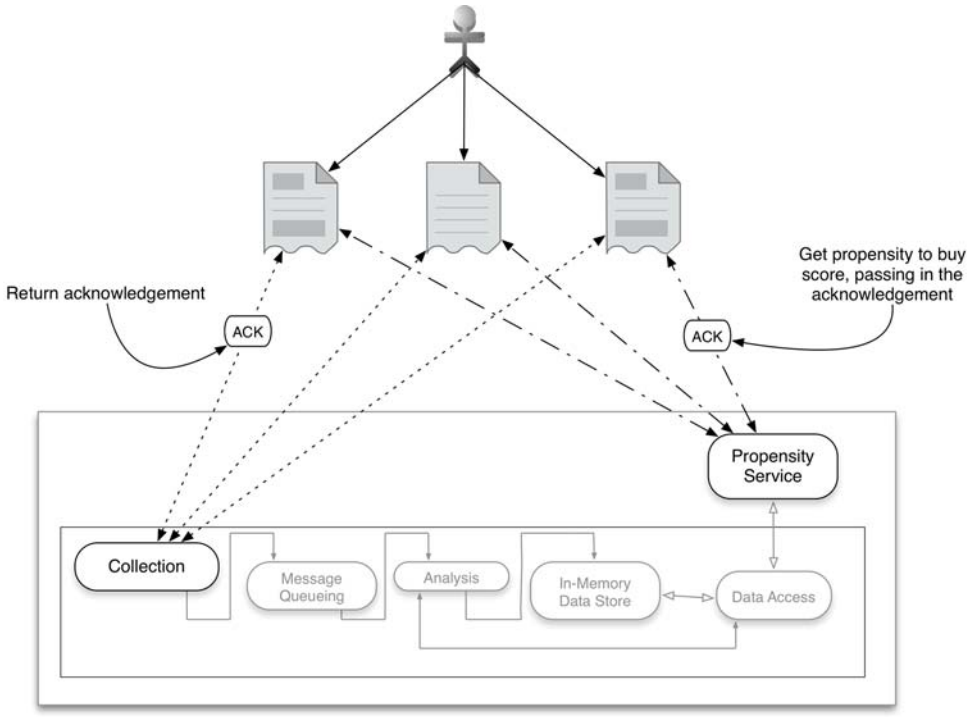


Figure 2.6 Visitor browsing while data is collected and their propensity-to-buy score is updated

you purchase an item online, you're often given a confirmation number. Subsequently, you can use this confirmation number to check on the status of your order.

If you're interested in learning more about this pattern, see Gregor Hohpe and Bobby Woolf's *Enterprise Integration Patterns* (Addison-Wesley, 2003).

2.1.3 Publish/subscribe

This is a common pattern with message-based data systems; the general flow is shown in figure 2.7.

The general data flow as illustrated in figure 2.7 starts with a producer publishing a message to a broker. The messages are often sent to a topic, which you can think of as a logical grouping for messages. Next, the message is sent to all the consumers who are subscribing to that topic. There's a subtlety in this last step that we'll cover in depth in chapter 3. For now, just realize that some technologies follow the data flow as illustrated here, pushing messages to consumers. But with other technologies the consumer pulls messages from the brokers. It may not be obvious at first, but oftentimes just because a producer publishes a message it doesn't mean that it needs to subscribe to a topic. Nor is it required that a subscriber produce a message.

Let's walk through an example of how this protocol can be used and its impact on our collection tier. After the success of our joint venture with Eric's company, we

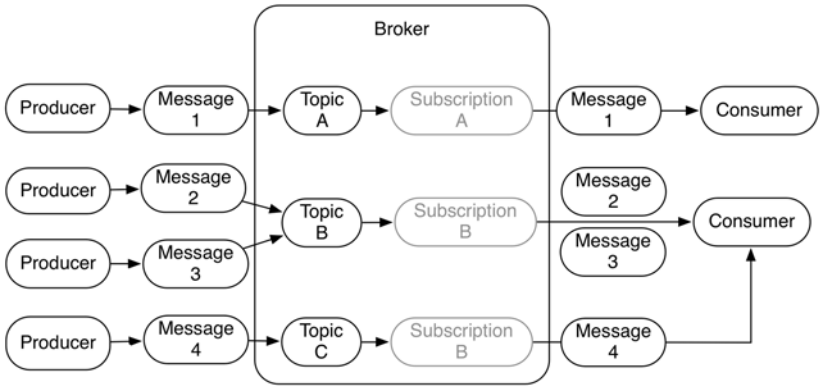


Figure 2.7 General data flow for the publish/subscribe message pattern

started to think about how we can take our in-vehicle traffic and routing service to the next level. After considering several ideas, we settled on the idea of making it social. In addition to the vehicle requesting traffic information and routing, it would send real-time traffic updates back to the service and subscribe to the real-time traffic reports from other vehicles traveling along the same route. Figure 2.8 shows the flow of messages we're talking about.

For simplicity we're showing only a handful of cars acting as producers and sending their current traffic conditions to the broker and a single car acting as the con-

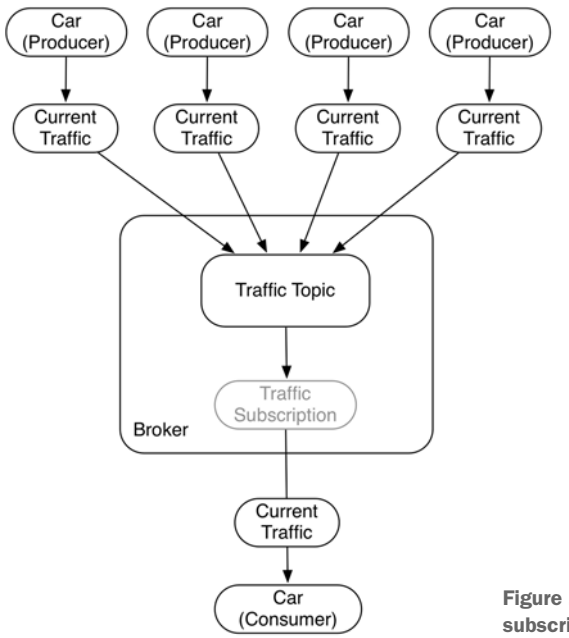


Figure 2.8 Current traffic publish/subscribe message pattern

sumer. If this was real, you could imagine how each producer would also consume and analyze all of the data. By using the publish/subscribe pattern we're able to decouple the sender of the traffic data from the consumer of it. As we scale this simple example of four cars sending data and one consuming data to all cars in the United States, I think you can imagine how important the decoupling this pattern provides is. If you're interested in learning some of the finer points about this pattern, a good place to start is *Enterprise Integration Patterns*, mentioned previously.

2.1.4 One-way

This interaction pattern is commonly found in cases where the system making the request doesn't need a response. Often you may also see this pattern referred to as the "fire and forget" message pattern. In some cases this pattern has distinct advantages and may be the only way for a client to communicate with a service. This pattern is similar to the request/response and request/acknowledge patterns in the way a message is sent from the client to the service. The major difference is that the service does not send back a response. Whereas in the other patterns the client knows the request was received and processed, in the case of the one-way pattern the client doesn't even know if the request was received by the service.

You may be wondering how or where a pattern that has zero guarantees that the message was even received by a service can be useful. This pattern is useful in environments where a client doesn't have the resources or the need to process a request. For example, think of the data available about the servers in your data center. You'd like for the server to send data about how much memory and CPU are being used every 10 seconds. You don't need the server to take any action or even worry about the result; it's purely producing data as fast as possible. Examples of this interaction pattern appear all around us and will continue to grow along with the proliferation of the Internet of Everything. It's infiltrating many aspects of our life, and sports is no exception. For example, a recent partnership between the NFL and Zebra (http://www.zebra.com/us/en/nfl.html) resulted in players during Thursday Night Football games being outfitted with quarter-sized radio frequency identifier (RFID) tags on their equipment. This tag will transmit data, such as the athlete's movement, distance, and speed, approximately 25 times a second to the 20 RFID receivers installed in the stadium. Within half a second, the data will be analyzed and relayed to the TV broadcast trucks to be used by commentators. In this scenario the RFID tag, which is the client, does not need and does not have the resources to process a response from the RFID receiver. Another aspect about this example as it relates to this pattern is that the data is being sent 25 times a second. If during one second five samples were lost and they were not received by the RFID receiver, would the resulting analysis be impacted? No, it would not. That's another noteworthy characteristic of this pattern—it is appropriate and often found in environments where losing some data is tolerable in exchange for simplicity, reduced resource utilization, and speed. To learn more about this pattern, see Nicolai M. Josuttis's *SOA in Practice* (O'Reilly, 2007).

2.1.5 Stream

This interaction style is quite different than all of the others we've talked about thus far. With all of the other patterns a client was making a request to a service that may or may not have returned a response. With this pattern we're flipping things around and the service becomes the client. A comparison of this to the other patterns you've seen is illustrated in figure 2.9.

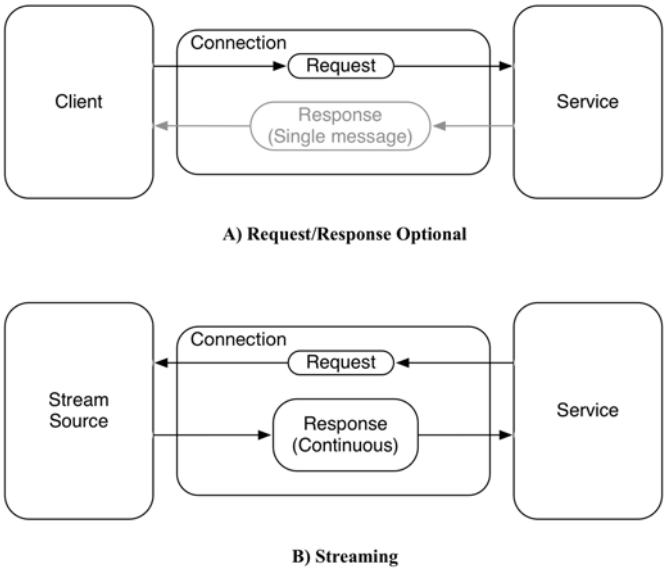


Figure 2.9 Comparing the request/response patterns to the stream pattern

There are a couple of important distinctions to point out when comparing the previous patterns (all the request/response optional patterns) with the stream pattern:

- With the request/response style of interaction as depicted at the top of figure 2.9, the client pushes data to the service in the request and the service may respond. This response is grayed out in the diagram, because the response is not required by some variations of this pattern. It boils down to a single request resulting in zero or one response. The stream pattern as depicted at the bottom of figure 2.9 is quite different; a single request results in no data or a continual flow of data as a response.
- The second difference between the request/response optional patterns and the stream pattern is that in the former a client external to the streaming system is pushing the message to it. In our previous examples this was a web browser, a car, or a phone—all clients that send a message to our collection tier. In the case of the stream pattern, our collection tier connects to a stream source and pulls data in. For example, you may be interested in building a streaming system to do sentiment analysis of tweets. To do so you'd build a collection tier that establishes a connection to Twitter and consumes the stream of tweets.

The usage of this pattern is very interesting and powerful; you're ingesting a stream of data and producing another stream. With this you can quickly build a streaming analysis system that consumes publicly available data and in turn create new streams of data based on your analysis. Unlike the other patterns where you need to create or find clients to send a request to your service, with the stream pattern you can chose to connect to and process the data from a stream source. The U.S. government provides an example input stream that you can use for exploring this interaction pattern or as input to a streaming system. The stream is composed of JSON events, each of which is generated every time someone clicks a 1.USA.gov URL, which is any .gov or .mil URL that has been Bit.ly shortened. To see this input stream in action, open your favorite browser and go to http://developer.usa.gov/1usagov. In this case, a simple, long-lived HTTP connection is established and data is subsequently streamed back to your browser until you end the HTTP connection. In the data stream you'll see JSON events that are similar to the following listing.

Listing 2.1 Example JSON stream event

```
{
    "h": "1t2pQ2p",
    "g": "1guGHEx",
    "l": "tweetdeckapi",
    "hh": "1.usa.gov",
    "u":
     "http://www.nasa.gov/aero/infographics.html",
    "r": "http://t.co/jEtfi0v786",
    "a": "Mozilla/5.0 (Linux; U; Android 4.3; es-us; SGH-T889 Build/JSS15J)
     AppleWebKit/534.30 (KHTML, like Gecko) Version/4.0 Mobile
     Safari/534.30",
    "t": 1416409750,
    "nk": 1,
    "hc": 1416344005,
    "_id": "546cb296-0003c-0784b-321cf10a",
    "al": "es-US, en-US",
    "c": "PR",
    "tz": "America/Puerto_Rico",
    "gr": "00",
    "cy": "Ensenada",
    "ll": [
        17.9638,
        -66.9452
    ]
}
```

The destination page that this event is for

The referring URL that led to the destination page

This is an example of using the stream interaction pattern. Imagine if you took this data and combined it with social data such as tweets about certain URLs. Again that's something that's hard to replicate with the other patterns. As you look at this data I'm sure you'll come up with a variety of questions about it without combining it with other streams. Perhaps you'd want to count the top pages viewed or maybe the top referrers by city. But let's not get ahead of ourselves; we're going to work through how

to answer these and other types of questions in chapter 4. For now it's enough to recognize this pattern and start to get a feel for this interaction pattern. If you're interested in learning more about this dataset, please see http://www.usa.gov/About/developer-resources/1usagov.shtml.

2.2 Scaling the interaction patterns

Now that we've discussed each of the interaction patterns, let's see how we'd scale our collection tier and talk about some of the things to keep in mind when implementing it. We're going to keep the discussion at the level of the two categories we grouped the interaction patterns into before.

2.2.1 Request/response optional

To discuss scaling this general pattern we'll continue with the example from our initial discussion of the request/response pattern, the real-time traffic and routing service for all vehicles on the road. To get a better sense for the scale of our idea—to provide this service for all vehicles on the road in the United States—we'll consider the 2012 (the last complete year) National Transportation Statistics report (produced by the Bureau of Transportation Statistics, http://www.rita.dot.gov/). According to this report approximately 253 million vehicles were registered in the United States and we collectively drove approximately 2.966 trillion miles. This means that at any time during the almost 3 trillion miles driven by one of the 253 million vehicles, we may get a request for the current traffic conditions and alternate route suggestions. Undoubtedly at any moment we'll need to handle thousands and possibly millions of requests. If you remember, in chapter 1 we talked about horizontal scaling being our overall goal for every tier of our streaming system. With this example and our use of the request/response optional pattern, horizontal scaling will work very well for two reasons: First, with this pattern we don't have any state information about the client making the request, which means that a client can connect and send a request to any service instance we have running. Second—and this is a result of the stateless nature of this pattern—we can easily add new instances of this service without changing anything about the currently running instances. The mode of scaling stateless services is so popular that many cloud hosting providers, such as Amazon, provide a feature called auto-scaling that will automatically increase or decrease the number of instances running based on demand. On top of horizontal scaling we also want our service to be stateless, which will allow any vehicle to make a request to any instance of our service at any time. This stateless trait is commonly found in systems that use this pattern. Taking horizontal scaling and statelessness into consideration, we arrive at figure 2.10, which shows these two aspects together.

We're using a load balancer here to be able to route requests from the vehicles to an instance of our service that's running. As instances are started or stopped based on demand, the running instances the load balancer routes requests to will change. We now have a pretty good idea of how we're going to scale our service and the protocol we're going to use with our clients.

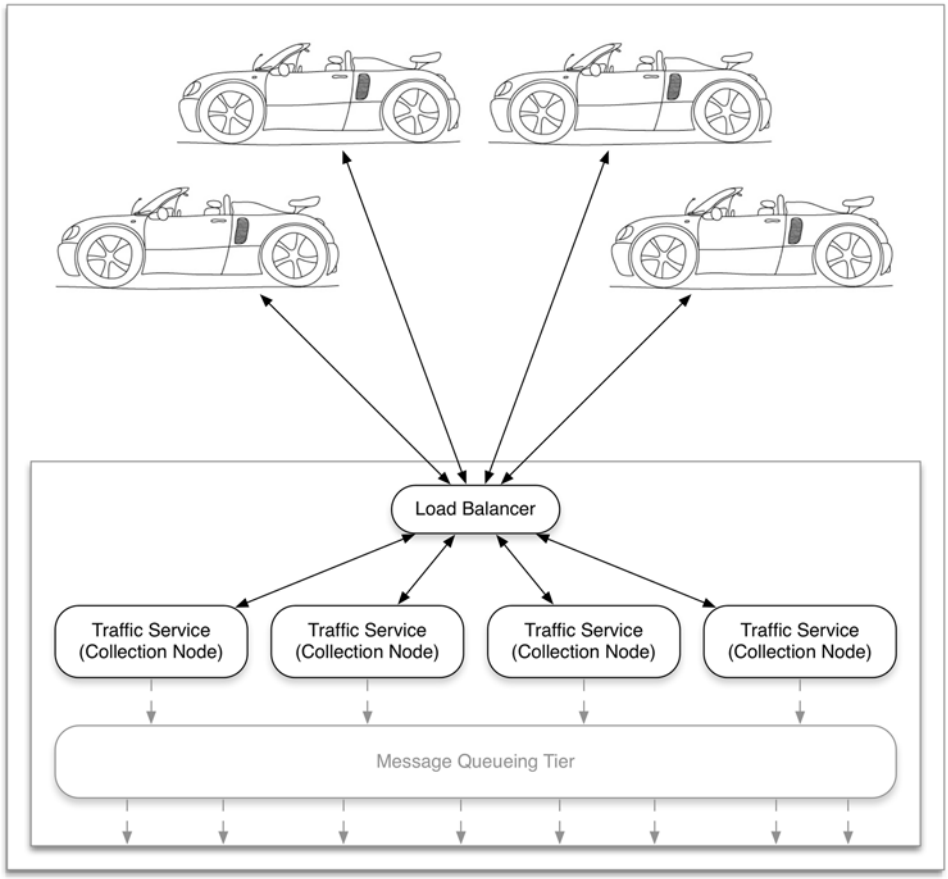


Figure 2.10 Vehicle and traffic service with a load balancer

2.2.2 Scaling the stream pattern

The 1.USA.gov stream we used as an example in section 2.1.5 for our discussion of the stream interaction pattern has a fairly low velocity (less than 10 events per second). Obviously that's not the best example to help us think through how to scale a collection tier when using the stream interaction pattern. Instead, let's imagine that Google provided a public stream of all the searches being performed as they happen; according to internetlivestats.com (http://www.internetlivestats.com/one-second/#google-band) that would result in approximately 46,000 search events per second. Previously we mentioned that horizontal scaling is our goal when building each tier of our streaming system. With many streaming protocols, just like you saw earlier when you consumed the 1.USA.gov stream in your browser, there's a direct and persistent connection between the client (our collection tier) and the server (the service we request data from), as illustrated in figure 2.11.

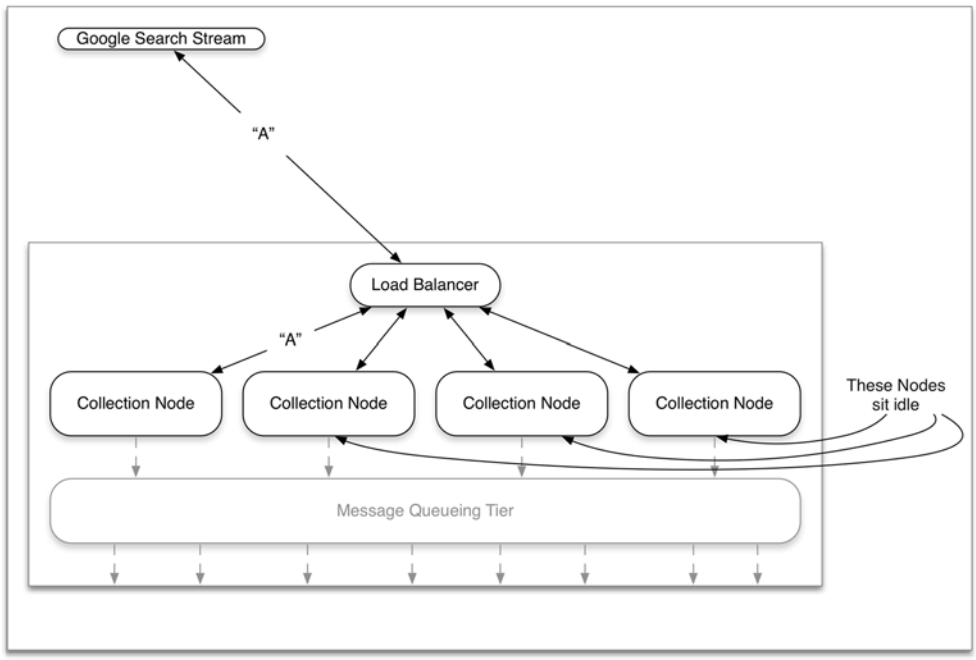


Figure 2.11 Search stream with direct connection to single collection node

In figure 2.11 you can see that three of the four nodes are idle because there's a direct connection between the search stream and the node handling the stream. To scale our collection tier we have a couple of options: scaling up the collection node that's consuming the stream and introducing a buffering layer in the collection tier. These are not mutually exclusive, and depending on the volume and velocity of the stream both may be required. Scaling up the node consuming the stream will only get us so far; at a certain point we'll reach the limits of the hardware our collection node is running on and won't be able to scale it up any further. To aid in preventing this from happening we'll introduce a buffering layer. Figure 2.12 shows what our collection tier looks like with the buffering layer in place.

The key to being able to put a buffering layer in the middle lies in making sure that no business logic is performed on the messages when they're consumed from the stream. Instead, they should be consumed from the stream and as quickly as possible pushed to the buffering layer. A separate set of collection nodes that may perform business logic on the messages will then consume the messages from the buffering layer. The second set of collection nodes can now also be scaled horizontally.

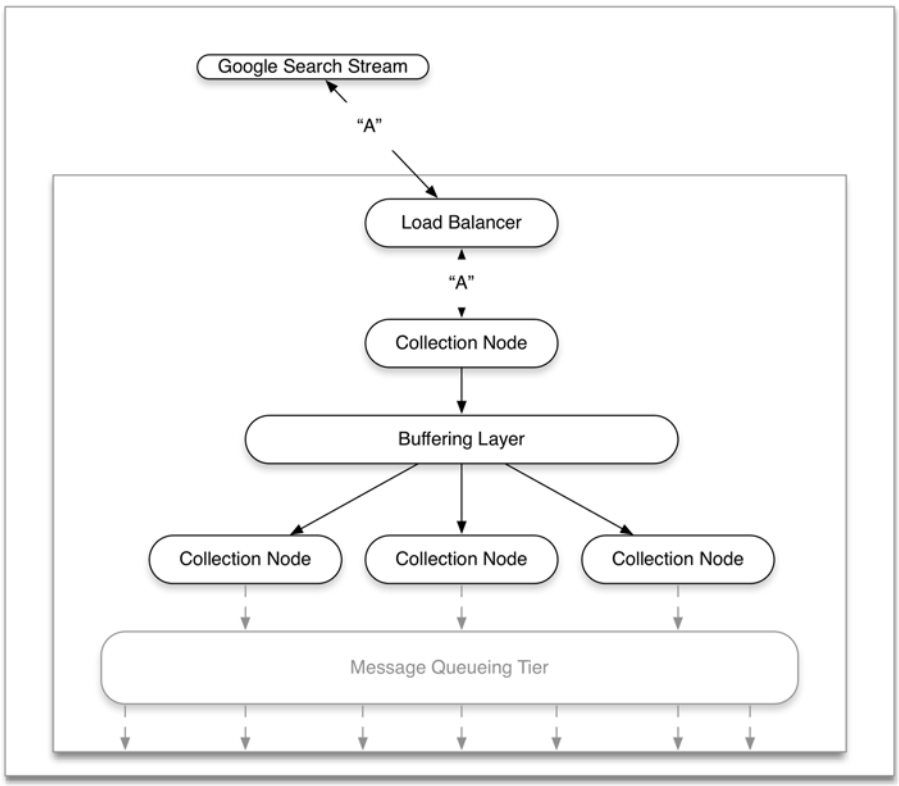


Figure 2.12 Collection tier with buffering layer in place

2.3 Fault tolerance

Regardless of the interaction pattern used, one thing is for sure: at some point we will have a failure with one or more of our collection nodes. The failure may be the result of a bug in our software, third-party software we rely on, or the hardware our service runs on. Regardless of the cause, our goal is to mask the failures and to improve the dependability of our collection tier. You may be wondering why we need to worry about this if we've done our job of horizontally scaling and increasing the redundancy of our tier. That's a fair question to ask. The answer actually is quite simple; the message our collection tier receives from a client may not be reproducible. In essence, there may be no way for our collection tier to ask for the client to send us the data again and in many cases no way for the client to actually do so even if our collection tier could ask. Depending on your business, there may be times when it's okay to lose data, but in many cases it's important that you do not lose data. In this section we're going to explore the fault-tolerance techniques we can employ to help us ensure we do not lose data and we improve the dependability of our collection tier. Our overarching goal is that when a collection node crashes (and it will) we do not lose data

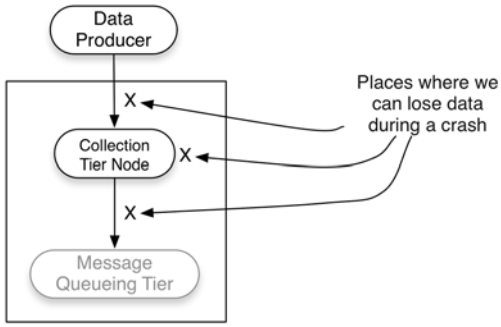


Figure 2.13 Collection scenario with our data loss potential identified

and can recover as if the crash had never occurred. To understand the areas we need to protect, take a look at figure 2.13, which shows the simplest possible collection scenario with the places we can lose data when the node crashes.

The two primary approaches to implementing fault tolerance, *checkpointing* and *logging*, are designed to protect against data loss and enable speedy recovery of the crashed node. The characteristics of checkpointing and logging are not the same, as you'll soon see.

First, let's consider checkpointing. There are a variety of checkpoint-based protocols in the literature to choose from, but when you boil them down, the following two characteristics can be found in all of them:

- *Global snapshot*—They require that a snapshot of the global state of the whole system be regularly saved to storage somewhere, not just the state of the collection tier
- *Potential for data loss*—They only ensure that the system is recoverable up to the most recent recorded global state; any messages that were processed and generated afterward are lost

What does it mean to have a global snapshot? It means we're able to capture the entire state of all data and computations from the collection tier through the data access tier and save it to a durable persistent store. This is what we're talking about when we refer to the *global state* of the system. This state is then used during recovery to put the system back into the last known state. The potential for data loss exists if we can't capture the global state every time data is changed in the system. An example I'm sure you've seen is AutoSave in popular document-editing software such as Microsoft Word or Google Docs. In both cases a snapshot is taken of the document as you are editing it, and if the application crashes you can recover to the last checkpoint. If you're like many people, you've seen checkpointing and the potential for data loss in action. Perhaps, like many, you've encountered the situation where a word processing program crashes and your most recent edits were not saved. When considering using a checkpoint protocol for implementing fault tolerance in a streaming system, you have to keep two things in mind: the implications of the previously mentioned attributes and the fact that a

streaming system is composed of many layers and many different technologies. This layering and the data movement make it very hard to consistently capture a global snapshot at a point in time. This makes checkpointing a bad choice for a streaming system. But checkpointing is a valid choice if you're building the next version of HDFS or perhaps a new NoSQL data store. Given that checkpointing isn't a good match for a streaming system, we won't spend more time on these protocols. Even though they're not a good fit, they're fascinating to study. If you're interested in learning about them, I would encourage you to start with the following great article by Elnozahy, En Mootaz, et al, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems" (*ACM Computing Surveys* 34.3 (2002): 375–408).

Turning our attention to the logging protocols, you have a variety to choose from. Reducing them to their essence, you'll find that they all share the common goals of overcoming the expense and complexity of checkpointing and providing the ability to recover up to the last message received before a crash. Part of the complexity of checkpointing that's eliminated is the global snapshot and thus the management and generation of the global state. In the end the goals of the logging technique manifest themselves in the basic idea that underpins all of the logging techniques: *If a message can be replayed, then the system can reach a global consistent state without the need for a global snapshot.*

This means that each tier in the system independently records all messages it receives and plays them back after a crash. Implementing a logging protocol frees us from worrying about maintaining global state, enabling us to focus on how to add fault tolerance to the collection tier. To do this we're going to discuss two classic techniques, *receiver-based message logging (RBML)* and *sender-based message logging (SBML)*, and an emerging technique called *hybrid message logging*. Along the way we'll also discuss how and why we can use these with our collection tier. Before moving on to discuss these different techniques, take a look at figure 2.14, which illustrates how they fit together and what data we're trying to protect.

Figure 2.14 shows a single collection tier node that's receiving a message, performing some logic on it, and then sending it to the next tier. As their names imply, receiver-based logging is concerned with protecting the data the node is receiving and sender-based logging is concerned with protecting the data that's going to be sent to the next tier. You can imagine your business logic being sandwiched between two layers of logging, one designed to capture the data before it's changed and one to capture it before it's sent to the next tier. If you're thinking that this is a lot of potential overhead and overlap, in some cases it may be, and this is where hybrid message logging (HML) aims to strike a balance between RBML and SBML. With that frame of reference let's start our discussion with receiver-based logging.

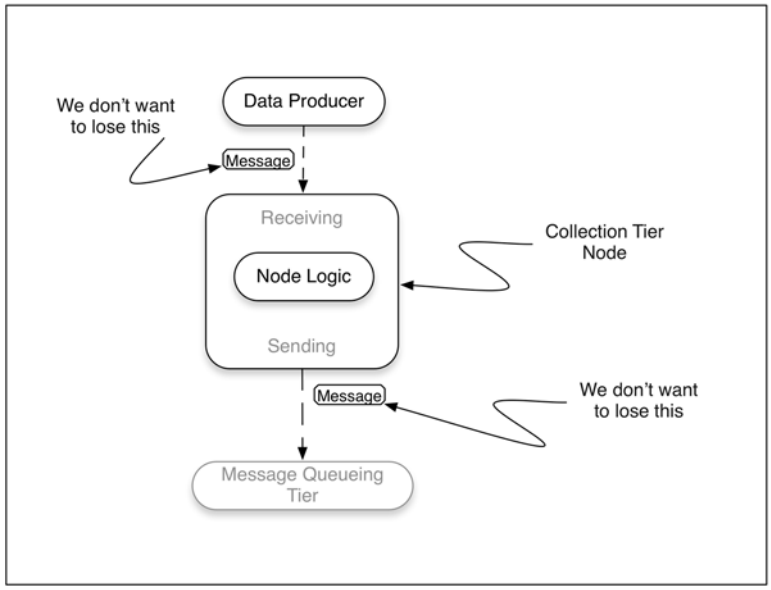


Figure 2.14 High-level overview of receiver-based and sender-based message logging

2.3.1 Receiver-based message logging

The RBML technique involves synchronously writing every received message to stable storage before any action is taken on it. By doing that, we can ensure that when our software crashes while handling the message, we already have it saved and upon recovering we can replay the message. Figure 2.15 illustrates how our collection node changes with the introduction of RBML.

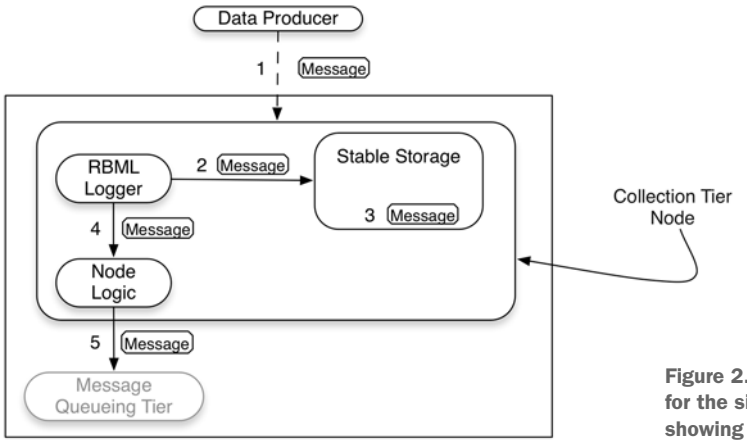


Figure 2.15 RBML implemented for the simple collection node showing the happy path

In figure 2.15 the message flows from step 1 through to step 5; this shows the happy path when there is no failure. We'll walk through the recovery side of it shortly, but first let's take a moment and briefly review the flow:

1. A message is sent from a data producer (any client).
2. A new piece of software we wrote for the collection node, called the RBML logger, gets the message from the data producer and sends it to storage.
3. The message is written to stable storage.
4. The message then proceeds through to any other logic we have in the node; perhaps we want to enrich the data we are collecting, filter it, and/or route it based on business rules. The important aspect is we are recording the data as soon as it is received and before we do anything to it.
5. The message is then sent to the message queueing tier, the next tier in the streaming system.

It's important to point out that depending on the type of stable storage used, steps 2 and 3 have the potential of negatively impacting the throughput performance of our collection node. Sometimes in the literature you'll see this called out as one of the drawbacks to logging protocols. The hybrid message logging technique we'll discuss in section 2.3.3 helps to address some of those concerns. For now we'll keep it simple—at the end of the day the simplicity and recoverability of using RBML for our collection node wins.

Now that you understand how the data flows during normal operation, take a look at figure 2.16, which shows what the recovery data flow looks like.

In figure 2.16 there are a couple of things to call out. First, once the crash occurs, all incoming messages to this collection node are stopped. Because you'll have more than one collection node and they'll be behind a load balancer, you would take this node out of rotation. Next, the RBML logger reads the messages that have not been processed from stable storage and sends them through the rest of the node logic as if

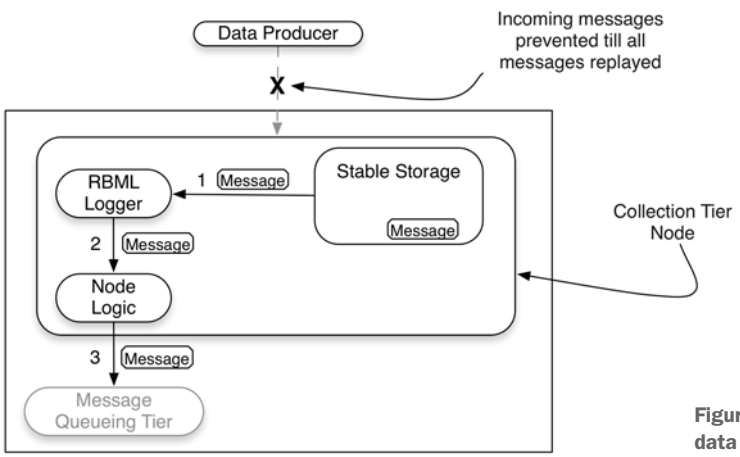


Figure 2.16 The recovery data flow for RBML

nothing has happened. Lastly, after all of pending messages are processed, the node is considered restored and can be put back into rotation, and the data flow resumes as in figure 2.15.

2.3.2 Sender-based message logging

The SBML technique involves writing the message to stable storage before it is sent. If you think of the RBML technique as logging all messages that come in the front door of our collection node as a means to protect us from ourselves, then SBML is the act of logging all outgoing messages from our collection node before we send them, protecting ourselves from the next tier crashing or a network interruption. Figure 2.17 shows the data flow for SBML.

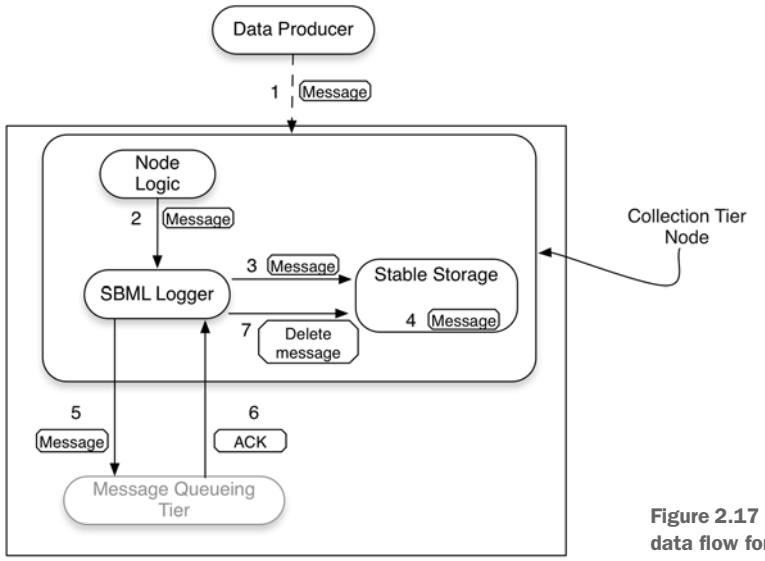


Figure 2.17 The normal execution data flow for SBML

Now that you understand RBML and in particular the data flow, I suspect that the data flow for SBML as depicted in figure 2.17 seems fairly reasonable to you through step 5. One thing to keep in mind that's different between the RBML data flow and SBML is that with RBML we are recording the message as soon as it is received before we do anything to it, and with SBML we are recording it before we send any data to the next tier. Thus the data recorded by an RBML logger is the raw incoming data and the data recorded by the SBML logger is after our node logic (remember, we may have augmented the data in some way) executes and before we send it on. Besides this nuance there's a little wrinkle we'll need to deal with during recovery. During recovery how do we know if the next tier has already processed the message we're replaying? There are several ways to handle this. One that is shown in figure 2.17 is that we use a message queueing tier that returns an acknowledgement that it received the message. With that acknowledgement in hand, we can either mark the message as replayed in stable

storage, or we can delete it from stable storage because we don't need to replay it anymore during recovery. If the technology you choose for your message queueing tier doesn't support returning an acknowledgement of any sort, then you may be forced into a situation where if no error occurs when sending the message to the message queueing tier, steps 6 and 7 will result in you deleting the message from stable storage. The recovery data flow as illustrated in figure 2.18 is a little more complex than how we handled recovery with the RBML.

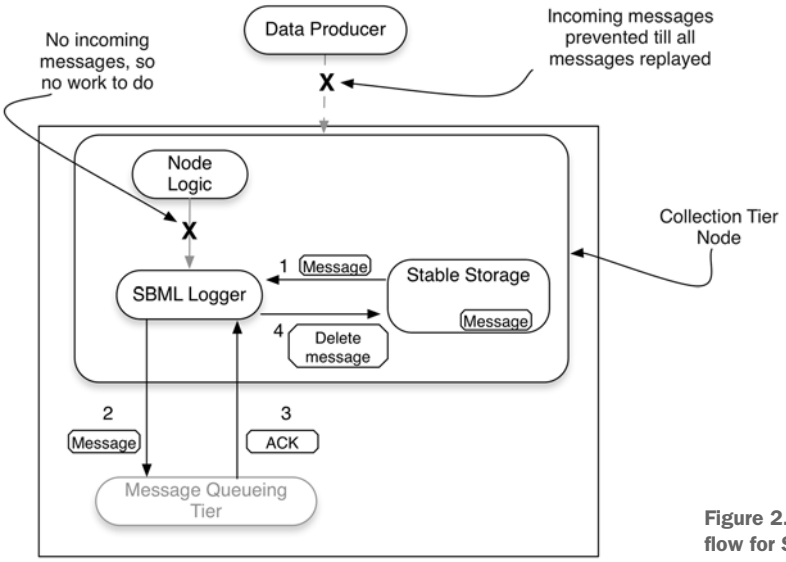


Figure 2.18 Recovery data flow for SBML

I think you'll agree that the recovery data flow for SBML is only marginally more complex than the RBML workflow, but it shouldn't look too foreign to you.

2.3.3 Hybrid message logging

If we stopped right now, we'd have two solutions that we can put in place to address our data loss concerns and dependability: RBML to handle the incoming messages and SBML to handle the outgoing messages. As we discussed, writing to stable storage can negatively impact our collection node's performance. Implementing both RBML and SBML means we're writing to stable storage at least twice during normal execution. Some may argue that we'll be doing more logging than processing of data; looking at figure 2.19 might lead you to believe they may not be far off.

To help with this, an emerging technique called hybrid message logging has been designed to take the best parts of RBML and SBML at the cost of minimal additional complexity. HML is also designed to provide the same data loss protection and recoverability found in RBML, SBML, and other logging techniques. There are several ways to implement HML; one common approach is illustrated in figure 2.20.

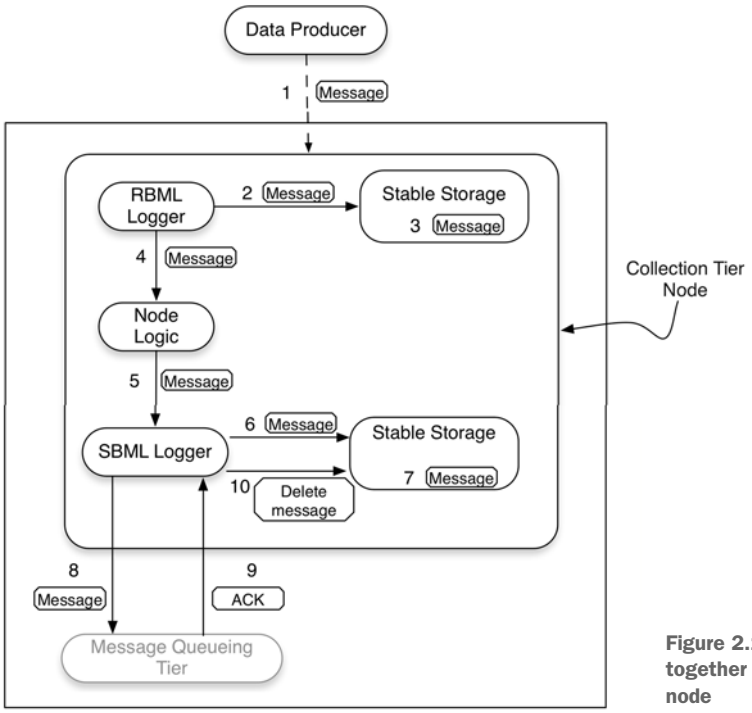


Figure 2.19 RBML and SBML together in the collection tier node

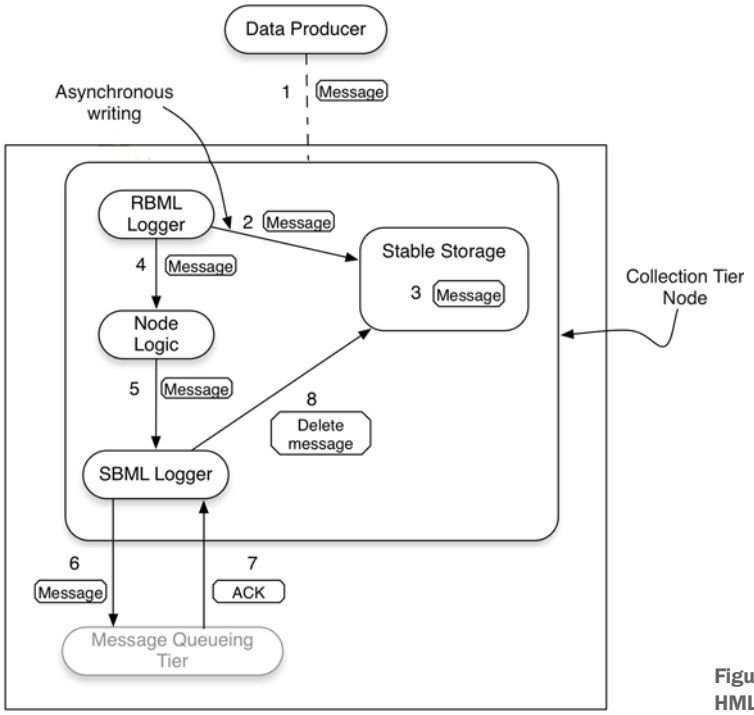


Figure 2.20 HML sample data flow

It's apparent when comparing the data flow shown in figure 2.19 with both RBML and SBML to the HML data flow shown in figure 2.20 that the HML approach is slightly less complex. Several factors contribute to this simplification. The first one, which may not come as a surprise, is that the two stable storage instances have been consolidated. This is a minor change, but it allows you to reduce the number of moving parts. The second change, writing to stable storage asynchronously, has a subtle difference. Arguably this has a more profound impact on the implementation complexity and performance. The complexity comes from making sure you are correctly handling any errors that happen, and the performance comes from leveraging the multi-core world we live in to perform more than one task at a time. The rest of the data flow should be routine for you by now. If you feel comfortable with the additional complexity and have a choice of implementing HML or standard RBML and SBML, you should implement HML. Regardless of whether you are implementing HML or not, if you're interested in learning more about this protocol a great article to start with is Meyer, Rexachs, and Luque, "Hybrid Message Logging. Combining advantages of Sender-based and Receiver-based Approaches" (*Procedia Computer Science* 29 (2014): 2380–90).

2.4 A dose of reality

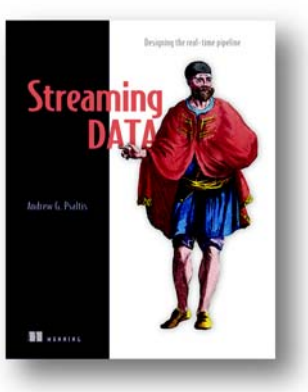
Here's a funny little story to put some of this scaling and fault tolerance into perspective. One time I was working on a streaming system that was populating fancy dashboards for marketers. It had all the bells and whistles—scaling, fault tolerance, monitoring, alerting—the whole nine yards. We had to have all of this and could not lose any data, because our customers wouldn't accept a solution that didn't have complete data. Once this system was running in production, I was curious as to how well our web-based dashboards that consumed our stream via WebSockets were keeping up. Well, come to find out, many of our customers were only able to keep up with about 60% of the stream that was being sent to them; the other 40% of the data was being dropped because they couldn't read it fast enough. When I mentioned this to coworkers, they were shocked and somewhat in disbelief because our customers and business folks loved what they were seeing. It really put things in perspective: the dashboards we produced were showing a picture of our customers' business that was not distorted by the missing data. To me this was like the difference between the high-end HDTV and the mid-level HDTV—sure the quality of the picture may be slightly better, but the picture doesn't change. Now, I'm not implying that you don't need to worry about scaling or fault tolerance, but it's good to keep things in perspective and then reflect on the difference between "we must have *xyz* features" and reality.

2.5 Summary

We've covered a lot of ground in this chapter exploring the various aspects of collecting data for a streaming system, from the interaction patterns through scaling and the fault-tolerance techniques.

Along the way you

- Learned about the collection tier
- Developed an understanding of the various collection patterns
- Had a chance to interact with a live stream
- Learned how to think about scaling your collection tier
- Learned about the common fault-tolerance techniques



There's a big difference between sipping a glass of water and drinking directly from the hydrant. In the same way, applications built to deal with streaming data present fundamentally different challenges than those that work with stored data. For example, live location data paired with a social media profile might allow a vendor to recommend a product or service to a user at just the right instant, and the split-nanosecond reaction of a pacemaker or anti-lock brakes can save lives. Emerging techniques and technologies that enable you to take immediate action on streaming data make it possible to design and build in-the-moment decision systems, dynamic reporting dashboards, live recommendation systems, and other real-time applications.

Streaming Data introduces the concepts and requirements of streaming and real-time data systems. Through this book you will develop a foundation to understand the challenges and solutions of building in-the-moment data systems before committing to specific technologies. Using copious diagrams, this book systematically builds up the blueprint for an in-the-moment system concept by concept. Although code may occasionally appear in examples, this book focuses on the big ideas of streaming and real time data systems rather than the implementation details.

Many of the technologies discussed in the book—Spark, Storm, Kafka, Impala, RabbitMQ, etc.-are covered individually in other books. As you read, you'll get a clear picture of how these technologies work individually and together, gain insight on how to choose the correct technologies, and discover how to fuse them together to architect a robust system.

What's inside

- Architect a complete system for collecting and analyzing data in real time
- Harness the Internet of Things by handling live data from billions of devices
- Combine emerging technologies like Spark, Storm, Kafka, RabbitMQ, and WebSockets
- Integrating and extending the Lambda architecture into a complete system

No experience with streaming or real-time data systems required. Perfect for developers or architects, this book is also written to be accessible to technical managers and business decision makers.

The Find layer

With the Access layer we ensure Things are accessible on the web. However, making Things accessible via a web API doesn't mean a client can "understand" what the Thing is, what data or services it offers, and so on. The Find layer deals with this problem. In *Building the Web of Things*, we propose a web-based protocol with a set of resources, data models, a payload syntax, and semantic extensions that web Things and applications should follow. This ensures that your Things and the services they provide can be easily understood and used by other web clients.

However, that isn't the end of the story. A web page offers nothing if users can't find it, and the same goes for Things. The Find layer looks into making Things findable. One interesting technique is to make them searcheable. Just like a lonely web page starts to attract traffic once it is indexed by Google, Things can benefit from being indexed by search engines. Imagine a not-too-distant future where you can Google your running shoes to locate them instead of desperately rooting through closets in your chaotic physical world!

In the next chapter, "Enhancing results from search engines" from *Linked Data: Structured Data on the Web*, you'll learn how to make any page efficiently searchable using the Semantic Web. The same approach can be applied to the pages of Things!

Chapter 6 from *Linked Data: Structured Data on the Web*
by David Wood, Marsha Zaidman, Luke Ruth,
and Michael Hausenblas

Enhancing results from search engines

This chapter covers

- Adding Resource Description Framework in Attributes (RDFa) to HTML
- Using RDFa and the GoodRelations vocabulary to enhance HTML
- Using RDFa with schema.org
- Applying SPARQL to extracted RDFa

Previous chapters covered discovering Linked Data on the Web. This chapter will guide you in enhancing your own web pages with RDFa. We'll start with a typical HTML web page designed for human readability and demonstrate how to embed RDFa content that will enable your page to be both human- and machine-consumable. The presence of this Linked Data will improve search engine optimization (SEO) and the likelihood that your web content will be discovered.

We'll then convert a web page designed to showcase a consumer product and embed RDFa that uses the GoodRelations vocabulary. These improvements will improve the discovery of this product by common search engines like Google,

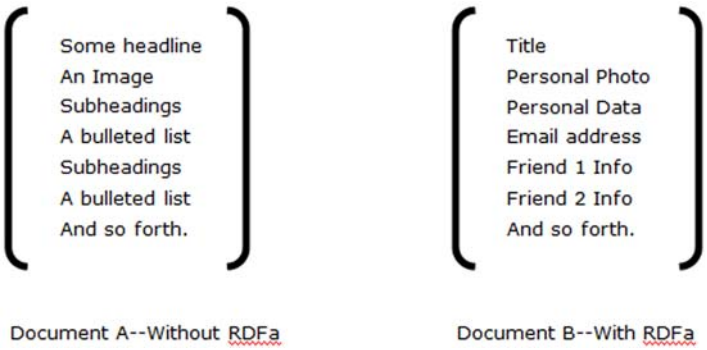


Figure 6.1 Machine interpretation of HTML documents versus human interpretation

Microsoft, Yandex, and Yahoo!. Finally, we'll demonstrate a similar outcome with the schema.org vocabularies. The embedded RDFa can be extracted and we'll illustrate searching the extracted RDFa using a SPARQL query.

Overall, our goal is to provide semantic meaning to your web content and enable the extraction of Linked Data. We use the FOAF vocabulary because it will be familiar from chapter 4. We use the GoodRelations vocabulary because of its significance to e-commerce, and we use schema.org because it's supported by a collaboration of three major search engines (Yahoo!, Bing, and Google). By embedding RDFa in your HTML documents, you enable search engines to provide more relevant search results and also allow for the incorporation of the content as Linked Data on the Web.

6.1 Enhancing HTML by embedding RDFa

Being digitally accessible isn't synonymous with being machine comprehendible. For instance, the cover of a publication may have a digitally accessible photo, but the significance of that photo is not machine understandable. But a barcode on that cover is machine consumable in that it enables a program to identify the object and likely to access its cost and track its purchase. Using RDFa on a web page serves an equivalent purpose to the barcode. It enables a search engine to identify the meaning of the digital data, making it structured data.

RDF provides a mechanism for expressing data and relationships. RDF in Attributes (RDFa) is a language that allows you to express RDF data within an HTML document. This enables your website to be both machine- and human-readable. HTML is a means of describing the desired visual appearance of your content. HTML doesn't differentiate between a book title and a job title. It can only differentiate the font displayed according to the author's direction. The human reader needs to interpret the information based on the context of the page and identify a book title as opposed to a job title. RDFa enables the author to embed structured data that will identify this differentiation. Authors can mark up human-consumable information for interpretation by browsers, search engines, and other programs. The RDFa-specific attributes don't affect the visual display of the HTML content and are ignored by the browser as it would any other attribute not recognizable as HTML.

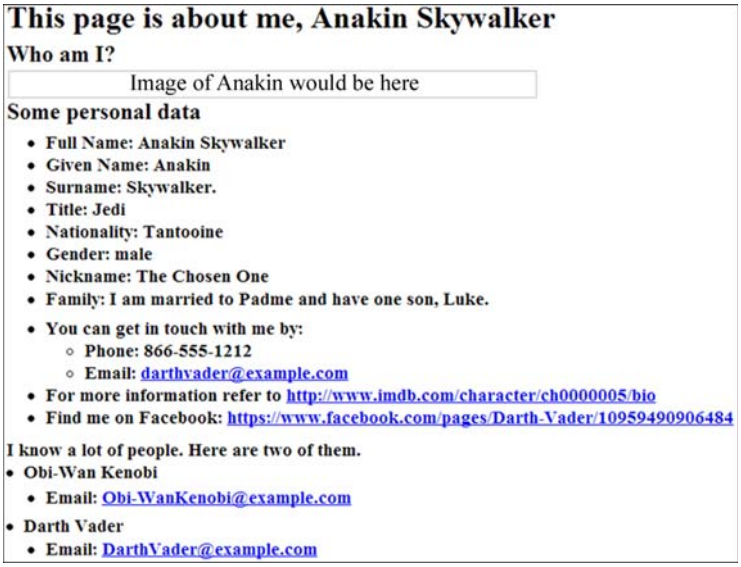


Figure 6.2 Web page produced by listing 6.1

We'll begin our application with a traditional basic HTML document about Anakin Skywalker, shown in listing 6.1. We'll mark up the HTML by embedding RDFa and explain what we're doing as we proceed. As an HTML page without RDFa, the browser interprets the content without regard to its semantic meaning. The page as displayed could contain any content, and the HTML elements affect its visual appearance. Hence, the content is simply as illustrated in Document A of figure 6.1.

As human readers (illustrated as Document B of figure 6.1), we recognize that the web page in figure 6.2 is about Anakin Skywalker. We recognize that it contains an image of Anakin, some of his personal information, and brief information about the people he knows. Our goal is to embed RDFa properties that would enable an automated interpretation of this page as a human reader would. The following listing contains the fundamental HTML description without any embedded RDFa.

Listing 6.1 An HTML description without embedded RDFa

```
<html>
<head>
<title>Anakin Skywalker</title>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type"
➥  />
</head>

<body>
<h1> This page is about me, Anakin Skywalker </h1>
<h2>Who am I?</h2>
<img
➥  src="http://www.starwars.com/img/explore/encyclopedia/
```

```
➥  characters/anakinskywalker_detail.png"
➥  alt="http://www.starwars.com/img/explore/encyclopedia/
➥  characters/anakinskywalker_detail.png">

<h2>
<p>I was born on the planet Tatooine.  I like to invent.
I invented my own droid, C-3PO, from salvaged parts.
My mother is Shmi and she says that I do not have a father.
I was trained as a Jedi knight by Obi-Wan Kenobi.
I am an excellent knight but I don't like authority figures.

While I was assigned to guard Padme, I fell in love with her.
She knew that I loved her and that I distrusted the
political process. I wished we had one strong leader. </p>

<p>As a Jedi Knight, I fought many battles for the Republic
and I rescued many captives. However, after a series
of such episodes, I was injured and succumbed to the
Dark Side.</p>
</h2>
<h2>

Some personal data
</h2>
<ul>
<h3>
<li>Full Name: Anakin Skywalker </li>

<li>Given Name: Anakin </li>

<li>Surname: Skywalker.</li>

<li>Title: Jedi </li>

<li>Nationality: Tatooine </li>

<li>Gender: male</li>

<li>Nickname:  The Chosen One </li>

<li>Family: I am married to Padme and have one son, Luke.</li>
</h3>
</ul>

<h3>
<ul>
<li> You can get in touch with me by: </li>
    <ul>
    <li>Phone:  866-555-1212</li>
    <li>Email:
➥   <a href="mailto:darthvader@example.com">
➥   darthvader@example.com</a></li>
    </ul>
<li> For more information refer to
➥   <a href= "http://www.imdb.com/character/ch0000005/bio">
```

```
➥  http://www.imdb.com/character/ch0000005/bio    </a> </li>
<li> Find me on Facebook:
➥  <a href= "https://www.facebook.com/pages/Darth-
➥  Vader/10959490906484">
➥  https://www.facebook.com/pages/Darth-Vader/10959490906484    </a> </li>
</ul>
</h3>

<h3>
I know a lot of people.  Here are two of them.
<li> Obi-Wan Kenobi</li>
    <ul>
    <li>Email: <a href="mailto:obiwankenobi@example.com">
➥  Obi-WanKenobi@example.com</a></li>
    </ul>
<li> Darth Vader</li>
    <ul>
    <li>Email:<a href="mailto:darthvader@example.com">
➥  DarthVader@example.com</a></li>
    </ul>
</h3>
</body>
</html>
```

6.1.1 RDFa markup using FOAF vocabulary

Now let's embed some RDFa into the HTML of listing 6.1. Our fully enhanced HTML document is contained in listing 6.2. Let's break down the additions to the basic HTML document from listing 6.1. As you enhance your HTML document with RDFa, you should periodically validate your efforts. You'll find an easy-to-use tool at http://www.w3.org/2012/pyRdfa/.

At the beginning of listing 6.2, you'll notice two statements that you need to support both HTML5 and RDFa:

```
<!DOCTYPE HTML>
<html version="HTML+RDFa 1.1" lang="en" >
```

In the remainder of the document, you embed the RDFa elements by applying them in conjunction with HTML tags. RDFa attributes allowed on all elements in the HTML5 content model are

- `vocab`
- `resource`
- `about`
- `datatype`
- `typeof`
- `prefix`
- `rel`
- `inlist`
- `property`
- `content`
- `rev`

All other attributes that RDFa may process, like `href` and `src`, are only allowed on the elements defined in the HTML5 specification.[1]

[1] HTML+RDFa 1.1, Support for RDFa in HTML4 and HTML5, W3C working draft, Sept. 11, 2012, http://www.w3.org/TR/2012/WD-rdfa-in-html-20120911/#extensions-to-the-html5-syntax.

The HTML `body` tag shown in the following snippet, extracted from listing 6.2, contains a `prefix` attribute. The `prefix` attribute serves the same purpose here as it does in Turtle documents. The various vocabularies listed in the `prefix` attribute of the `body` tag can be conveniently referred to throughout the body of the document using the associated shorthand prefixes defined.

```
<body id=me
prefix = "
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs: http://www.w3.org/2000/01/rdf-schema#
xsd: http://www.w3.org/2001/XMLSchema#
dc: http://purl.org/dc/elements/1.1/
foaf: http://xmlns.com/foaf/0.1/
rel: http://purl.org/vocab/relationship/
stars: http://www.starwars.com/explore/encyclopedia/characters/ "
>
```

As you further examine listing 6.2, you'll notice extensive use of the HTML `div` and `span` tags. These tags don't affect the visual appearance of the document and are primarily used as grouping indicators. The `span` and `div` elements are similar to `<div>a contained block</div>` that starts on a new line. `<span>some text</span>` is an inline separator that identifies the enclosed text as a single entity. The `typeof` attribute defines the enclosed entity as being an object of type `foaf:Person`.

Listing 6.2 HTML sample with RDFa markup from the FOAF vocabulary

```
<!DOCTYPE html>
<html version="HTML+RDFa 1.1" lang="en">
<head>
<title>Anakin Skywalker</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<base href= "http://rosemary.umw.edu/~marsha/starwars/foaf.ttl#" >

</head>

<body id=me
prefix = "
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs: http://www.w3.org/2000/01/rdf-schema#
xsd: http://www.w3.org/2001/XMLSchema#
dc: http://purl.org/dc/elements/1.1/
foaf: http://xmlns.com/foaf/0.1/
rel: http://purl.org/vocab/relationship/
stars: http://www.starwars.com/explore/encyclopedia/characters/ "
>
<div id="container"
about="http://rosemary.umw.edu/~marsha/starwars/foaf.ttl#me"
typeof="foaf:Person">
<h1> This page is about me, Anakin Skywalker </h1>

<h2>Who am I?</h2>
<img property="foaf:img" class="flr"
```

Statements alerting browser to RDFa and use of HTML5

Prefix statement

Defines the start of a block named container

Defines the enclosed entity as being an object of type foaf:Person

Identifies the image to display and identifies this object as being a foaf:img property

```
src="http://www.starwars.com/img/explore/encyclopedia/characters/
➥  anakinskywalker_detail.png" alt="http://www.starwars.com/img/explore/
     encyclopedia/characters/anakinskywalker_detail.png">

<h2>
<p>I was born on the planet Tatooine. I like to invent. I invented my own
     droid, C-3PO, from salvaged parts. My mother is Shmi and she says that I
     do not have a father.
I was trained as a Jedi knight by Obi-Wan Kenobi. I am an excellent knight
     but I don't like authority figures.

While I was assigned to guard Padme, I fell in love with her. She knew that I
     loved her and that I distrusted the political process. I wished we had
     one strong leader. </p>

<p>As a Jedi Knight, I fought many battles for the Republic and I rescued
     many captives. However, after a series of such episodes, I was injured
     and succumbed to the Dark Side.</p>
</h2>
<h2>

Some personal data
</h2>
<ul>
<h3>
<li>Full Name: <span property="foaf:name">Anakin Skywalker</span> </li>
<li>Given Name: <span property="foaf:givenname">Anakin</span> </li>
<li>Surname: <span property="foaf:family_name">Skywalker</span></li>
<li>Title: <span property="foaf:title">Jedi</span> </li>
<li>Nationality: Tatooine </li>
<li>Gender: <span property="foaf:gender">male</span></li>
<li>Nickname:  <span property="foaf:nick">The Chosen One</span></li>
<li>Family: I am married to <span property="foaf:knows rel:spouseOf">Padme</
     span> and have one son, <span property="foaf:knows">Luke Skywalker</
     span>.</li>
</h3>
</ul>

<h3>
<ul>
<li> You can get in touch with me by: </li>
     <ul>
     <div vocab="http://xmlns.com/foaf/0.1/">
     <li>Phone:  <span property="phone">866-555-1212</span></li>
     <li>Email: <span property="mbox_sha1sum"
➥       content="cc77937087f686e222bcf1194fb8c671d8591e00">
     <a href="mailto:darthvader@example.com">AnakinSkywalker</a>
➥       </span></li>
     </div>
     </ul>

<li> For more information refer to <a property="foaf:homepage" href= "http://
     www.imdb.com/character/ch0000005/bio">http://www.imdb.com/character/
     ch0000005/bio    </a> </li>
```

Defines Padme as someone known to Anakin Skywalker and clarifies that relationship as one of spouse using the Relationship vocabulary[2]

[2] "Relationship: A vocabulary for describing relationships between people," created by Ian Davis and Eric Vitiello Jr., http://purl.org/vocab/relationship.

```
<li> Find me on Facebook:   <a typeof="foaf:account" href= "https://
     www.facebook.com/pages/Darth-Vader/10959490906484"> https://
     www.facebook.com/pages/Darth-Vader/10959490906484    </a> </li>
</ul>
I know a lot of people.  Here are two of them.
<div rel="foaf:knows" typeof="foaf:Person">
<ul>
    <li>
    <a property="foaf:homepage" href="http://live.dbpedia.org/page/
➥       Obi-Wan_Kenobi" />
    <span property="foaf:name">Obi-Wan Kenobi</span>
    <ul>
       <li>
       Email: <span property="foaf:mbox_sha1sum"
➥          content="aadfbacb9de289977d85974fda32baff4b60ca86">
       <a href="mailto:obiwankenobi@example.com">Obi-Wan Kenobi</a>
       </li>
     </ul>
     </li>
</ul>
</div>

<div rel="foaf:knows" typeof="foaf:Person">
<ul>
    <li>
    <a property="foaf:homepage"
➥       href="http://www.imdb.com/character/ch0000005/bio" />
    <span property="foaf:name">Darth Vader</span>
    <ul>
       <li>
       Email: <span property="foaf:mbox_sha1sum"
➥          content="cc77937087f686e222bcf1194fb8c671d8591e00">
       <a href="mailto:darthvader@example.com">DarthVader</a>
       </li>
    </ul>
    </li>
</ul>
</div>
</h3>
</div>
</body>
</html>
```

6.1.2 Using the HTML span attribute with RDFa

The first use of the `<span></span>` tag in conjunction with the RDFa `property` attribute is in the bulleted items excerpted in listing 6.3. The `property` attribute identifies which class property is being defined. In the case of

```
<li>Full Name: <span property="foaf:name">Anakin Skywalker</span> </li>
```

it's defining Anakin Skywalker as a `foaf:name`. Hence the characters "Anakin Skywalker" are now more than just some text to be displayed but are associated with the meaning defined by a `foaf:name`.

Listing 6.3 Bulleted list excerpt

```
<li>Full Name: <span property="foaf:name">Anakin Skywalker</span> </li>
<li>Given Name: <span property="foaf:givenname">Anakin</span> </li>
<li>Surname: <span property="foaf:family_name">Skywalker</span></li>
<li>Title: <span property="foaf:title">Jedi</span> </li>
<li>Nationality: Tatooine </li>
<li>Gender: <span property="foaf:gender">male</span></li>
<li>Nickname:  <span property="foaf:nick">The Chosen One</span></li>
<li>Family: I am married to <span property="foaf:knows rel:spouseOf">Padme</
     span> and have
➥  one son, <span property="foaf:knows">Luke Skywalker</span>.</li>
```

NOTE The full RDFa 1.1 specification is at http://www.w3.org/TR/rdfa-core/.

6.1.3 Extracting Linked Data from a FOAF-enhanced HTML document

Entering the HTML document shown in listing 6.2 into the validator and RDFa 1.1 distiller (http://www.w3.org/2012/pyRdfa/) generates the Turtle content shown in the next listing. Although this isn't a necessary step in using RDFa, it illustrates two important points:

- The RDFa enhancements are extractable as Linked Data.
- The extracted RDF data can be saved in a separate file, published, and used as input to other applications, as illustrated in section 6.4.

Listing 6.4 Turtle generated from listing 6.2 RDFa-enhanced HTML

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdfa: <http://www.w3.org/ns/rdfa#> .
@prefix rel: <http://purl.org/vocab/relationship/> .

<http://rosemary.umw.edu/~marsha/starwars/foaf.ttl> rdfa:usesVocabulary foaf: .

<http://rosemary.umw.edu/~marsha/starwars/foaf.ttl#me> a foaf:Person;
    rel:spouseOf "Padme";
    foaf:family_name "Skywalker";
    foaf:gender "male";
    foaf:givenname "Anakin";
    foaf:homepage <http://www.imdb.com/character/ch0000005/bio>;
    foaf:img <http://www.starwars.com/img/explore/encyclopedia/characters/
➥   anakinskywalker_detail.png>;
    foaf:knows [ a foaf:Person;
            foaf:homepage <http://live.dbpedia.org/page/Obi-Wan_Kenobi>;
            foaf:mbox_sha1sum "aadfbacb9de289977d85974fda32baff4b60ca86";
            foaf:name "Obi-Wan Kenobi" ],
        "Padme",
        [ a foaf:Person;
            foaf:homepage <http://www.imdb.com/character/ch0000005/bio>;
            foaf:mbox_sha1sum "cc77937087f686e222bcf1194fb8c671d8591e00";
            foaf:name "Darth Vader" ],
        "Luke Skywalker";
    foaf:mbox_sha1sum "cc77937087f686e222bcf1194fb8c671d8591e00";
    foaf:name "Anakin Skywalker";
```

```
    foaf:nick "The Chosen One";
    foaf:phone "866-555-1212";
    foaf:title "Jedi" .

<https://www.facebook.com/pages/Darth-Vader/10959490906484> a foaf:account .
```

You'll notice that this output in the previous listing bears a close resemblance to the FOAF profiles that we developed in chapter 4.

You can use the Google Structured Data Testing Tool (http://www.google.com/webmasters/tools/richsnippets) to see the result of your efforts. Unfortunately, you're limited to 1500 characters.

This section illustrated how to use RDFa to enhance a typical HTML homepage to provide meaningful structure to the content that enables machine interpretation of the content. In general, RDFa enhancements improve SEO. In the following section we'll further illustrate how RDFa can be used to enhance business websites.

6.2 Embedding RDFa using the GoodRelations vocabulary

GoodRelations is the most widely used RDF vocabulary for e-commerce. It enables you to publish details of your products and services in a way that search engines, mobile applications, and browser extensions can utilize the information and improve your click-through rates. In this section, we use the GoodRelations vocabulary to enhance a web page that describes a product, the Sony Cyber-shot DSC-WX100 camera, thus giving that description more meaning and improving its SEO.

Search engines like Google and Yahoo! recognize GoodRelations data in web pages provided by more than 10,000 product vendors like Sears, Kmart, and Best Buy.

Martin Hepp,[3] professor of e-commerce at University of Bundeswehr München and inventor of the GoodRelations ontology, says that preliminary evidence shows that enhancing your web pages with RDFa will improve your click-through rate by 30%. This is consistent with the results reported by Jay Myers, lead web development engineer at bestbuy.com.[4]

6.2.1 An overview of the GoodRelations vocabulary

The GoodRelations website at http://www.heppnetz.de/projects/goodrelations/ contains complete information on the vocabulary and its use. The goals and an overview of the conceptual model of this vocabulary are published at http://wiki.goodrelations-vocabulary.org/Documentation/Conceptual_model. As described there, the purpose of GoodRelations is to enable you to define an object for e-commerce that's industry neutral, valid from raw materials through retail to after-sales services, and syntax neutral.

This is achieved by using just four entities for representing e-commerce scenarios:

- An agent (for example, a person or an organization)
- An object (for example, a camera, a house, a bicycle) or service (for example, a manicure)

[3] Personal homepage of Martin Hepp, professor at the chair of General Management and E-Business at Universität der Bundeswehr Munich, http://www.heppnetz.de/.

[4] Paul Miller, "SemTechBiz Keynote: Jay Myers discusses Linked Data at Best Buy," June 6, 2012, http://semanticweb.com/semtechbiz-keynote-jay-myers-discusses-linked-data-at-best-buy_b29622

- A promise (offer) to transfer some rights (ownership, temporary usage, a certain license) on the object or to provide the service for a certain compensation (for example, an amount of money), made by the agent and related to the object or service
- A location from which this offer is available

This Agent-Promise-Object Principle can be found across most industries and is the foundation of the generic power of GoodRelations. It allows you to use the same vocabulary for offering a camera as for a manicure service or for the disposal of used motorcycles.

The respective classes in GoodRelations are:

- `gr:BusinessEntity` for the agent; that is, the company or individual
- `gr:Offering` for an offer to sell, repair, or lease something or to express interest in such an offer
- `gr:ProductOrService` for the object or service
- `gr:Location` for a store or location from which the offer is available

In table 6.1, the first column lists the characteristics that you'd want to specify about a product. The second column has the GoodRelations term associated with each characteristic. Some properties are new to GoodRelations and others are reused from other vocabularies (for example, FOAF and RDF-data vocabularies). You'll see many of these applied in the Sony camera HTML page that we enhance with RDFa. We're including these tables here for your ready reference and to give you an idea of the kind of data that we'd want to enhance in support of e-commerce.

Table 6.1 Google-supported GoodRelations properties associated with products or services[a]

Product characteristic	GoodRelations property
name	`gr:name`
image	`foaf:depiction`
brand	`gr:hasManufacturer` (for the brand link) and `gr:BusinessEntity` for the manufacturer name
description	`gr:description`
review information	`v:hasReview` (from http://rdf.data-vocabulary.org/#)
review format	`v:Review-aggregate` (from http://rdf.data-vocabulary.org/#)

identifier	`gr:hasStockKeepingUnit`
	`gr:hasEAN_UCC-13`
	`gr:hasMPN`
	`gr:hasGTN`

a. Google Webmaster Tools, "Produce properties: GoodRelations and hProduct," May 27, 2013, http://support.google.com/webmasters/bin/answer.py?hl=en&answer=186036

Table 6.2 lists the characteristics of an offer and the associated terms in the GoodRelations vocabulary that you'd use in modeling these characteristics. The second column lists the associated term in GoodRelations and offers guidance on how to apply it. `foaf:page` is the only term not contained in the GoodRelations vocabulary.

Table 6.2 Google-supported GoodRelations properties associated with an offer[a]

Offer characteristic	GoodRelations property
price	Price information is enclosed in the `gr:hasPriceSpecification` tag. Use the `content` attribute of the child `gr:hasCurrencyValue` to specify the actual price (use only a decimal point as a separator).
priceRangeLow	Price information is enclosed in the `gr:hasPriceSpecification` tag. Use the `content` attribute of the child `gr:hasMinCurrencyValue` to specify the lowest price of the available range (use only a decimal point as a separator).
priceRangeHigh	Price information is enclosed in the `gr:hasPriceSpecification` tag. Use the `content` attribute of the child `gr:hasMaxCurrencyValue` to specify the highest price of the available range (use only a decimal point as a separator).
priceValidUntil	`gr:validThrough`
currency	Price information is enclosed in the `gr:hasPriceSpecification` tag. Use the child `gr:hasCurrency` to specify the actual currency.
seller	`gr:BusinessEntity`
condition	`gr:condition`
availability	Inventory level is enclosed in the `gr:hasInventoryLevel` tag. Use the child tag `gr:QuantitativeValue` to specify the quantity in stock. For example, an item is in stock if the value of the content attribute of the enclosed tag `gr:hasMinValue` is greater than 0. Listing 6.6 applies this property.
offerURI	`foaf:page`

identifier	`gr:hasStockKeepingUnit`
	`gr:hasEAN_UCC-13`
	`gr:hasMPN`
	`gr:hasGTN`

a. Google Webmaster Tools, "Product properties: GoodRelations and hProduct," May 27, 2013, http://support.google.com/webmasters/bin/answer.py?hl=en&answer=186036.

When a single product that has different offers (for example, the same pair of running shoes is offered by different merchants), an aggregate offer can be used. These properties and associated GoodRelations terms are listed in table 6.3. As you'd expect, many of these terms are also associated with an offer.

Table 6.3 Google-supported GoodRelations properties associated with an offer-aggregate[a]

Offer-aggregate characteristics	GoodRelations property
priceRangeLow	Price information is enclosed in the `gr:hasPriceSpecification` tag. Use the `content` attribute of the child `gr:hasMinCurrencyValue` to specify the lowest price of the available range.
priceRangeHigh	Price information is enclosed in the `gr:hasPriceSpecification` tag. Use the `content` attribute of the child `gr:hasMaxCurrencyValue` to specify the highest price of the available range.
currency	Price information is enclosed in the `gr:hasPriceSpecification` tag. Use the child `gr:hasCurrency` to specify the actual currency.
seller	`gr:BusinessEntity`
condition	`gr:condition`
availability	Inventory level is enclosed in the `gr:hasInventoryLevel` tag. Use the child tag `gr:QuantitativeValue` to specify the quantity in stock. For example, an item is in stock if the value of the content attribute of the enclosed tag `gr:hasMinValue` is greater than 0. See listing 6.6 for more details.
offerURI	`foaf:page`
identifier	`gr:hasStockKeepingUnit` `gr:hasEAN_UCC-13` `gr:hasMPN` `gr:hasGTN`

a. Google Webmaster Tools, "Product properties: GoodRelations and hProduct," May 27, 2013, http://support.google.com/webmasters/bin/answer.py?hl=en&answer=186036.

6.2.2 Enhancing HTML with RDFa using GoodRelations

As we did in section 6.1, we'll start with a basic HTML file, marking it up using RDFa and the GoodRelations vocabulary. As we mentioned earlier in this chapter, GoodRelations is an important vocabulary for e-commerce. A basic HTML version of a web page for the camera was previously added to our wish list in chapter 4. This HTML document is shown in the next listing. This description will be annotated with many of the properties described in tables 6.1 and 6.2 shortly.

Listing 6.5 Basic HTML without GoodRelations markup

```
<html>
<head>
<title>SONY Camera</title>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
</head>

<body>

<h2> Sony - Cyber-shot DSC-WX100 <BR>
  18.2-Megapixel Digital Camera - Black
```

```
</h2>
 <BR>
<img src="http://images.bestbuy.com/BestBuy_US/images/products/5430/
➥  5430135_sa.jpg" alt="http://http://images.bestbuy.com/BestBuy_US/images/
➥  products/5430/5430135_sa.jpg">
<BR>
Model: DSCWX100/B      SKU: 5430135 <BR>
Customer Reviews:  4.9 of 5 Stars(14 reviews)
<BR>
Best Buy
http://www.bestbuy.com
<BR>
Sale Price: $199.99<BR>
Regular Price: $219.99<BR>
In Stock <BR>

<h3>
  Product Description
  <ul>
  <li>10x optical/20x clear image zoom </li>
  <li>2.7" Clear Photo LCD display</li>
  <li>1080/60i HD video</li>
  <li>Optical image stabilization</li>
  </ul>
</h3>
<BR>
Sample Customer Reviews<BR>
<BR>
Impressive - by: ABCD, November 29, 2012 <BR>

At 4 ounces this is a wonder. With a bright view screen and tons of features,
     this camera can't be beat.
<BR>
5.0/5.0 Stars<BR>
<BR>
Nice Camera, easy to use, panoramic feature by: AbcdE, November 26, 2012 <BR>
Great for when you don't feel like dragging the SLR around. Panoramic feature
     and video quality are very good.<BR>
4.75/5.0 Stars<BR>
<BR>
</body>
</html>
```

Although RDFa supports the entire GoodRelations vocabulary,[5] we're electing to limit our markup to the Google-supported properties listed in table 6.1. We encourage you to generate rich snippets for your web page by using the tools provided by GoodRelations.[6] You should heed the additional recommendations from the developers of GoodRelations (http://wiki.goodrelations-vocabulary.org/Quickstart).

[5] "GoodRelations Language Reference, V1.0, Release Oct. 1, 2011, http://www.heppnetz.de/ontologies/goodrelations/v1.html.

[6] Google Webmaster Tools, "Product properties: GoodRelations and hProduct," May 27, 2013, http://support.google.com/webmasters/bin/answer.py?hl=en&answer=186036.

The following listing is an annotated version of the basic HTML shown in listing 6.5. This web page is for the Sony camera from our wish list in chapter 4. We selected the camera because it's a product often marketed online, and GoodRelations will enable us to annotate the sale price, the vendor, the manufacturer, and product reviews.

Listing 6.6 Sample listing 6.5 using GoodRelations

```
<!DOCTYPE html>
<html version="HTML+RDFa 1.1" lang="en">
<head>
<title>Illustrating RDFa and GoodRelations</title>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
<base href =
"http://rosemary.umw.edu/~marsha/other/sonyCameraRDFaGRversion3.html" />
</head>

<body id="camera"
prefix = "
review: http://purl.org/stuff/rev#
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs: http://www.w3.org/2000/01/rdf-schema#
xsd: http://www.w3.org/2001/XMLSchema#
foaf: http://xmlns.com/foaf/0.1/
rel: http://purl.org/vocab/relationship
v: http://rdf.data-vocabulary.org/# "
>

<!—Company related data—Put this on your main page -->
  <div typeof="gr:BusinessEntity" about="#company">
    <div property="gr:legalName" content="Linked Data Practitioner's
➥  Guide"></div>
    <div property="vcard:tel" content="540-555-1212"></div>
    <div rel="vcard:adr">
      <div typeof="vcard:Address">
        <div property="vcard:country-name" content="United States"></div>
        <div property="vcard:locality" content="Fredericksburg"></div>
        <div property="vcard:postal-code" content="22401"></div>
        <div property="vcard:street-address" content="1234 Main
➥  Street"></div>
      </div>
    </div>
    <div rel="foaf:page" resource=""></div>
  </div>

  <div typeof="gr:Offering" about="#offering">
    <div rev="gr:offers" resource="http://www.example.com/#company"></div>
    <div property="gr:name" content="Cyber-shot DSC-WX100"
➥  ml:lang="en"></div>
    <div property="gr:description" content="18.2-Megapixel
➥   Digital Camera - Black &lt;li&gt;10x optical/20x
➥  clear image zoom &lt;/li&gt; &lt;li&gt;2.7&quot; Clear Photo LCD
```

Generated using the URL in the footnote but modified to centralize all prefix declarations under body tag[7]

[7] Generated using http://www.ebusiness-unibw.org/tools/grsnippetgen/.

```
➥ display&lt;/li&gt; &lt;li&gt;1080/60i HD video&lt;
➥ /li&gt; &lt;li&gt;Optical image stabilization&lt;/li&gt;"
➥ xml:lang="en"></div>
    <div property="gr:hasEAN_UCC-13" content="0027242854031"
➥ datatype="xsd:string"></div>
    <div rel="foaf:depiction"
➥ resource="http://images.bestbuy.com/BestBuy_US/images/products
➥ /5430/5430135_sa.jpg"></div>
    <div rel="gr:hasPriceSpecification">
      <div typeof="gr:UnitPriceSpecification">
        <div property="gr:hasCurrency" content="USD"
➥ datatype="xsd:string"></div>
        <div property="gr:hasCurrencyValue" content="199.99"
➥ datatype="xsd:float"></div>
        <div property="gr:hasUnitOfMeasurement" content="C62"
➥ datatype="xsd:string"></div>
      </div>
    </div>

    <div rel="gr:hasBusinessFunction"
➥ resource="http://purl.org/goodrelations/v1#Sell"></div>
    <div rel="foaf:page" resource="http://www.example.com/dscwx100/"></div>
    <div rel="gr:includes">
      <div typeof="gr:SomeItems" about="#product">
        <div property="gr:category" content="ProductOrServiceModel"
➥ xml:lang="en"></div>
        <div property="gr:name" content="Cyber-shot DSC-WX100"
➥ xml:lang="en"></div>
        <div property="gr:description" content="18.2-Megapixel Digital
➥ Camera - Black &lt;li&gt;10x optical/20x clear image zoom &lt;/li&gt;
➥  &lt;li&gt;2.7&quot; Clear Photo LCD display&lt;/li&gt;
➥  &lt;li&gt;1080/60i HD video&lt;/li&gt; &lt;li&gt;Optical image
➥ stabilization&lt;/li&gt;" xml:lang="en"></div>
        <div property="gr:hasEAN_UCC-13" content="0027242854031"
➥ datatype="xsd:string"></div>
        <div rel="foaf:depiction"
➥ resource="http://images.bestbuy.com/BestBuy_US/images/products/
➥ 5430/5430135_sa.jpg"></div>
        <div rel="foaf:page"
➥ resource="http://www.example.com/dscwx100/"></div>
      </div>
    </div>
  </div>

<h2> Sony - Cyber-shot DSC-WX100 <BR>
 18.2-Megapixel Digital Camera - Black </h2>
 <BR>
 <img src="http://images.bestbuy.com/BestBuy_US/images/products/5430/
➥  5430135_sa.jpg" alt="http://http://images.bestbuy.com/BestBuy_US/images/
    products/
➥  5430/5430135_sa.jpg">
 <BR>

<span rel="v:hasReview">
  <span typeof="v:Review-aggregate" about="#review_data">
```

```
Customer Reviews:
<span property="v:average" datatype="xsd:string"> 4.9 </span>
of <span property="v:best">5.0</span> Stars (<span property="v:count"
     datatype="xsd:string">14 </span>reviews)
<BR>
  </span>
</span>
Best Buy <BR>
<div rel="foaf:page" resource="http://www.bestbuy.com"></div>
<BR>

<span rel="gr:hasPriceSpecification">
<span typeof="gr:UnitPriceSpecification">
Sale Price: $<span property="gr:hasCurrencyValue v:lowprice"
     datatype="xsd:float">199.99</span><BR>
Regular Price: $<span property="gr:hasCurrencyValue v:highprice"
     datatype="xsd:float">219.99</span><BR>
</span>
</span>
Availability: <div rel="gr:hasInventoryLevel">
       <div typeof="gr:QuantitativeValue">
         <div property="gr:hasMinValue" content="1" datatype="xsd:float">In-
     stock</div>
       </div>
  </div>
  <BR>

<h3>
Product Description
<ul>
<li>10x optical/20x clear image zoom </li>
<li>2.7" Clear Photo LCD display</li>
<li>1080/60i HD video</li>
<li>Optical image stabilization</li>
</ul>
</h3>
<BR>

Sample Customer Reviews<BR>
<BR>

<br />Product Reviews:
<div rel="review:hasReview v:hasReview">
<span typeof="v:Review-aggregate review:Review">
<br />Average:
<span property="review:rating v:average" datatype="xsd:float">4.5</span>, avg.:
<span property="review:minRating" datatype="xsd:integer">0</span>, max:
<span property="review:maxRating" datatype="xsd:integer">5</span> (count:
<span property="review:totalRatings v:votes"
datatype="xsd:integer">45</span>)<br />
</span>
</DIV>

<div rel="review:hasReview v:hasReview" typeof="v:Review">
Impressive - by: <span property="v:reviewer">ABCD</span>, <span
```

Additional properties that improve the accessibility of your data

In RDFa, in the absence of a resource attribute, the typeof attribute on the enclosing div implicitly sets the subject of the properties marked up within that div.

Sample of a review aggregate

```
➥  property="v:dtreviewed" content="2012-11-29">November 29, 2012
➥  </span><BR>
<span property="v:summary">At 4 ounces this is a wonder. with a bright view
➥  screen and tons of features this camera can't be beat </span>
<span property="v:value">5.0</span>of
<span property="v:best">5.0</span> Stars<BR>
</div>
<BR>
<BR>

<div rel="review:hasReview v:hasReview" typeof="v:Review">
Nice Camera, easy to use, panoramic feature by: <span property="v:reviewer">
➥  AbcdE</span>, <span property="v:dtreviewed" content="2012-11-26">November

➥  26, 2012 </span><BR>
<span property="v:summary">Great for when you don't feel like dragging the
➥  SLR around. Panoramic feature and video quality are very good.</span><BR>
<span property="v:value">4.75</span> of
<span property="v:best">5.0</span> Stars<BR>
</div>
<BR>
<div rel="gr:hasBusinessFunction"
resource="htt;://purl.org/goodrelations/v1#Sell"></div>
<div property="gr:hasEAN_UCC-13" content="0027242854031"
➥  datatype="xsd:string"></div>
<div rel="foaf:page" resource=""></div>
<div rev="gr:offers" resource="http://www.bestbuy.com"></div>
</div>
</body>
</html>
```

From GoodRelations website

As you can glean from examining listing 6.6, the GoodRelations vocabulary fulfilled its expectations. Every business-related item on the page is associated with its meaning. Martin Hepp recommends that developers follow the original Google patterns[8] for marking up their pages with the following additions. These additions will make your data understood by all RDFa-aware search engines, shopping comparison sites, and mobile services. The Google recommendations are for Google only. The additional items are as follows:

- Add "about" attributes for turning your key data elements into identifiable resources so you can refer to your offer data.
- Add "datatype" attributes for all literal values to fulfill valid RDF requirements.
- Add `alt="Product image"` to all images for XHTML compatibility.
- Add `foaf:page` link. Empty quotation marks are sufficient for this link if it doesn't exist.
- Add `gr:hasEAN_UCC-13` for the EAN/ISBN13 code. The UPC code can be easily translated to this format by appending a leading zero. This is useful for linking your offer to datasheets provided by their manufacturers.

[8] Google Webmaster Tools, "Product properties: GoodRelations and hProduct," May 27, 2013, http://support.google.com/webmasters/bin/answer.py?hl=en&answer=186036.

- Add the `gr:hasBusinessFunction` to make clear you're selling the item.
- Add `gr:offers` link to the company via the `rev` attribute. This can also be inserted on your main page.

Hence, in compliance with Martin Hepp's recommendations, listing 6.6 includes the following code:

```
<div rel="gr:hasBusinessFunction"
resource="http://purl.org/goodrelations/v1#Sell"></div>
<div property="gr:hasEAN_UCC-13" content="0027242854031"
datatype="xsd:string"></div>
<div rel="foaf:page" resource=""></div>
<div rev="gr:offers" resource="http://www.bestbuy.com"></div>
```

Notice the data type associated with each of the literal currency values.

```
Sale Price: $<span property="gr:hasCurrencyValue v:lowprice"
datatype="xsd:float">199.99</span><BR>
Regular Price: $<span property="gr:hasCurrencyValue v:highprice"
datatype="xsd:float">219.99</span><BR>
```

6.2.3 A closer look at selections of RDFa GoodRelations

Breaking down the document section by section, the start of the document contains these statements:

```
<!DOCTYPE html>
<html version="HTML+RDFa 1.1" lang="en">
<head>
<title>Illustrating RDFa and GoodRelations</title>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
<base href = "http://www.example.com/sampleProduct/">
</head>
```

These statements identify the document type as HTML5 and set the `html version` attribute to HTML+RDFa1.1. These settings will ensure that most clients extract the RDF and recognize its existence. The purpose of the `<base href…>` statement is to provide an absolute URI for reference, and it should contain the URI of the company web reference.

> **NOTE** Use the actual URI associated with the publication of the product's document as the expression within the quotes.

Including the `prefix` statement in the `<body…>` statement, shown in the next listing, establishes access to the schema at each of these locations for the entire `body` section of the document and establishes each vocabulary within this namespace.

Listing 6.7 Excerpt showing centralization of prefix information

```
<body id="camera"
prefix = "
review: http://purl.org/stuff/rev#
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs: http://www.w3.org/2000/01/rdf-schema#
```

```
xsd: http://www.w3.org/2001/XMLSchema#
foaf: http://xmlns.com/foaf/0.1/
rel: http://purl.org/vocab/relationship
v: http://rdf.data-vocabulary.org/# "
>
```

Sections of code in listing 6.8 were generated using the GoodRelations snippet generator at http://www.ebusiness-unibw.org/tools/grsnippetgen/. We modified the output from the snippet generator. The namespace declarations were removed to simplify the example and replace the `xmlns` statements with their HTML5 equivalents. We also consolidated and centralized all the prefix declarations.

The excerpt highlighted in this listing describes the company web page, the legal name of the company, and its country, city, ZIP code, and physical address.

Listing 6.8 Excerpt of company information

```
  <div typeof="gr:BusinessEntity" about="#company">
    <div property="gr:legalName"
➥   content="Linked Data Practitioner's Guide"></div>
    <div property="vcard:tel" content="540-555-1212"></div>
    <div rel="vcard:adr">
      <div typeof="vcard:Address">
        <div property="vcard:country-name" content="United States"></div>
        <div property="vcard:locality" content="Fredericksburg"></div>
        <div property="vcard:postal-code" content="22401"></div>
        <div property="vcard:street-address"
➥   content="1234 Main Street"></div>
      </div>
    </div>
    <div rel="foaf:page" resource=""></div>
  </div>
```

The next listing, also generated using the online form at http://www.ebusiness-unibw.org/tools/grsnippetgen/, was modified to reflect the existing presence of the prefix declarations. It annotates the individual product information. It includes the product name, description, digital image of the product, UPC, seller, and cost.

Listing 6.9 Excerpt of product information

```
   <div typeof="gr:Offering" about="#offering">
    <div rev="gr:offers" resource="http://www.example.com/#company"></div>
<div property="gr:name" content="Cyber-shot DSC-WX100"
➥   xml:lang="en"></div>
    <div property="gr:description" content="18.2-Megapixel Digital Camera
➥    - Black &lt;li&gt;10x optical/20x clear image zoom &lt;/li&gt;
➥   &lt;li&gt;2.7&quot; Clear Photo LCD display&lt;/li&gt;
➥   &lt;li&gt;1080/60i HD video&lt;/li&gt; &lt;li&gt;Optical image
➥   stabilization&lt;/li&gt;" xml:lang="en"></div>
    <div property="gr:hasEAN_UCC-13" content="0027242854031"
➥   datatype="xsd:string"></div>
    <div rel="foaf:depiction"
resource="http://images.bestbuy.com/BestBuy_US/images/products/5430/
```

```
➥ 5430135_sa.jpg"></div>
    <div rel="gr:hasPriceSpecification">
      <div typeof="gr:UnitPriceSpecification">
        <div property="gr:hasCurrency"
➥ content="USD" datatype="xsd:string"></div>
        <div property="gr:hasCurrencyValue"
➥ content="199.99" datatype="xsd:float"></div>
        <div property="gr:hasUnitOfMeasurement"
➥ content="C62" datatype="xsd:string"></div>
      </div>
    </div>
```

Listing 6.10 highlights the annotation of an individual product review. You'll notice that the entire review is wrapped in a `<div rel=…>` to establish a relationship between our Sony camera and this review. Listing 6.6 contains two individual reviews and one aggregate review. All three are similarly annotated. Because the aggregate review represents a composite review, you'll notice that some of the represented properties are different from those in the next listing.

Listing 6.10 Excerpt showing annotation of an individual product review

```
<div rel="review:hasReview v:hasReview" typeof="v:Review">
Nice Camera, easy to use, panoramic feature by: <span
➥ property="v:reviewer"> AbcdE</span>, <span property="v:dtreviewed"
➥ content="2012-11-26">November 26, 2012 </span><BR>
<span property="v:summary">Great for when you don't feel like dragging
➥ the SLR around. Panoramic feature and video quality are very
➥ good.</span><BR>
<span property="v:value">4.75</span> of
<span property="v:best">5.0</span> Stars<BR>
</div>
```

6.2.4 Extracting Linked Data from GoodRelations-enhanced HTML document

As we illustrated in section 6.1, entering the HTML document shown in listing 6.6 into the validator and RDF 1.1 distiller (http://www.w3.org/2012/pyRdfa/) generates the Turtle content shown in the next listing. As we mentioned earlier in this chapter, this output can be retained and published. It can be used as input to other applications. In section 6.4, we'll illustrate mining this Linked Data using SPARQL.

Listing 6.11 Turtle statements derived from listing 6.6

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix gr: <http://purl.org/goodrelations/v1#> .
@prefix rev: <http://purl.org/stuff/rev#> .
@prefix v: <http://rdf.data-vocabulary.org/#> .
@prefix vcard: <http://www.w3.org/2006/vcard/ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://rosemary.umw.edu/~marsha/other/sonyCameraRDFaGRversion3.html
➥ #company> a gr:BusinessEntity;
    gr:legalName "Linked Data Practitioner's Guide"@en;
```

```
    vcard:adr [ a vcard:Address;
            vcard:country-name "United States"@en;
            vcard:locality "Fredericksburg"@en;
            vcard:postal-code "22401"@en;
            vcard:street-address "1234 Main Street"@en ];
    vcard:tel "540-555-1212"@en;
    foaf:page
➥   <http://rosemary.umw.edu/~marsha/other/sonyCameraRDFaGRversion3.html> .

<http://www.example.com/#company> gr:offers
➥   <http://rosemary.umw.edu/~marsha/other/sonyCameraRDFaGRversion3.html
➥   #offering> .

<http://rosemary.umw.edu/~marsha/other/sonyCameraRDFaGRversion3.html
➥   #offering> a gr:Offering;
    gr:description "18.2-Megapixel Digital Camera - Black
➥   <li>10x optical/20x clear image zoom </li>
➥   <li>2.7\" Clear Photo LCD display</li>
➥   <li>1080/60i HD video</li>
➥   <li>Optical image stabilization</li>"@en;
    gr:hasBusinessFunction gr:Sell;
    gr:hasEAN_UCC-13 "0027242854031"^^xsd:string;
    gr:hasPriceSpecification [ a gr:UnitPriceSpecification;
            gr:hasCurrency "USD"^^xsd:string;
            gr:hasCurrencyValue "199.99"^^xsd:float;
            gr:hasUnitOfMeasurement "C62"^^xsd:string ];
    gr:includes
➥   <http://rosemary.umw.edu/~marsha/other/sonyCameraRDFaGRversion3.html
➥   #product>;
    gr:name "Cyber-shot DSC-WX100"@en;
    foaf:depiction
➥   <http://images.bestbuy.com/BestBuy_US/images/products/5430/
     5430135_sa.jpg>;
    foaf:page <http://www.example.com/dscwx100/> .

<http://rosemary.umw.edu/~marsha/other/sonyCameraRDFaGRversion3.html
➥   #product> a gr:SomeItems;
    gr:category "ProductOrServiceModel"@en;
   gr:description "18.2-Megapixel Digital Camera - Black
➥   <li>10x optical/20x clear image zoom </li>
➥   <li>2.7\" Clear Photo LCD display</li>
➥   <li>1080/60i HD video</li> <li>Optical image stabilization</li>"@en;
    gr:hasEAN_UCC-13 "0027242854031"^^xsd:string;
    gr:name "Cyber-shot DSC-WX100"@en;
    foaf:depiction
➥   <http://images.bestbuy.com/BestBuy_US/images/products/5430/
     5430135_sa.jpg>;
    foaf:page <http://www.example.com/dscwx100/> .

<http://rosemary.umw.edu/~marsha/other/sonyCameraRDFaGRversion3.html#
➥   review_data> a v:Review-aggregate;
    v:average " 4.9"^^xsd:string;
    v:best "5.0"@en;
    v:count "14"^^xsd:string .
```

```
<http://www.bestbuy.com> gr:offers
➥  <http://rosemary.umw.edu/~marsha/other/sonyCameraRDFaGRversion3.html> .

<http://rosemary.umw.edu/~marsha/other/
➥  sonyCameraRDFaGRversion3.html> gr:hasBusinessFunction
➥   <http://rosemary.umw.edu/~marsha/other/#Sell>;
➥  gr:hasEAN_UCC-13 "0027242854031"^^xsd:string;
    gr:hasInventoryLevel [ a gr:QuantitativeValue;
            gr:hasMinValue "1"^^xsd:float ];
    gr:hasPriceSpecification [ a gr:UnitPriceSpecification;
            gr:hasCurrencyValue "199.99"^^xsd:float,
                "219.99"^^xsd:float;
            v:highprice "219.99"^^xsd:float;
            v:lowprice "199.99"^^xsd:float ];
    rev:hasReview _:_7a58d778-3981-4844-96e6-71b32fe1b439,
        _:_8d4ade4e-7085-4104-9a4e-d6131abe5853,
        _:_c6154cb1-03bf-4ba9-b237-67e0984a7a86;
    v:hasReview _:_7a58d778-3981-4844-96e6-71b32fe1b439,
        _:_8d4ade4e-7085-4104-9a4e-d6131abe5853,
        _:_c6154cb1-03bf-4ba9-b237-67e0984a7a86,
        <http://rosemary.umw.edu/~marsha/other/sonyCameraRDFaGRversion3.html
➥   #review_data>;
    foaf:page <http://rosemary.umw.edu/~marsha/other/
     sonyCameraRDFaGRversion3.html>,
        <http://www.bestbuy.com> .

_:_7a58d778-3981-4844-96e6-71b32fe1b439 a v:Review;
    v:best "5.0"@en;
    v:dtreviewed "2012-11-26"@en;
    v:reviewer " AbcdE"@en;
    v:summary "Great for when you don't feel like dragging the SLR
➥   around. Panoramic feature and video quality are very good."@en;
    v:value "4.75"@en .

_:_8d4ade4e-7085-4104-9a4e-d6131abe5853 a rev:Review,
        v:Review-aggregate;
    rev:maxRating 5;
    rev:minRating 0;
    rev:rating "4.5"^^xsd:float;
    rev:totalRatings 45;
    v:average "4.5"^^xsd:float;
    v:votes 45 .

_:_c6154cb1-03bf-4ba9-b237-67e0984a7a86 a v:Review;
    v:best "5.0"@en;
    v:dtreviewed "2012-11-29"@en;
    v:reviewer "ABCD"@en;
    v:summary "At 4 ounces this is a wonder. with a bright view screen and
tons of features this camera can't be beat "@en;
    v:value "5.0"@en .
```

_:_7a58d778-3981-4844-96e6-71b32fe1b439 represents a blank node. Refer to chapter 2 for a more complete explanation.

_:_8d4ade4e-7085-4104-9a4e-d6131abe5853 represents a blank node. Refer to chapter 2 for a more complete explanation.

_:_c6154cb1-03bf-4ba9-b237-67e0984a7a86 represents a blank node. Refer to chapter 2 for a more complete explanation.

In this section we've embedded RDFa using the GoodRelations vocabulary. This specialized vocabulary will enable you to embed product, service, and company information in your web pages. This additional information improves SEO and click-through

rates. Stay tuned; we understand that GoodRelations is in the process of being integrated into schema.org.

6.3 Embedding RDFa using the schema.org vocabulary

Schema.org is a collaborative initiative by three major search engines: Yahoo!, Bing, and Google. Its purpose is to create and support a common set of schema for structured data markup on web pages and to provide a common means for webmasters to mark up their pages so that the search results are improved and human users have a more satisfying experience. We'll follow a progression similar to what we did in section 6.2. We'll take a brief look at the schema.org vocabulary and apply it by embedding RDFa into the same basic HTML page that describes our Sony camera.

6.3.1 An overview of schema.org

The designers of schema.org provided a single common vocabulary and markup syntax (Microdata[9]) that's supported by the major search engines. This approach enables webmasters to use a single syntax and avoid tradeoffs based on which markup type is supported by which search engine. As you can see in table 6.4, schema.org supports a broad collection of object types and isn't limited to e-commerce terminology.

Table 6.4 Commonly used schema.org object types by category

Parent type	Subtypes
`Creative works`	`CreativeWork, Article, Blog, Book, Comment, Diet, ExercisePlan, ItemList, Map, Movie, MusicPlaylist, MusicRecording, Painting, Photograph, Recipe, Review, Sculpture, SoftwareApplication, TVEpisode, TVSeason, TVSeries, WebPage, WebPageElement`
`MediaObject` (Embedded non-text objects)	`AudioObject, ImageObject, MusicVideoObject, VideoObject`
`Event`	`BusinessEvent, ChildrensEvent, ComedyEvent, DanceEvent, EducationEvent, Festival, FoodEvent, LiteraryEvent, MusicEvent, SaleEvent, SocialEvent, SportsEvent, TheaterEvent, UserInteraction, VisualArtsEvent`
`Organization`	`Corporation, EducationalOrganization, GovernmentOrganization, LocalBusiness, NGO, PerformingGroup, SportsTeam`
`Intangible`	`Audience, Enumeration, JobPosting, Language, Offer, Quantity, Rating, StructuredValue`
`Person`	

[9] Defining the HTML microdata mechanism, HTML Microdata W3C Working Draft May 24, 2011, http://dev.w3.org/html5/md-LC/.

Table 6.4 Commonly used schema.org object types by category *(continued)*

Parent type	Subtypes
Place	LocalBusiness, Restaurant, AdministrativeArea, CivicStructure, Landform, LandmarksOrHistoricalBuildings, LocalBusiness, Residence, TouristAttraction
Product	
Primitive Types	Boolean, Date, DateTime, Number, Text, Time

NOTE The schema.org specification is accessible from http://schema.org/docs/schemas.html.

Unlike RDF, schema.org was not designed to

- Provide resource description for purposes other than discovery
- Publish data not displayed on web pages
- Facilitate machine-to-machine communication
- Support other ontologies outside of those agreed on by the partners of schema.org

Subsequent feedback from the web community encouraged the developers of schema.org to accept and adopt RDFa Lite as an alternative syntax to encode schema.org terms. Schema.org members are search engines, which really care about scalability, thus making the use of RDFa Lite strongly preferred. The difference is that RDFa 1.1 is a complete syntax for RDF (and can thus express anything that RDF can). RDFa Lite consists of only five simple attributes: `vocab`, `typeof`, `property`, `resource`, and `prefix`. One of the convenient features about RDFa 1.1 Lite and RDFa 1.1 is that a number of commonly used prefixes (http://www.w3.org/2011/rdfa-context/rdfa-1.1.html) are predefined. Therefore, you can omit declaring them and just use them, but the W3C recommended style is to include the prefix declarations.

The full specification for RDFa 1.1 Lite is at http://www.w3.org/TR/rdfa-lite/. RDFa 1.1 Lite is a subset of RDFa and consists of just five attributes that are used together with HTML tags to enable web developers to mark up their sites with Linked Data. We'll briefly discuss these attributes and then develop an example illustrating how RDFa 1.1 Lite works with HTML to enable meaningful markup of web pages.

Table 6.5 Properties of the schema.org `Product` class

Property	Type	Description
aggregateRating	AggregateRating	Based on a collection of reviews or ratings, this is the overall rating of the item.
brand	Organization	The brand of the product, for example, Sony, Minolta.
description	Text	A brief narrative about the item.

Table 6.5 Properties of the schema.org `Product` class *(continued)*

Property	Type	Description
`image`	`URI`	The URI of an image of the item.
`manufacturer`	`Organization`	The manufacturer of this product.
`model`	`Text`	The model identifier for this product.
`name`	`Text`	The name of the product.
`offers`	`Offer`	An offer to sell this product.
`productID`	`Text`	The product identifier, such as a UPC code.
`review`	`Review`	A review of this product.
`URI`	`URI`	The URI of this product.

As we mentioned in section 6.1, HTML without RDFa annotations to the browser looks like this:

Headline
Some image
More text
Bulleted list
More text

Adding RDFa markup will add meaning to all this text and enable the search engines to interpret this content as a human reader would. The search engines will "see" this:

Product name
Product image
Product description
Product rating
More product description
Consumer reviews

Obviously, this is a more meaningful web page.

6.3.2 Enhancing HTML with RDFa Lite using schema.org

Using schema.org with RDFa 1.1 Lite is similar to annotating a web page with RDFa except you limit your terms to those defined in the Lite subset. This example was intentionally restricted to just those terms defined in schema.org. Although doing so was certainly not required, we thought it the best approach given the purpose of our illustration.

As we illustrated in section 6.1, we'll start with a plain HTML file designed around an item from our chapter 4 wish list, shown in listing 6.12, without semantic annotations and then enhance that web document with schema.org using RDFa 1.1 Lite notation. The enhanced document is in listing 6.13. As we illustrated in section 6.2, the web page contains product information for the Sony Cyber-shot DSC-WX100 that we added to our wish list in chapter 4.

In prior examples, we used multiple vocabularies and specified which ones using prefix statements. In this case, we need only designate a single vocabulary, schema.org, which will be our default vocabulary, for example:

```
<div vocab = "http://schema.org/" typeof= "Product">
Some other text
</div>
```

We've actually defined two attributes: the default vocabulary that we're going to use and the type of the object we'll be describing. We also need to identify the properties of the object that we'll be describing. Looking more closely at the schema.org `Product` class summarized in table 6.5, we find that it has a number of properties/attributes that we'll use to enhance our Sony camera web page.

Listing 6.12 Basic HTML

```
<html>
<head>
<title>SONY Camera</title>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
</head>

<body>

<h2> Sony - Cyber-shot DSC-WX100 <BR>
  18.2-Megapixel Digital Camera - Black
</h2>
 <BR>
<img
➥  src="http://images.bestbuy.com/BestBuy_US/images/products/5430/
➥  5430135_sa.jpg"
➥  alt="http://http://images.bestbuy.com/BestBuy_US/images/
➥  products/5430/5430135_sa.jpg">
<BR>
Model: DSCWX100/B      SKU: 5430135 <BR>
Customer Reviews:  4.9 of 5 Stars(14 reviews)
<BR>
Best Buy
http://www.bestbuy.com
<BR>
Sale Price: $199.99<BR>
Regular Price: $219.99<BR>
In Stock <BR>

<h3>
  Product Description
```

```
  <ul>
  <li>10x optical/20x clear image zoom </li>
  <li>2.7" Clear Photo LCD display</li>
  <li>1080/60i HD video</li>
  <li>Optical image stabilization</li>
  </ul>
</h3>
<BR>
Sample Customer Reviews<BR>
<BR>
Impressive - by: ABCD, November 29, 2012 <BR>

At 4 ounces this is a wonder. With a bright view screen and tons of features,
     this camera can't be beat
<BR>
5.0/5.0 Stars<BR>
<BR>
Nice Camera, easy to use, panoramic feature by: AbcdE, November 26, 2012 <BR>
Great for when you don't feel like dragging the SLR around. Panoramic feature
     and video quality are very good.<BR>
4.75/5.0 Stars<BR>
<BR>
</body>
</html>
```

6.3.3 A closer look at selections of RDFa Lite using schema.org

As you examine listing 6.13, you'll notice extensive use of the `<span></span>` and `<div></div>` HTML tags with their property attributes. The values associated with those properties are from table 6.5, the properties of the schema.org `Product` class. One difference is the use of the `typeof` attribute when the property is of a non-basic type. For example,

```
<div property="offers" typeof="AggregateOffer">
```

designates the encapsulated property as type `offers` and that an item classified as `offers` is of `typeof AggregateOffer`.

Listing 6.13 HTML sample with RDFa markup from the schema.org vocabulary

```
<!DOCTYPE html>
<html version="HTML+RDFa 1.1" lang="en">
<head>
<title>Illustrating RDFa 1.1 Lite and schema.org</title>

<meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
<base href = "http://www.example.com/sampleProduct"/>
</head>

<body vocab="http://schema.org/">
<div typeof="Product">
 <h2 > <span property="brand" typeof="Organization">Sony </span>
 <span property="name"> Cyber-shot DSC-WX100 </span>
 <span property="description">18.2-Megapixel Digital Camera - Black
➥  </span></h2>
```

Product description

```
 <BR>
 <img property="image" src="http://images.bestbuy.com/BestBuy_US/images/
     products/5430/5430135_sa.jpg" alt="http://images.bestbuy.com/BestBuy_US/
     images/products/5430/5430135_sa.jpg">
 <BR>
Model: <span property="model">DSCWX100/B </span>   SKU: <span
➥  property="productID">5430135 </span>
<div property="aggregateRating" typeof="AggregateRating">
Customer Reviews:  <span property="ratingValue">4.9</span> of
<span property="bestRating">5.0</span> Stars (<span property="ratingCount">14
➥  </span>reviews)
<BR>
</div>
Best Buy <BR>
<span property="URI">http://www.bestbuy.com </span>
<BR>
<div property="offers" typeof="AggregateOffer">
Sale Price: $<span property="lowPrice">199.99</span><BR>
Regular Price: $<span property="highPrice">219.99</span><BR>
In Stock <BR>
</div>
<h3>
Product Description
<div property="description">
<ul>
<li>10x optical/20x clear image zoom </li>
<li>2.7" Clear Photo LCD display</li>
<li>1080/60i HD video</li>
<li>Optical image stabilization</li>
</ul>
</div>
</h3>
<BR>
Sample Customer Reviews<BR>
<BR>
<div property="review" typeOf="Review">
Impressive - by: <span property="author">ABCD</span>,
<span   property="datePublished" content="2012-11-29">November 29, 2012
➥  </span><BR>
At 4 ounces this is a wonder. with a bright view screen and tons of
➥  features this camera can't be beat
<span property="ratingValue">5.0</span> of
<span property="bestRating">5.0</span> Stars<BR>
</div>
<BR>
<BR>
<div property="review" typeOf="Review">
Nice Camera, easy to use, panoramic feature by: <span
➥  property="author">AbcdE</span>, <span property="datePublished"
➥  content="2012-11-26">November 26, 2012 </span><BR>
Great for when you don't feel like dragging the SLR around. Panoramic feature
➥  and video quality are very good.<BR>
<span property="ratingValue">4.75</span> of
<span property="bestRating">5.0</span> Stars<BR>
```

Review aggregate

Single person review

```
</div>
<BR>
</div>
</body>
</html>
```

6.3.4 Extracting Linked Data from a schema.org enhanced HTML document

The resulting Turtle obtained from entering the HTML document shown in listing 6.13 into the validator and RDF 1.1 distiller (http://www.w3.org/2012/pyRdfa/) is illustrated in the following listing. As we mentioned previously, this output can be retained and published. It can be used as input to other applications. In section 6.4, we'll illustrate mining similar Linked Data using SPARQL.

Listing 6.14 Resulting Turtle from HTML annotated with schema.org vocabulary

```
@prefix rdfa: <http://www.w3.org/ns/rdfa#> .
@prefix schema: <http://schema.org/> .

<http://www.example.com/sampleProduct> rdfa:usesVocabulary schema: .

[] a schema:Product;
    schema:aggregateRating [ a schema:AggregateRating;
            schema:bestRating "5.0"@en;
            schema:ratingCount "14"@en;
            schema:ratingValue "4.9"@en ];
    schema:brand [ a schema:Organization ];
    schema:description """

10x optical/20x clear image zoom
2.7" Clear Photo LCD display
1080/60i HD video
Optical image stabilization

"""@en,
        "18.2-Megapixel Digital Camera - Black "@en;
    schema:highPrice "219.99"@en;
    schema:image <http://images.bestbuy.com/BestBuy_US/images/products/5430/
     5430135_sa.jpg>;
    schema:lowPrice "199.99"@en;
    schema:model "DSCWX100/B "@en;
    schema:name " Cyber-shot DSC-WX100 "@en;
    schema:offers """
Sale Price: $199.99
Regular Price: $219.99
In Stock
"""@en;
    schema:productID "5430135 "@en;
    schema:review [ a schema:Review;
            schema:author "AbcdE"@en;
            schema:bestRating "5.0"@en;
            schema:datePublished "2012-11-26"@en;
            schema:ratingValue "4.75"@en ],
        [ a schema:Review;
            schema:author "ABCD"@en;
```

```
            schema:bestRating "5.0"@en;
            schema:datePublished "2012-11-29"@en;
            schema:ratingValue "5.0"@en ];
    schema:URI "http://www.bestbuy.com "@en .
```

NOTE [ ] represents a blank node. Refer to chapter 2 for additional explanation.

Annotating your data with the schema.org vocabulary and RDFa 1.1 Lite will provide a more satisfying result from SEO, expose your information to more consumers, and contribute to the community of publically accessible published Linked Data.

6.4 How do you choose between using schema.org or GoodRelations?

As a web developer, how do you choose between GoodRelations and schema.org? This question isn't easy to answer and may not have a one-size-fits-all response. When schema.org was introduced, web authors who wanted to improve their SEO and target the major search engines—Google, Microsoft, Yahoo!, and Yandex—would select schema.org and annotate their web pages using the microdata syntax. But now that schema.org supports RDFa 1.1 Lite, web authors have more options.

The W3C, the developers of RDFa, took the feedback they received from Google, Microsoft, and Yahoo! very seriously and proposed RDFa Lite in response to their concerns. The schema.org community was concerned about the complexity of RDFa. Manu Sporny, the chair of the W3C RDF Web Applications Working Group, addressed this: "With RDFa 1.1, our focus has been on simplifying the language for Web authors. In some cases, we've simplified the RDFa markup to only require two HTML attributes to markup some of the schema.org examples. In most cases you only need three HTML attributes to express a concept that will enhance your search ranking… it's as simple as that."[10] With the support of RDFa 1.1 Lite, a developer can annotate a page using the schema.org vocabulary and if needed supplement those annotations with vocabulary from other sources, including GoodRelations.

The advantage of using RDFa 1.1 Lite with either schema.org or GoodRelations is that a web author can apply other features of the full RDFa standard because RDFa Lite is a subset of RDFa. For the time being, features outside of RDFa Lite would be ignored by the schema.org community. Consequently, at the moment, the choice between GoodRelations or schema.org seems to be one of best fit for the task at hand. Examine the vocabularies and determine which one addresses your needs. Because they can be used in conjunction with each other, we think that the choice is one of personal preference.

6.5 Extracting RDFa from HTML and applying SPARQL

RDF extracted from the RDFa-enhanced HTML files can be queried using SPARQL. The following listing illustrates a SPARQL query selecting the individual reviews and yields the results. The source being queried is a file copy of the Turtle extracted by the validator

[10] Eric Franzon, "Schema.org announces intent to support RDFa Lite!" November 11, 2011, http://semanticweb.com/breaking-schema-org-announces-intent-to-support-rdfa-lite_b24623#more-24623.

and RDF 1.1 distiller (http://www.w3.org/2012/pyRdfa/). This output is shown after the listing.

Listing 6.15 Sample SPARQL query to select reviews

```
prefix v: <http://rdf.data-vocabulary.org/#>
prefix rev: <http://purl.org/stuff/rev#>

SELECT ?date ?summary ?value
FROM <http://rosemary.umw.edu/~marsha/other/sonyCameraGRversion3.ttl>

WHERE {
  ?review a v:Review
    ; v:dtreviewed ?date
    ; v:summary ?summary
    ; v:value ?value .
} LIMIT 10
```

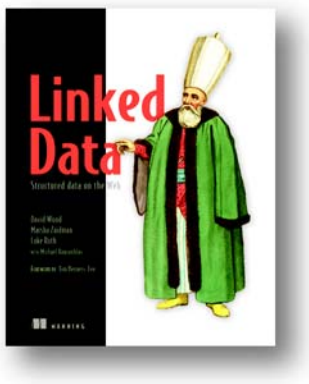
This query selects a review that identifies a date, narrative summary, and a rating value and extracts the date of the review, its narrative summary, and the numeric value of the rating. As we'd expect based on chapter 5, we have three columns of output representing the date, narrative summary, and numeric value. Following are the results of this query showing the two reviews selected.

```
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| date            | summary
                                                              | value      |
===============================================================================
===============================================================================
| "2012-11-29"@en | "At 4 ounces this is a wonder. With a bright view screen
    and
 tons of features, this camera can't be beat "@en          | "5.0 "@en  |
| "2012-11-26"@en | "Great for when you don't feel like dragging the SLR
    around.
 Panoramic feature and video quality are very good."@en | "4.75 "@en |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
```

In this example, we applied a SPARQL query to RDFa extracted from our HTML page describing our Sony camera. The query selected the individual person reviews and extracted the date, summary, and star value contained in each review. Our HTML page contained two reviews.

6.6 Summary

We've illustrated three techniques for enhancing your web pages with RDFa 1.1. Section 6.1 illustrated how to use RDFa and the FOAF vocabulary to add structured data and semantic meaning to your HTML content. Section 6.2 illustrated how to annotate your HTML content with RDFa and the GoodRelations business-oriented vocabulary to add structured data and hence semantic meaning to a product description and sales information. Section 6.3 accomplished the same goal but is limited to RDFa 1.1 Lite because it's supported by schema.org.



The current Web is mostly a collection of linked documents useful for human consumption. The evolving Web includes data collections that may be identified and linked so that they can be consumed by automated processes. The W3C approach to this is Linked Data and it is already used by Google, Facebook, IBM, Oracle, and government agencies worldwide.

Linked Data presents practical techniques for using Linked Data on the Web via familiar tools like JavaScript and Python. You'll work step-by-step through examples of increasing complexity as you explore foundational concepts such as HTTP URIs, the Resource Description Framework (RDF), and the SPARQL query language. Then you'll use various Linked Data document formats to create powerful Web applications and mashups.

What's inside

- Finding and consuming Linked Data
- Using Linked Data in your applications
- Building Linked Data applications using standard Web techniques

Written to be immediately useful to Web developers, this book requires no previous exposure to Linked Data or Semantic Web technologies.

The Share layer

The Web of Things is largely based on the idea of Things pushing data to the web, where more intelligence and big data techniques can be applied, for example, to help us manage our health or optimize our energy consumption. But this can only happen in a large-scale way if some of the data can be efficiently—and securely—shared across services. This is the responsibility of the Share layer: once Things are accessible and findable on the web, how do we share them (and the data they generate) in an efficient and secure manner over the web.

In *Building the Web of Things*, we look into applying fine-grained sharing mechanisms on top of web APIs. We also look at delegated web authentication mechanisms and integrate OAuth[1] to our Things' APIs. Finally, we look into implementing the Social Web of Things by leveraging social networks to share Things and their resources.

All of this helps to share Things, but sharing can't happen if there aren't enough security safeguards implemented in the first place! We believe that putting Things on the web will happen, but it has to happen in a secure manner: you might want your Things to be accessible and findable on the web but you surely don't want anyone to be able to control and monitor your Things! Luckily enough, the web offers one of the best compromises between usability and security and benefits from constant security improvements. In the following chapter, "Security" from *Express in Action*, you'll learn how to secure the Node.js servers that we teach you to deploy on your Things in *Building the Web of Things*.

[1] See http://oauth.net/

Chapter 10 from *Express in Action*
by Evan M. Hahn

Security

This chapter covers

- Keeping your Express code bug-free, using tools and testing
- Dealing with attacks; knowing how they work and how to prevent them
- Handling the inevitable server crash
- Auditing your third-party code

In chapter 8, I told you that I had three favorite chapters. The first was chapter 3, where I discussed the foundations of Express in an attempt to give you a solid understanding of the framework. The second favorite was chapter 8, where your applications used databases to become more real. Welcome to my final favorite: the chapter about security.

I probably don't have to tell you that computer security is important, and it's becoming more so by the day. You've surely seen news headlines about data breaches, cyberwarfare, and hacktivism. As our world moves more and more into the digital sphere, our digital security becomes more and more important.

Keeping your Express applications secure should (hopefully) be important—who *wants* to be hacked? In this chapter, we'll discuss ways your applications could be subverted and how to defend yourself.

This chapter doesn't have as much of a singular flow as the others. You'll find yourself exploring a topic and then jumping to another, and although there may be some similarities, most of these attacks are relatively disparate.

10.1 The security mindset

Famous security technologist Bruce Schneier describes something that he calls the *security mindset*:

> Uncle Milton Industries has been selling ant farms to children since 1956. Some years ago, I remember opening one up with a friend. There were no ants included in the box. Instead, there was a card that you filled in with your address, and the company would mail you some ants. My friend expressed surprise that you could get ants sent to you in the mail.
>
> I replied: "What's really interesting is that these people will send a tube of live ants to anyone you tell them to."
>
> Security requires a particular mindset. Security professionals—at least the good ones—see the world differently. They can't walk into a store without noticing how they might shoplift. They can't use a computer without wondering about the security vulnerabilities. They can't vote without trying to figure out how to vote twice. They just can't help it.
>
> "The Security Mindset" by Bruce Schneier, at https://www.schneier.com/blog/archives/2008/03/the_security_mi_1.html

Bruce Schneier isn't advocating that you should steal things and break the law. He's suggesting that the best way to secure yourself is to *think* like an attacker—how could someone subvert a system? How could someone abuse what they're given? If you can think like an attacker and seek out loopholes in your own code, then you can figure out how to close those holes and make your application more secure.

This chapter can't possibly cover every security vulnerability out there. Between the time I write this and the time you read this, there will likely be a new attack vector that *could* affect your Express applications. Thinking like an attacker will help you defend your applications against the endless onslaught of possible security flaws.

Just because I'm not going through *every* security vulnerability doesn't mean I won't go through the common ones. Read on!

10.2 Keeping your code as bug-free as possible

At this point in your programming career, you've likely realized that most bugs are bad and that you should take measures to prevent them. It should come as no surprise that many bugs can cause security vulnerabilities. For example, if a certain kind of user input can crash your application, a hacker could simply flood your

servers with those requests and bring the service down for everyone. You definitely don't want that!

There are numerous methods to keep your Express applications bug-free and therefore less susceptible to attacks. In this section, I won't cover the general principles for keeping your software bug-free, but here are a few to keep in mind:

- *Testing is terribly important.* We discussed testing in the previous chapter.
- *Code reviews can be quite helpful.* More eyes on the code almost certainly means fewer bugs.
- *Don't reinvent the wheel.* If someone has made a library that does what you want, you should probably use the library, but make sure it is well-tested and reliable!
- *Stick to good coding practices. We'll go over Express-* and JavaScript-specific issues, but you should make sure your code is well-architected and clean.

We'll talk about Express specifics in this section, but the principles just mentioned are hugely helpful in preventing bugs and therefore in preventing security issues.

10.2.1 Enforcing good JavaScript with JSHint

At some point in your JavaScript life, you've probably heard of *JavaScript: The Good Parts* (O'Reilly Media, 2008). If you haven't, it's a famous book by Douglas Crockford, the inventor of JSON (or the *discoverer*, as he calls it). It carves out a subset of the language that's deemed *good*, and the rest is discouraged.

For example, Crockford discourages the use of the double-equals operator (`==`) and instead recommends sticking to the triple-equals operator (`===`). The double-equals operator does type coercion, which can get complicated and can introduce bugs, whereas the triple-equals operator works pretty much how you'd expect.

In addition, a number of common pitfalls befall JavaScript developers that aren't necessarily the language's fault. To name a few: missing semicolons, forgetting the `var` statement, and misspelling variable names.

If there were a tool that enforced good coding style *and* a tool that helped you fix errors, would you use them? What if they were *just one tool*? I'll stop you before your imagination runs too wild: there's a tool called JSHint (http://jshint.com/).

JSHint looks at your code and points out what it calls suspicious use. It's not *technically* incorrect to use the double-equals operator or to forget `var`, but it's likely to be an error.

You'll install JSHint globally with `npm install jshint -g`. Now, if you type `jshint myfile.js`, JSHint will look at your code and alert you to any suspicious usage or bugs. The file in the following listing is an example.

Listing 10.1 A JavaScript file with a bug

```
function square(n) {
  var result n * n;          <-- = sign is missing.
  return result;
}
square(5);
```

Notice that the second line has an error: it's missing an equals sign. If you run JSHint on this file (with `jshint myfile.js`), you'll see the following output:

```
myfile.js: line 2, col 13, Missing semicolon.
myfile.js: line 3, col 18, Expected an assignment or function call and instead saw an
      expression.

2 errors
```

If you see this, you'll know that something's wrong! You can go back and add the equals sign, and then JSHint will stop complaining.

In my opinion, JSHint works best when integrated with your editor of choice. Visit the JSHint download page at http://jshint.com/install/ for a list of editor integrations. Figure 10.1 shows JSHint integrated with the Sublime Text editor. Now, you'll see the errors before you even run the code!

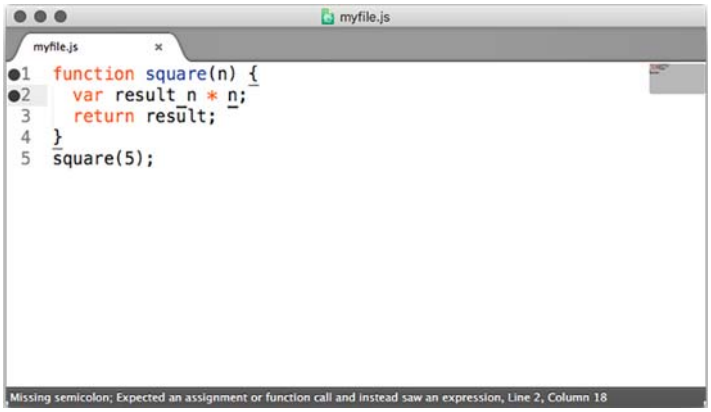


Figure 10.1 JSHint integration in the Sublime Text editor. Notice the error on the left side of the window and the message at the bottom in the status bar.

JSHint has saved me a *ton* of time when working with JavaScript and has fixed countless bugs. I know some of those bugs have been security holes.

10.2.2 Halting after errors happen in callbacks

Callbacks are a pretty important part of Node. Every middleware and route in Express uses them, not to mention … well, nearly everything else! Unfortunately, people make a few mistakes with callbacks, and these can create bugs.

See if you can spot the error in this code:

```
fs.readFile("myfile.txt", function(err, data) {
  if (err)  {
    console.error(err);
  }
  console.log(data);
});
```

In this code, you're reading a file and outputting its contents with `console.log` if everything works. But if it *doesn't* work for some reason, you output the error and then continue on to try to output the file's data.

If there's an error, you should be halting execution. For example:

```
fs.readFile("myfile.txt", function(err, data) {
  if (err)  {
    console.error(err);
    throw err;
  }
  console.log(data);
});
```

It's usually important to *stop* if there's any kind of error. You don't want to be dealing with errant results—this can cause your server to have buggy behavior.

10.2.3 Perilous parsing of query strings

It's very common for websites to have query strings. For example, almost every search engine you've ever used features a query string of some sort. A search for "crockford backflip video" might look something like this:

```
http://mysearchengine.com/search?q=crockford+backflip+video
```

In Express, you can grab the query by using `req.query`, as shown in the next listing.

Listing 10.2 Grabbing req.query (note: contains bugs!)

```
app.get("/search", function(req, res) {
  var search = req.query.q.replace(/\+/g, " ");
  // … do something with the search …
});
```

Contains the string "crockford backflip video"

This is all well and good, unless the input isn't exactly as you expect. For example, if a user visits the /search route with no query named `q`, then you'd be calling `.replace` on an undefined variable! This can cause errors.

You'll always want to make sure that your users are giving you the data you expect, and if they aren't, you'll need to do *something* about it. One simple option is to provide a default case, so if they don't give anything, assume the query is empty. See the next listing as an example.

Listing 10.3 Don't assume your queries exist (note: still contains bugs!)

```
app.get("/search", function(req, res) {
  var search = req.query.q || "";
  var terms = search.split("+");
  // … do something with the terms …
});
```

Adds a default value if req.query.q is undefined

This fixes one important bug: if you're expecting a query string that isn't there, you won't have undefined variables.

But there's another important gotcha with Express's parsing of query strings: they can also be of the wrong type (but still be defined)!

If a user visits /search?q=abc, then `req.query.q` will be a string. It'll still be a string if they visit /search?q=abc&name=douglas. But if they specify the `q` variable twice, like this

```
/search?q=abc&q=xyz
```

then `req.query.q` will be the array `["abc", "xyz"]`. Now, if you try to call `.replace` on it, it'll fail again because that method isn't defined on arrays. Oh, no!

Personally, I think that this is a design flaw of Express. This behavior should be allowed, but I don't think that it should be enabled by default. Until they change it (and I'm not sure they have plans to), you'll need to assume that your queries could be arrays.

To solve this problem (and others), I wrote the `arraywrap` package (available at https://www.npmjs.org/package/arraywrap). It's a very small module; the whole thing is only 19 lines of code. It's a function that takes one argument. If the argument isn't already an array, it wraps it in an array. If the argument *is* an array, it returns the argument because it is already an array.

You can install it with `npm install arraywrap --save` and then you can use it to coerce *all* of your query strings to arrays, as shown in the following listing.

Listing 10.4 Don't assume your queries aren't arrays

```
var arrayWrap = require("arraywrap");

// …

app.get("/search", function(req, res) {
  var search = arrayWrap(req.query.q || "");
  var terms = search[0].split("+");
  // … do something with the terms …
});
```

Note the changed line.

Now, if someone gives you more queries than you expect, you just take the first one and ignore the rest. This still works if someone gives you one query argument or *no* query argument. Alternatively, you could detect if the query was an array and do something different there.

This brings us to a big point of the chapter: *never trust user input*. Assume that every route will be broken in some way.

10.3 Protecting your users

Governments have had their sites defaced; Twitter had a kind of tweet virus; bank account information has been stolen. Even products that aren't dealing with particularly sensitive data can still have passwords leaked—Sony and Adobe have been caught up in such scandals. If your site has users, you'll want to be responsible and protect them. There are a number of things you can do to protect your users from harm, and we'll look at those in this section.

10.3.1 Using HTTPS

In short, use HTTPS instead of HTTP. It helps protect your users against all kinds of attacks. Trust me—you want it!

There are two pieces of Express middleware that you'll want to use with HTTPS. One will force your users to use HTTPS and the other will keep them there.

FORCE USERS TO HTTPS

The first middleware we'll look at is express-enforces-ssl. As the name suggests, it enforces SSL (HTTPS). Basically, if the request is over HTTPS, it continues on to the rest of your middleware and routes. If not, it redirects to the HTTPS version.

To use this module, you'll need to do two things.

1. Enable the "trust proxy" setting. Most of the time, when you deploy your applications, your server isn't *directly* connecting to the client. If you're deployed to the Heroku cloud platform (as you'll explore in chapter 11), Heroku servers sit between you and the client. To tell Express about this, you need to enable the "trust proxy" setting.
2. Call the middleware.

Make sure you `npm install express-enforces-ssl`, and then run the code in the following listing.

Listing 10.5 Enforcing HTTPS in Express

```
var enforceSSL = require("express-enforces-ssl");
// …
app.enable("trust proxy");
app.use(enforceSSL());
```

There's not much more to this module, but you can see more at https://github.com/aredo/express-enforces-ssl.

KEEP USERS ON HTTPS

Once your users are on HTTPS, you'll want to tell them to avoid going back to HTTP. New browsers support a feature called HTTP Strict Transport Security (HSTS). It's a simple HTTP header that tells browsers to stay on HTTPS for a period of time.

If you want to keep your users on HTTPS for one year (approximately 31,536,000 seconds), you'd set the following header:

```
Strict-Transport-Security: max-age=31536000
```

There are approximately 31,536,000 seconds in a year.

You can also enable support for subdomains. If you own slime.biz, you'll probably want to enable HSTS for cool.slime.biz.

To set this header, you'll use Helmet (https://github.com/helmetjs/helmet), a module for setting helpful HTTP security headers in your Express applications. As you'll see throughout the chapter, it has various headers it can set. We'll start with its HSTS functionality.

First, as always, `npm install helmet` in whatever project you're working on. I'd also recommend installing the ms module, which translates human-readable strings (like `"2 days"`) into 172,800,000 milliseconds. Now you can use the middleware, as shown in the next listing.

Listing 10.6 Using Helmet's HSTS middleware

```
var helmet = require("helmet");
var ms = require("ms");
// …
app.use(helmet.hsts({
  maxAge: ms("1 year"),
  includeSubdomains: true
}));
```

Now, HSTS will be set on every request.

WHY CAN'T I JUST USE HSTS? This header is only effective if your users are *already* on HTTPS, which is why you need `express-enforces-ssl`.

10.3.2 Preventing cross-site scripting attacks

I probably shouldn't say this, but there are a lot of ways you could steal my money. You could beat me up and rob me, you could threaten me, or you could pick my pocket. If you were a hacker, you could also hack into my bank and wire a bunch of my money to you (and of all the options listed, this is the one I most prefer).

If you could get control of my browser, even if you didn't know my password, you could still get my money. You could wait for me to log in and then take control of my browser. You'd tell my browser to go to the "wire money" page on my bank and take a large sum of money. If you were clever, you could hide it so that I'd never even know it happened (until, of course, all of my money was gone).

But how would you get control of my browser? Perhaps the most popular way would be through use of a cross-site scripting (XSS) attack.

Imagine that, on my bank's homepage, I can see a list of my contacts and their names, as shown in figure 10.2.

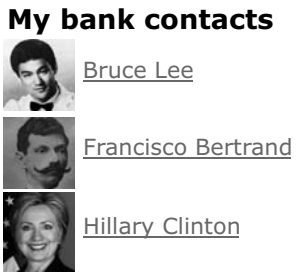


Figure 10.2 A fictional list of my bank contacts

Users have control over their names. Bruce Lee can go into his settings and change his name to Bruce Springsteen if he wants to. But what if he changed his name to this:

```
Bruce Lee<script>transferMoney(1000000, "bruce-
     lee's-account");</script>
```

The list of contacts would still show up the same, but now my web browser will also execute the code inside the `<script>` tag! Presumably, this will transfer a million dollars to Bruce Lee's account, and I'll never be the wiser. Bruce Lee could also add `<script src="http://brucelee.biz/hacker.js"></script>` to his name. This script could send data (like login information, for example) to brucelee.biz.

There's one big way to prevent XSS: never blindly trust user input.

ESCAPING USER INPUT

When you have user input, it's almost always possible that they'll enter something malicious. In the previous example, you could set your name to contain `<script>` tags, causing XSS issues. You can sanitize or escape user input, so that when you put it into your HTML, you aren't doing anything unexpected.

Depending on where you're putting the user input, you'll sanitize things differently. As a general principle, you'll want to sanitize things as much as you can and always keep the context in mind.

If you're putting user content inside HTML tags, for example, you'll want to make sure that it can't define any HTML tags. You'll want this kind of string

```
Hello, <script src="http://evil.com/hack.js"></script>world.
```

to become something like this:

```
Hello, &lt;script src="http://evil.com/hack.js"&gt;&lt;/script&gt;world.
```

By doing that, the `<script>` tags will be rendered useless.

This kind of escaping (and more) is handled by most templating engines for you. In EJS, simply use the default `<%= myString %>` and *don't* use the `<%- userString %>`. In Pug, this escaping is done by default. Unless you're certain that you don't want to sanitize something, make sure to use the safe version whenever you're dealing with user strings.

If you *know* that the user should be entering a URL, you'll want to do more than escaping; you'll want to do your best to validate that something is a URL. You'll also want to call the built-in `encodeURI` function on a URL to make sure it's safe.

If you're putting something inside an HTML attribute (like the `href` attribute of a link), you'll want to make sure your users can't insert quotation marks, for example. Unfortunately, there isn't a one-size-fits-all solution for sanitizing user input; the way you sanitize depends on the context. But you should *always* sanitize user input as much as you can.

You can also escape the input before you ever put it into your database. In the examples just used, we're showing how to sanitize things whenever we're displaying them. But if you know that your users should enter homepages on their user profiles, it's also useful to sanitize that before you ever store it in the database. If I enter "hello, world" as my homepage, the server should give an error. If I enter http://evanhahn.com as my homepage, that should be allowed and put into the database. This can have security benefits *and* UI benefits.

Mitigating XSS with HTTP headers

There's one other way to help mitigate XSS, but it's quite small, and that's through the use of HTTP headers. Once again, we'll break out Helmet.

There's a simple security header called `X-XSS-Protection`. It can't protect against all kinds of XSS, but it can protect against what's called reflected XSS. The best example of reflected XSS is on an insecure search engine. On every search engine, when you do a search, your query appears on the screen (usually at the top). If you search for "candy," the word *candy* will appear at the top, and it'll be part of the URL:

```
https://mysearchengine.biz/search?query=candy
```

Now imagine you're searching `"<script src="http://evil.com/hack.js"></script>"`. The URL might look something like this:

```
https://mysearchengine.biz/search?query=<script%20src="http://evil.com/
hack.js"></script>
```

Now, if this search engine puts that query into the HTML of the page, you've injected a script into the page! If I send this URL to you and you click the link, I can take control and do malicious things.

The first step against this attack is to *sanitize the user's input*. After that, you can set the `X-XSS-Protection` header to keep some browsers from running that script should you make a mistake. In Helmet, it's just one line:

```
app.use(helmet.xssFilter());
```

Helmet also lets you set another header called `Content-Security-Policy`. Frankly, `Content-Security-Policy` could be its own chapter. Check out the HTML5 Rocks guide at www.html5rocks.com/en/tutorials/security/content-security-policy/ for more information, and once you understand it, use Helmet's csp middleware.

Neither of these Helmet headers is anywhere near as important as sanitizing user input, so do that first.

10.3.3 Cross-site request forgery (CSRF) prevention

Imagine that I'm logged into my bank. You *want* me to transfer a million dollars into your account, but you aren't logged in as me. (Another challenge: I don't have a million dollars.) How can you get me to send you the money?

THE ATTACK

On the bank site, there's a "transfer money" form. On this form, I type the amount of money and the recipient of the money and then hit Send. Behind the scenes, a POST request is being made to a URL. The bank will make sure my cookies are correct, and if they are, it'll wire the money.

You can make the POST request with the amount and the recipient, but you don't know my cookie and you can't guess it; it's a long string of characters. So what if you could make *me* do the POST request? You'd do this with cross-site request forgery (CSRF and sometimes XSRF).

To pull off this CSRF attack, you'll basically have me submit a form without knowing it. Imagine that you've made a form like the one in the next listing.

Listing 10.7 A first draft of a hacker form

```
<h1>Transfer money</h1>
<form method="post" action="https://mybank.biz/transfermoney">
  <input name="recipient" value="YourUsername" type="text">
  <input name="amount" value="1000000" type="number">
  <input type="submit">
</form>
```

Let's say that you put this in an HTML file on a page *you* controlled; maybe it's hacker.com/stealmoney.html. You could email me and say, "Click here to see some photos of my cat!" If I clicked on it, I'd see something like figure 10.3:

And if I saw that, I'd get suspicious. I wouldn't click Submit and I'd close the window. But you can use JavaScript to automatically submit the form, as shown here.

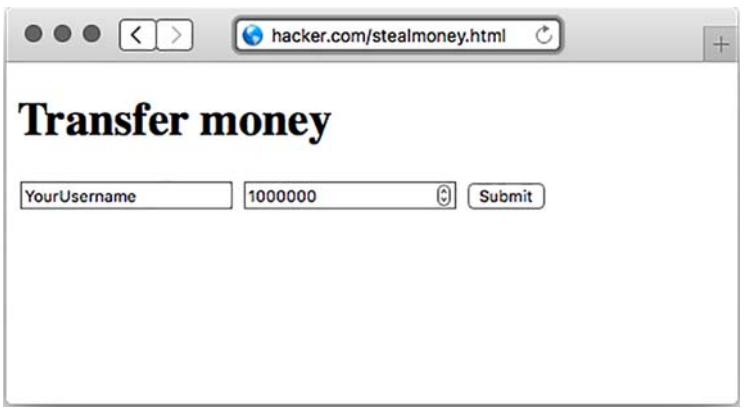


Figure 10.3 A suspicious-looking page that could steal my money

Listing 10.8 Automatically submitting the form

```
<form method="post" action="https://mybank.biz/transfermoney">
  <!-- … -->
</form>

<script>
var formElement = document.querySelector("form");
formElement.submit();
</script>
```

If I get sent to *this* page, the form will immediately submit and I'll be sent to my bank, to a page that says, "Congratulations, you've just transferred a million dollars." I'll probably panic and call my bank, and the authorities can likely sort something out.

But this is progress—you're now sending money to yourself. I won't show it here, but you can completely hide this from the victim. First, you make an `<iframe>` on your page. You can then use the form's `target` attribute, so that when the form submits, it submits *inside* the `iframe`, rather than on the whole page. If you make this `iframe` small or invisible (easy with CSS!), then I'll never know I was hacked until I suddenly had a million fewer dollars.

My bank needs to protect against this. But how?

Overview of protecting against CSRF

My bank already checks cookies to make sure that I am who I say I am. A hacker can't perform CSRF attacks without getting me to do *something*. But once the bank knows it's me, how does it know that I meant to do something and wasn't being tricked into doing something?

My bank decides this: if a user is submitting a POST request to mybank.biz/transfermoney, they aren't just doing that out of the blue. Before doing that POST, the user will be on a page that's asking where they want to transfer their money—perhaps the URL is mybank.biz/transfermoney_form.

So when the bank sends the HTML for mybank.biz/transfermoney_form, it's going to add a hidden element to the form: a completely random, unguessable string called a token. The form might now look like the code in the next listing.

Listing 10.9 Adding CSRF protections

```
<h1>Transfer money</h1>
<form method="post" action="https://mybank.biz/transfermoney">
  <input name="_csrf" type="hidden"
   ➥ value="1dmkTNkhePMTB0DlGLhm">
  <input name="recipient" value="YourUsername" type="text">
  <input name="amount" value="1000000" type="number">
  <input type="submit">
</form>
```

Value of the CSRF token will be different for every user, often every time

You've probably used thousands of CSRF tokens while browsing the web, but you haven't seen them because they are hidden from you. (You'll see CSRF tokens if you're like me and you enjoy viewing the HTML source of pages.)

Now, when the user submits the form and sends the POST request, the bank will make sure that the CSRF token sent is the same as the one the user just received. If it is, the bank can be pretty sure that the user just came from the bank's website and therefore intended to send the money. If it's not, the user might be being tricked—don't send the money.

In short, you need to do two things:

1. Create a random CSRF token every time you're asking users for data.
2. Validate that random token every time you deal with that data.

PROTECTING AGAINST CSRF IN EXPRESS

The Express team has a simple middleware that does those two tasks: csurf (https://github.com/expressjs/csurf). The csurf middleware does two things:

- *It adds a method to the request object called req.csrfToken.* You'll send this token whenever you send a form, for example.
- *If the request is anything other than a GET, it looks for a parameter called _csrf to validate the request, creating an error if it's invalid.* (Technically, it also skips HEAD and OPTIONS requests, but those are much less common. There are also a few other places the middleware will search for CSRF tokens; consult the documentation for more.)

To install this middleware, run `npm install csurf --save`.

The `csurf` middleware depends on some kind of session middleware and middleware to parse request bodies. If you need CSRF protections, you probably have some notion of users, which means that you're probably already using these, but `express-session` and `body-parser` do the job. Make sure you're using those before you use csurf. If you need an example, you can check out chapter 8's code for app.js or look at the CSRF example app at https://github.com/EvanHahn/Express.js-in-Action-code/blob/master/Chapter_10/csrf-example/app.js.

To use the middleware, simply `require` and `use` it. Once you've used the middleware, you can grab the token when rendering a view, like in the following listing.

Listing 10.10 Getting the CSRF token

```
var csrf = require("csurf");

// …

app.use(csrf());

app.get("/", function(req, res) {
  res.render("myview", {
    csrfToken: req.csrfToken()
  });
});
```

Include a body parser and session middleware before this.

Now, inside a view, you'll output the `csrfToken` variable into a hidden input called `_csrf`. It might look like the code in the next listing in an EJS template.

Listing 10.11 Showing the CSRF token in a form

```
<form method="post" action="/submit">
  <input name="_csrf" value="<%= csrfToken %>" type="hidden">
  <! -- … -->
</form>
```

And that's all. Once you've added the CSRF token to your forms, the csurf middleware will take care of the rest.

It's not required, but you'll probably want to have some kind of handler for failed CSRF. You can define an error middleware that checks for a CSRF error, as shown in the following listing.

Listing 10.12 Handling CSRF errors

```
// …

app.use(function(err, req, res, next) {
  if (err.code !== "EBADCSRFTOKEN") {
    next(err);
    return;
  }
  res.status(403);
  res.send("CSRF error.");
});

// …
```

Skips this handler if it's not a CSRF error

Error code 403 is "Forbidden."

This error handler will return `"CSRF error"` if there's, well, a CSRF error. You might want to customize this error page, and you might also want this to send you a message—someone's trying to hack one of your users!

You can place this error handler wherever in your error stack you'd like. If you want it to be the first error you catch, put it first. If you want it to be last, you can put it last.

10.4 Keeping your dependencies safe

Any Express application will depend on at least one third-party module: Express. If the rest of this book has shown you anything, it's that you'll be depending on *lots* of third-party modules. This has the huge advantage that you don't have to write a lot of boilerplate code, but it does come with one cost: you're putting your trust in these modules. What if the module creates a security problem?

There are three big ways that you can keep your dependencies safe:

- Audit the code yourself
- Make sure you're on the latest versions
- Check against the Node Security Project

10.4.1 Auditing the code

It might sound a bit crazy, but you can often easily audit the code of your dependencies. Although some modules like Express have a relatively large surface area, many of the modules you'll install are only a few lines, and you can understand them quickly. It's a fantastic way to learn, too.

Just as you might look through your own code for bugs or errors, you can look through other people's code for bugs and errors. If you spot them, you can avoid the module. If you're feeling generous, you can submit patches because these packages are all open source.

If you've already installed the module, you can find its source code in your node_modules directory. You can almost always find modules on GitHub with a simple search or from a link on the npm registry.

It's also worth checking a project's overall status. If a module is old but works reliably and has no open bugs, then it's probably safe. But if it has lots of bug reports and hasn't been updated in a long time, that's not a good sign!

10.4.2 Keeping your dependencies up to date

It's almost always a good idea to have the latest versions of things. People tune performance, fix bugs, and improve APIs. You *could* manually go through each of your dependencies to find out which versions were out of date, or you could use a tool built into npm: npm outdated.

Let's say that your project has Express 5.0.0 installed, but the latest version is 5.4.3 (which I'm sure will be out of date by the time you read this). In your project directory, run `npm outdated --depth 0` and you'll see output something like this:

```
Package        Current  Wanted  Latest  Location
express          5.0.0   5.4.3   5.4.3   express
```

If you have other outdated packages, this command will report those too. Go into your package.json, update the versions, and run `npm install` to get the latest versions. It's a good idea to check for outdated packages frequently.

What's that depth thing?

`npm outdated --depth 0` will tell you all of the modules that are outdated that you've installed. `npm outdated` without the `depth` flag tells you modules that are outdated, even ones you didn't directly install. For example, Express depends on a module called cookie. If cookie gets updated but Express doesn't update to the latest version of cookie, then you'll get a warning about cookie, even though it isn't your fault.

There's not much I can do if Express doesn't update to the latest version (that's largely out of my control), other than update to the latest version of Express (which is in my control). The `--depth` flag only shows actionable information, whereas leaving it out gives you a bunch of information you can't really use.

Another side note: you'll want to make sure that you're on the latest version of Node, too. Check https://nodejs.org and make sure you're on the latest version.

10.4.3 Check against the Node Security Project

Sometimes, modules have security issues. Some nice folks set up the Node Security Project, an ambitious undertaking to audit every module in the npm registry. If they find an insecure module, they post an advisory at http://nodesecurity.io/advisories.

The Node Security Project also comes with a command-line tool called nsp. It's a simple but powerful tool that scans your package.json for insecure dependencies (by comparing them against their database).

To install it, run `npm install -g nsp` to install the module globally. Now, in the same directory as your package.json, type

```
nsp audit-package
```

Most of the time, you'll get a nice message that tells you that your packages are known to be secure. But sometimes, one of your dependencies (or, more often, one of your dependencies' dependencies) has a security hole.

For example, Express depends on a module called serve-static; this is express.static, the static file middleware. In early 2015, a vulnerability was found in serve-static. If you're using a version of Express that depended on serve-static, run `nsp audit-package` and you'll see something like this:

```
Name           Installed  Patched  Vulnerable Dependency
serve-static       1.7.1  >=1.7.2    myproject > express
```

There are two important things here. The left column tells you the name of the problematic dependency. The right column shows you the chain of dependencies that leads to the problem. In this example, your project (called myproject) is the first issue, which depends on Express, which then depends on serve-static. This means that Express needs to update in order to get the latest version of serve-static. If you depended on serve-static directly, you'd only see your project name in the list, like this:

```
Name           Installed  Patched  Vulnerable Dependency
serve-static       1.7.1  >=1.7.2              myproject
```

Note that modules can still be insecure; there are *so many* modules on npm that the Node Security Project can't possibly audit all of them. But it's another helpful tool to keep your apps secure.

10.5 Handling server crashes

I have bad news: your servers might crash at some point. There are loads of things that can crash your servers: perhaps there's a bug in your code and you're referencing an undefined variable; perhaps a hacker has found a way to crash your server with

malicious input; perhaps your servers have reached their capacities. Unfortunately, these servers can get wildly complicated, and at some point, they might crash.

And, although this chapter has tips to help keep your apps running smoothly, you don't want a crash to completely ruin your day. You should recover from crashes and keep on chugging.

There is a simple tool called Forever (https://github.com/foreverjs/forever) that can help with this. Its name might be a hint: it keeps your apps running forever. The important part: if your app crashes, Forever will try to restart it.

To install Forever, run `npm install forever --save`. You've probably had an npm start script in your package.json for a while, and you need to change it from the code in the following listing to that in listing 10.14.

Listing 10.13 A classic npm start script

```
…
"scripts": {
  "start": "node app.js"
}
…
```

Listing 10.14 npm start with Forever

```
…
"scripts": {
  "start": "forever app.js"
}
…
```

And now your server will restart if it crashes!

> **NOTE** You can see a simple code example of this in action at the book's source code repository at https://github.com/EvanHahn/Express.js-in-Action-code/tree/master/Chapter_10/forever-example.

10.6 Various little tricks

We've covered most of the big topics like cross-site scripting and HTTPS. There are a few other tricks that you can employ to make your Express applications even more secure. The topics in this section are hardly as essential as the earlier ones, but they're quick and easy and can lower the number of places that you can be attacked.

10.6.1 No Express here

If a hacker wants to break into your site, they have a lot of things to try. If they know that your site is powered by Express and they know that Express or Node has some kind of security flaw, they can try to exploit it. It'd be better to leave hackers in the dark about this!

By default, however, Express publicizes itself. In every request, there's an HTTP header that identifies your site as powered by Express. `X-Powered-By: Express` is sent with every request, by default. You can easily disable it with a setting:

```
app.disable("x-powered-by");
```

Disabling the `x-powered-by option` disables the setting of the header. Disabling this will make it a little harder for hackers. It'll hardly make you invincible—there are plenty of other avenues for attack—but it can help a little, and every little bit helps.

10.6.2 Preventing clickjacking

I think clickjacking is quite clever. It's relatively easy to prevent, but I almost feel guilty for doing so. It's such a clever trick.

Imagine I'm a hacker, and I want to find out information from your private social networking profile. I'd love it if you would just make your profile public. It'd be so easy, if I could get you to click the big button shown in figure 10.4.

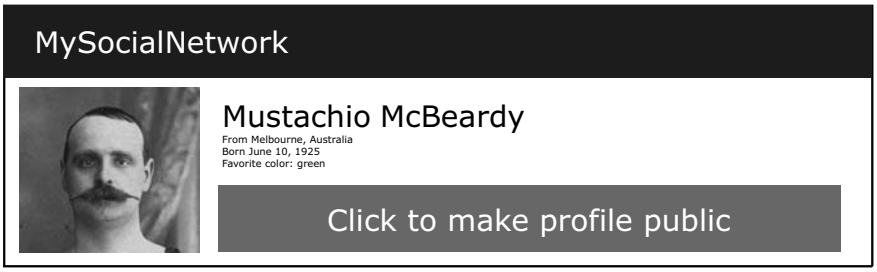


Figure 10.4 An example page for a social network

Clickjacking takes advantage of browser frames—the ability to embed one page in another—to make this happen. I could send you a link to an innocent-looking page, which might look something like figure 10.5.

Figure 10.5 An innocent-looking page that's concealing a clickjacking attack

Figure 10.6 Not so innocent now, is it?

But in reality, this innocent-looking page is concealing the social network page! There's an `<iframe>` of the social network site, and it's invisible. It's positioned *just right*, so that when you click "Click here to enter my page," you're actually clicking "Click to make profile public," as figure 10.6 reveals.

I don't know about you, but I think that's quite clever. Unfortunately for hackers, it's quite easily prevented.

Most browsers (and *all* modern ones) listen for a header called `X-Frame-Options`. If it's loading a frame or iframe and that page sends a restrictive `X-Frame-Options`, the browser won't load the frame any longer.

`X-Frame-Options` has three options. `DENY` keeps *anyone* from putting your site in a frame, period. `SAMEORIGIN` keeps anyone *else* from putting your site in a frame, but your own site is allowed. You can also let *one* other site through with the `ALLOW-FROM` option. I'd recommend the `SAMEORIGIN` or `DENY` options. As before, if you're using Helmet, you can set them quite easily, as shown in the following listing.

Listing 10.15 Keeping your app out of frames

```
app.use(helmet.frameguard("sameorigin"));
// or …
app.use(helmet.frameguard("deny"));
```

This Helmet middleware will set `X-Frame-Options` so you don't have to worry about your pages being susceptible to clickjacking attacks.

10.6.3 Keeping Adobe products out of your site

Adobe products like Flash Player and Reader can make cross-origin web requests. As a result, a Flash file could make requests to your server. If another website serves a malicious Flash file, users of that site could make arbitrary requests to your Express

application (likely unknowingly). This could cause them to hammer your server with requests or to load resources you don't intend them to.

This is easily preventable by adding a file at the root of your site called crossdomain.xml. When an Adobe product is going to load a file off of your domain, it will first check the crossdomain.xml file to make sure your domain allows it. As the administrator, you can define this XML file to keep certain Flash users in or out of your site. It's likely, however, that you don't want *any* Flash users on your page. In that case, make sure you're serving this XML content at the root of your site (at /crossdomain.xml), as the next listing shows.

Listing 10.16 The most restrictive crossdomain.xml

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM
➥ "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <site-control permitted-cross-domain-policies="none">
</cross-domain-policy>
```

This prevents any Flash users from loading content off of your site, unless they come from your domain. If you're interested in changing this policy, take a look at the spec at https://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html.

You can place the restrictive crossdomain.xml file into a directory for your static files so that it's served up when requested.

10.6.4 Don't let browsers infer the file type

Imagine a user has uploaded a plain-text file to your server called file.txt. Your server serves this with a `text/plain` content type, because it's plain text. So far, this is simple. But what if file.txt contains something like the script in the next listing?

Listing 10.17 A malicious script that could be stored as plain text

```
function stealUserData() {
  // something evil in here …
}
stealUserData();
```

Even though you're serving this file as plain text, this looks like JavaScript, and some browsers will try to sniff the file type. That means that you can still run that file with `<script src="file.txt"></script>`. Many browsers will allow file.txt to be run even if the content type isn't for JavaScript!

This example extends further if file.txt looks like HTML and the browser interprets it as HTML. That HTML page can contain malicious JavaScript, which could do lots of bad things!

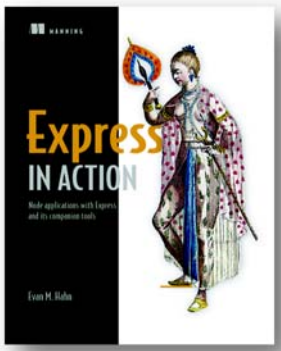
Luckily, you can fix this with a single HTTP header. You can set the `X-Content-Type-Options` header to its only option, `nosniff`. Helmet comes with `noSniff` middleware, and you can use it like this:

```
app.use(helmet.noSniff());
```

Nice that one HTTP header can fix this!

10.7 Summary

- Thinking like a hacker will help you spot security holes.
- Using a syntax checker like JSHint can help you spot bugs in your code.
- Parsing query strings in Express has a few pitfalls. Make sure you know what variable types your parameters could be.
- HTTPS should be used instead of HTTP.
- Cross-site scripting, cross-site request forgery, and man-in-the-middle attacks can be mitigated. Never trusting user input and verifying things each step of the way can help secure you.
- Crashing servers is a given. Forever is one tool that you can use to make sure your application restarts after a failure.
- Auditing your third-party code using the Node Security Project (and common sense!).



Node.js is white hot, powering the server side of major web apps from Walmart, PayPal, and Netflix. While super-powerful, raw Node can be complex and awkward. Express.js is a web application framework for Node that organizes your server-side JavaScript into testable, maintainable modules. It provides a powerful set of features to efficiently manage routes, requests, and views, along with beautiful boilerplate for your web applications. Lightweight, fast, and unobtrusive, Express helps you harness Node's raw power so you can concentrate on what your application does instead of managing nit-picky technical details.

Express in Action is a carefully-designed tutorial that teaches you how to build web applications using Node and Express. It starts by introducing Node's unique characteristics and then showing you how they map to the features of Express. With a clear vision of how an Express application looks, you'll systematically explore key development techniques, meet the rich ecosystem of companion tools and libraries, and even get a glimpse into its inner workings. After just a few chapters, you'll be able to build a simple Node app. By the end of the book, you'll know how to test it, hook it up to a database, and even automate the dev process.

What's inside

- Learn Express 4, the fastest way to spin up a Node application
- Using Grunt with CoffeeScript, Stylus, Jade, and more
- Testing with Mocha and Jasmine
- Data storage with Mongo and Redis
- Integrating with other libraries and tools

You'll need to know the basics of web application design and be proficient with JavaScript. No prior exposure to Node or Express is required.

The Compose layer

Once Things are on the web (the Access layer) where they can be found by humans and machines (the Find layer), and their resources can be shared securely with others (the Share layer), it's time to look at how to build large-scale, meaningful applications for the WoT. In other words, we need to understand the integration of data and services from heterogeneous Things into an immense ecosystem of web tools such as analytics software and mashup platforms. The goal of the Compose layer is to make it even simpler to create applications involving Things and virtual web services.

Tools at the Compose layer range from web toolkits (for example, JavaScript SDKs offering higher-level abstractions) to dashboards with programmable widgets and finally to physical mashup tools such as Node-RED. Inspired by Web 2.0 participatory services and in particular web mashups, the physical mashups offer a unified view of the classical web and the Web of Things and empower people to build applications using WoT services without requiring programming skills. We look at how to do this in *Building the Web of Things*.

Another aspect of the Compose layer is look at the *meaning of data*. The most interesting side of the IoT is the sheer amount of data it generates. Think about it: 1 million connected devices all sending a sensor reading (e.g., temperature) every second to an IoT cloud[1] means 86.4 billion messages per day—yes, *billion*. That's roughly 170 times more than all tweets posted globally that same day![2] Dealing with all of this IoT data really is challenging. It has the potential to make our world smart and more aware, but first we need to be able to process

[1] IoT clouds are services in the cloud that manage the connectivity IoT devices and offer additional features such as data storage or advanced analytics capabilities. Examples of IoT clouds are http://evrythng.com or https://thethings.io.

[2] See "What Happens in an Internet Minute": http://www.intel.co.uk/content/www/uk/en/communications/internet-minute-infographic.html.

and understand very large sets of data. This is exactly what you'll learn in "Example: NYC taxi data" from *Real-World Machine Learning*. In this chapter, you'll discover the power of manipulating big data with libraries such as Scikit-Learn[1] and Pandas[2] to extract insightful information about tipping opportunities for taxi drivers in New York City.

[1] See http://scikit-learn.org/stable/
[2] See http://pandas.pydata.org/

Chapter 6 from *Real-World Machine Learning*
by Henrik Brink, Joseph W. Richards,
and Mark Fetherolf

Example: NYC taxi data

This chapter covers

- Introducing, visualizing, and preparing a real-world dataset: NYC taxi trip information
- Building a classification model to predict whether the passenger(s) will tip the driver or not
- Optimizing models by tuning model parameters and engineering features
- Building and optimizing a regression model to predict the tip amountUsing these models to gain a deeper understanding of the data and the behavior of taxi drivers and passengers

In the previous five chapters you learned how to go from raw, messy data to building, validating, and optimizing models by tuning parameters and engineering features that capture the domain knowledge of the problem. Although we've used a variety of minor examples throughout these chapters to illustrate the points of the individual sections, it's time for you to use the knowledge you've acquired and work through a full real-world example. This is the first of three chapters (along with chapters 8 and 10) where the entire chapter is dedicated to a full example.

In the first section of this chapter, we'll take a closer look at the data and various useful visualizations that help you gain a better understanding of the possibilities of the data. We'll explain how the initial data preparation is performed, so the data is ready for our modeling experiments in the subsequent sections. In the second section we'll set up a classification problem and improve the performance of the model by tuning model parameters and engineering new features. The last section summarizes the chapter and provides a table of important terms.

6.1 Data: NYC taxi trip and fare information

With companies and organizations producing more and more data, a large set of very interesting datasets is becoming available. In addition, some of these organizations are embracing the concept of *open data*, enabling the use of the data by anyone interested in the domain at hand.

A very interesting dataset has become available through the U.S. Freedom of Information Law (FOIL): New York City taxi trip records from all of 2013.[1] This dataset collected various information on individual taxi trips, including the pickup and drop-off locations, trip time, distance, and fare amount. You'll see that this data qualifies as real-world data not only because of the way it's been generated but also in the way it's messy: there are missing data, spurious records, unimportant columns, people biases, and so on, and there's a lot of it. The full dataset is over 19 GB of CSV data, making it too large for most advanced machine learning (ML) algorithms on most systems to handle. In this chapter we're going to work with a smaller subset of the data. In chapters 9 and 10 we'll investigate methods that are able to scale to sizes like this and even larger.

The data is available for download at `http://www.andresmh.com/nyctaxitrips/`. It consists of 12 pairs of trip/fare compressed CSV files. There are ~14 million records in each file, where the trip/fare files are matched line by line.

We'll follow our basic ML workflow: analyzing the data; extracting features; building, evaluating, and optimizing models; and predicting on new data. In the next subsection, we'll take a look at the data using some of our visualization methods from chapter 2.

6.1.1 Visualizing the data

The first method for gaining an understanding of what the dataset contains is almost always to view it in a table. In all of the following, we've joined the trip/fare lines into a single dataset. Figure 6.1 shows the first six rows of data.

[1] Initially released in a blog post by Chris Wong: http://chriswhong.com/open-data/foil_nyc_taxi/

medallion	hack_license	vendor_id	rate_code	store_and_fwd_flag
CD847FE5884F10A28217E9FBA11B275B	5FEFD00D9773268B72EE4E879852F190	CMT	1	N
20D9ECB2CA0767CF7A01564DF2844A3E	598CCE5B9C1918568DEE71F43CF26CD2	CMT	1	N
A954A71B6D44265AE756BF807E069396	D5CA7D478A14BA3BBFC20153C5C88B1A	CMT	1	N
F6F7D02179BE915B23EF2DB57836442D	088879B44B80CC9ED43724776C539370	VTS	1	0
BE386D8524FCD16B3727DCF0A32D9B25	4EB96EC9F3A42794DEE233EC8A2616CE	VTS	1	0
E9FF471F36A91031FE5B6D6228674089	72E0B04464AD6513F6A613AABB04E701	VTS	1	0

pickup_datetime	dropoff_datetime	passenger_count	trip_time_in_secs	trip_distance
2013-01-08 10:44:06	2013-01-08 10:46:09	1	123	0.30
2013-01-08 07:51:16	2013-01-08 07:51:20	1	4	0.00
2013-01-07 10:05:50	2013-01-07 10:13:17	1	446	1.10
2013-01-13 04:36:00	2013-01-13 04:46:00	5	600	3.12
2013-01-13 04:37:00	2013-01-13 04:48:00	2	660	3.39
2013-01-13 04:41:00	2013-01-13 04:45:00	1	240	1.16

pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	payment_type
-73.989296	40.756313	-73.987885	40.751122	DIS
-73.945396	40.802090	-73.945412	40.802025	NOC
-73.989090	40.748367	-73.974983	40.756035	DIS
-73.996933	40.720055	-73.993546	40.693043	CRD
-74.000313	40.730068	-73.987373	40.768406	CRD
-73.997292	40.720982	-74.000443	40.732376	CRD

fare_amount	surcharge	mta_tax	tip_amount	tolls_amount	total_amount
3.5	0.0	0.5	0.00	0	4.00
2.5	0.0	0.5	0.00	0	3.00
7.0	0.0	0.5	0.00	0	7.50
12.0	0.5	0.5	1.75	0	14.75
12.0	0.5	0.5	3.12	0	16.12
5.5	0.5	0.5	1.20	0	7.70

Figure 6.1 The first six rows of the NYC taxi trip and fare record data. Most of the columns are self-explanatory, but we'll introduce some of them in more detail in the text.

The `medallion` and `hack_license` columns look like simple ID columns that are less useful from an ML perspective. A few of the columns look like categorical data, like `vendor_id`, `rate_code`, `store_and_fwd_flag`, and `payment_type`. Figure 6.2 shows the distribution of values in these categorical columns.

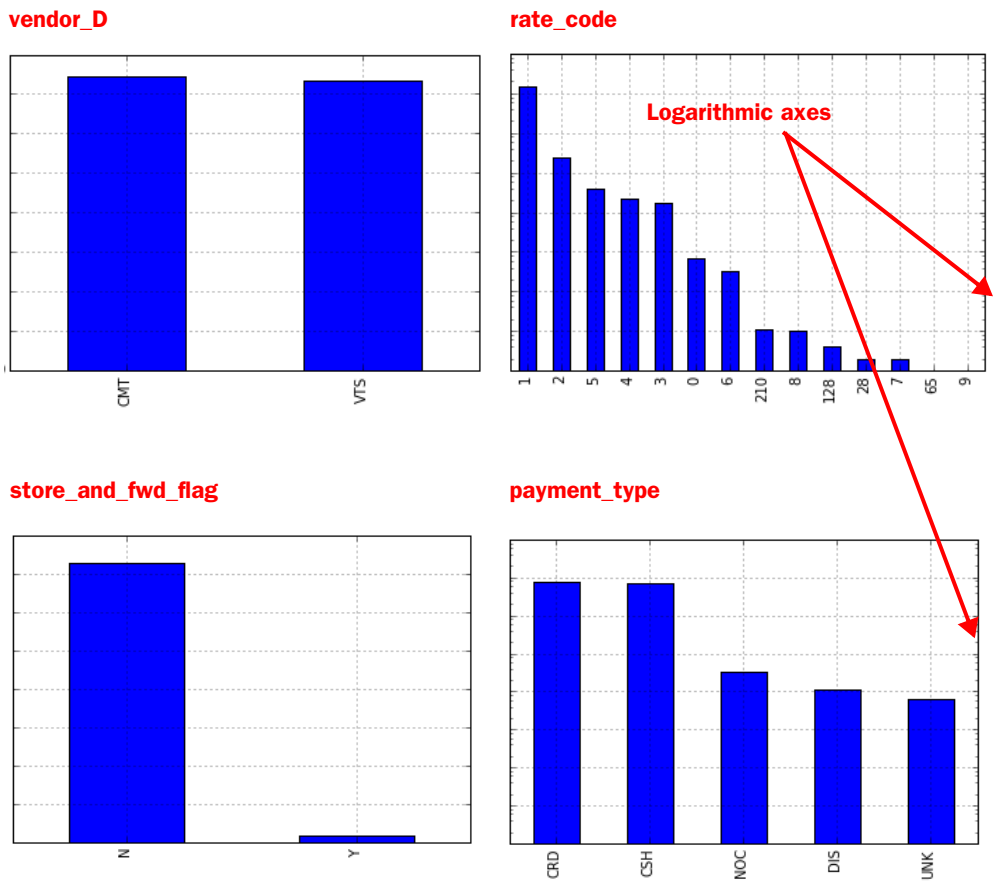


Figure 6.2 The distribution of values across some of the categorical-looking columns in our dataset

Let's look at some of the numerical columns in the dataset. It's interesting to validate, for example, that there are correlations between things like the duration (`trip_time_in_secs`), distance, and total cost of a trip. Figure 6.3 shows scatter plots of some of these plotted against each other.

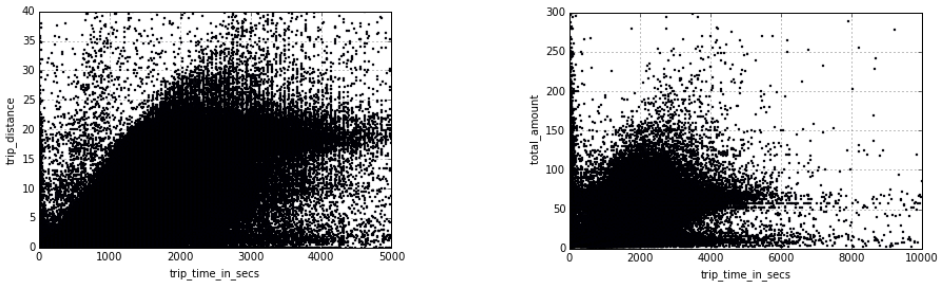


Figure 6.3 The scatter of trips for the time in seconds versus the total cost and the trip distance, respectively. There's a certain amount of correlation, as expected, but the scatter is still relatively high. There are also some less-logical clusters such as a lot of zero-time trips, even very expensive ones.

Lastly, figure 6.4 visualizes the pickup locations in the latitude/longitude space, defining a map of NYC taxi trips.

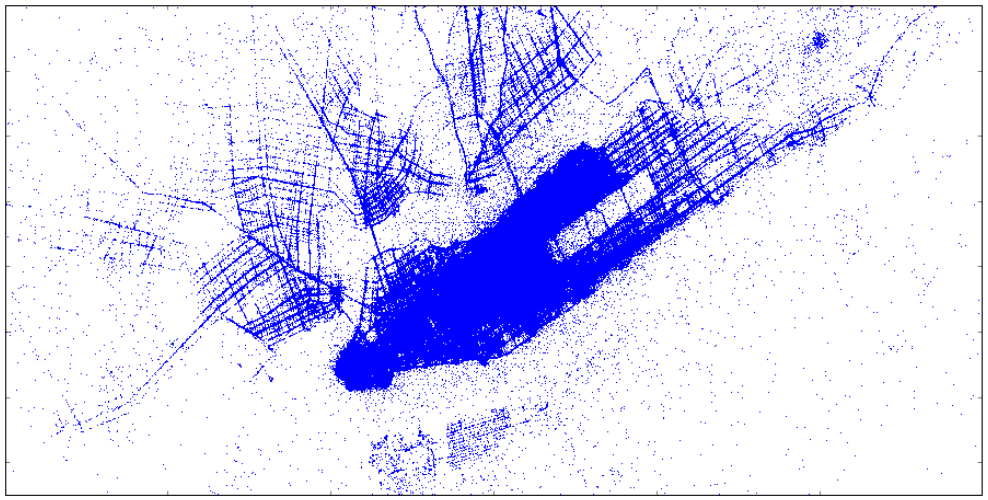


Figure 6.4 The latitude/longitude of pickup locations. Note that the X-axis is flipped, compared to a regular map. You can see a huge amount of pickups on Manhattan, dropping off as you move away from the city center.

With a fresh perspective on the data we're dealing with, let's go ahead and define how we want to use machine learning to solve a problem on this dataset.

6.1.2 Defining the problem and preparing the data

When we first looked at this data, a particular column immediately grabbed our attention: `tip_amount`. This column stores the information on the amount of the tip given for each ride, and it seems quite interesting to understand in greater detail what could cause a difference in the amount of the tip for a particular trip. We might want to build

a classifier that uses all of the trip information to try to predict whether passenger(s) will tip the driver or not. With such a model, we'd be able to predict tip versus no tip at the end of a specific trip. A taxi driver could have our model installed on a mobile device and would get no-tip alerts and be able to alter the situation before it was too late. While we wait for approval for having our app installed in all NYC taxis, we can simply use the model to give us insight into which parameters were most important, or predictive, of tip versus no tip. Figure 6.5 shows a histogram of the tip amount.

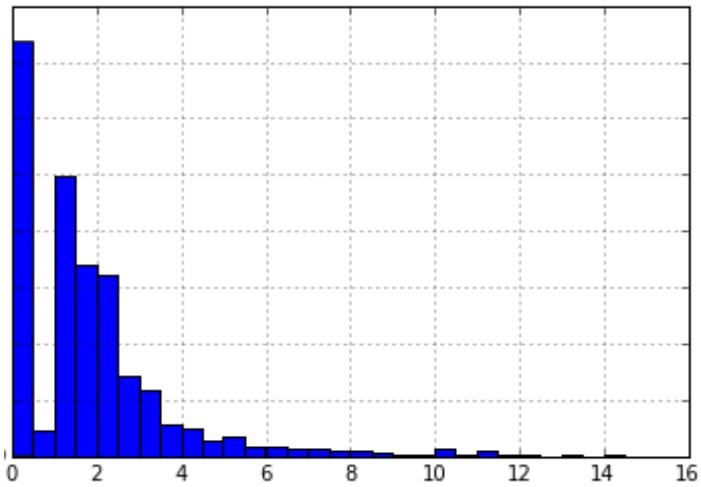


Figure 6.5 The distribution of tip amount. Around half the trips yielded $0 tips, which is more than we'd expect intuitively.

So, the idea for our model is to predict the no tips from the rest, in a binary classifier. The point is to be able to help the taxi driver predict a no-tip situation and to gain understanding of why such a situation might arise. That is, what are the driving factors in the dataset, if any, of trips yielding no tip?

A STORY FROM THE REAL WORLD

Before we start building this model, we'll tell you the real story of how our first attempt at this was quite unsuccessful, disguised as very successful—the worst kind of unsuccessful—and how we fixed it. This type of detour is extremely common when working with real data, so it's helpful to include the lessons learned here. When working with machine learning, it's important to watch out for two pitfalls: *too-good-to-be-true scenarios* and making *premature assumptions* that are not rooted in the data.

If the accuracy is higher than you would have expected, chances are your model is cheating somewhere. The real world is creative when trying to make your life as a data scientist difficult. When building initial tip/no-tip classification models, we quickly obtained a very high predictive accuracy of the model. Because we were excited about our new dataset and model, and we just nailed it, we ignored the warnings of a

cheating model. But having been bitten by such things a few times before, the results led us to investigate further. One of the things we looked at was the importance of the input features (as you'll see in more detail in later sections). In our case, a certain feature totally dominated the performance of the model: payment type.

From our own taxi experience, it could make sense. People paying with credit cards may have a lower probability of tipping. If you pay with cash, you almost always round up to whatever you have the bills for. So, we started segmenting the number of tips versus no tips for people paying with a credit card rather than cash. It looked like quite the majority (more than 95%) of millions of passengers paying with a credit card did actually tip. So much for that theory. So how many people paying with cash tipped? *All* of them, it seemed.

Because of the assumptions we had already made, we didn't look properly. Actually, *none* of the passengers paying with cash had tipped. Then it quickly became obvious. When a passenger pays with cash and gives a tip, the driver doesn't register it in whatever way is necessary to be part of the dataset in our possession. The data can't be trusted. This is a setback.

In a situation like this, there's a problem in the generation of the data, and there's no way to trust that part of the data for building an ML model. If the answers are incorrect, how is the model supposed to learn?

What we did to fix the problem was to remove from the dataset all of the trips payed for with cash. It always feels wrong to throw away data, but in this case we had no choice. Of course, there's no guarantee that any of the other tip recordings are not wrong as well, but we can at least check the new distribution of tip amounts. Figure 6.6 shows the histogram of tip amounts after filtering out any cash-payed trips.

With the bad data removed, the distribution is looking much better. There are only around 5% no-tip trips. Our job in the next section is to find out why.

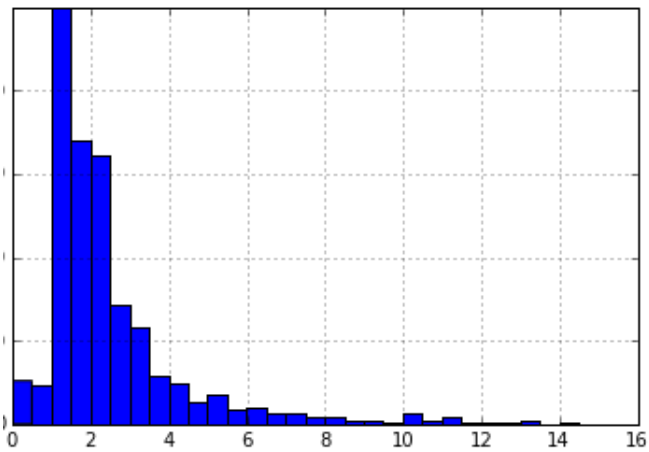


Figure 6.6 The distribution of tip amounts after omitting cash payments, after discovering that tips are never recorded in the system

6.2 Modeling

With the data prepared for modeling, we can easily use our knowledge from chapter 3 to set up and evaluate models. In the following subsections, we'll build different versions of the models, trying to improve the performance with each iteration.

6.2.1 Basic linear model

We'll start our modeling endeavor as simply as possible. We're going to work with the logistic regression algorithm, which is a simple, linear algorithm. We're also going to restrict ourselves initially to the numerical values in the dataset, because those are handled by the algorithms without any work.

We'll use the `scikit-learn` and `pandas` libraries in Python to develop our model. Before building the models, we shuffled the instances randomly and split the training and testing sets 80/20. We also need to scale the data so no column is considered more important than others a priori. If the data has been loaded into a pandas DataFrame, the code to build and validate this model then looks something like the following listing.

Listing 6.1 Logistic regression tip-prediction model

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import roc_curve, roc_auc_score
from pylab import *

sc = StandardScaler()
data_scaled = sc.fit_transform(data[feats])

sgd = SGDClassifier(loss="modified_huber")

sgd.fit(
     data.ix[train_idx,feats],
     data['tipped'].ix[train_idx]
)

preds = sgd.predict_proba(
     data.ix[test_idx,feats]
)

fpr, tpr, thr = roc_curve(
     data['tipped'].ix[test_idx],
     preds[:,1]
)
auc = roc_auc_score(data['tipped'].ix[test_idx], preds[:,1])

plot(fpr,tpr)
plot(fpr,fpr)
xlabel("False positive rate")
ylabel("True positive rate")
```

Scale the data to be between -1 and 1

Use loss-function that handles outliers well

Fit the classifier on the training features and target data

Make predictions on the held-out test set

Calculate ROC curve and AUC statistics

Plot ROC curve

The last part of listing 6.1 plots the ROC curve for our first classifier. It's shown in figure 6.7.

There's no way around it: the performance of this classifier is not good. With an AUC of around 0.5, the model is basically no better than random guessing, which is obviously not very useful. Luckily, we started out simply and have a few ways of trying to improve the performance of this model.

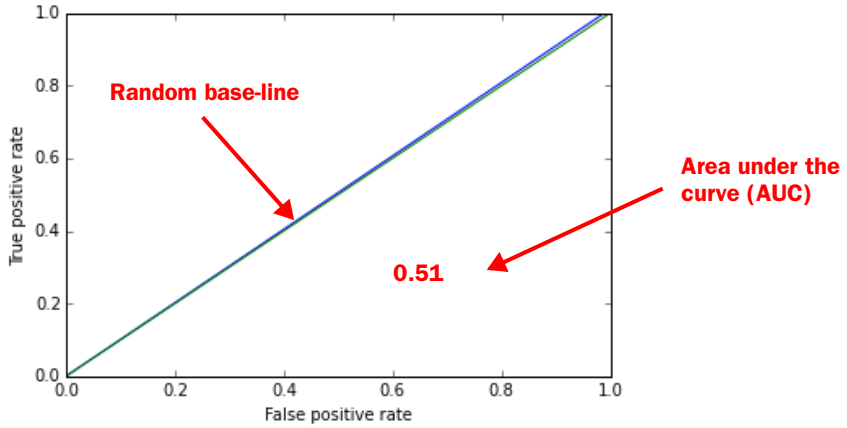


Figure 6.7 The receiving operator characteristic (ROC) curve of the linear logistic regression tip/no-tip classifier. With an area under the curve (AUC) of 0.5, the model seems to perform no better than random guessing. Not a good sign for our problem.

6.2.2 Nonlinear classifier

The first thing we'll try is to switch to a different algorithm—one that's nonlinear in nature. From figure 6.7, it seems that a linear model is just not going to cut it for us. Instead, we're going to use a nonlinear algorithm called *random forest*, well known for its high level of accuracy on real-world datasets. We could have chosen any of a number of other algorithms (see appendix A), but we'll leave it as an exercise to the reader to evaluate different algorithms. Here's the code (relative to the previous model) for building this model.

Listing 6.2 Random forest tip-prediction model

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_curve, roc_auc_score
from pylab import *

rf = RandomForestClassier(n_estimators=100)
rf.fit(data.ix[train_idx,feats], data['tipped'].ix[train_idx])
preds = rf.predict_proba(data.ix[test_idx,feats])

fpr, tpr, thr = roc_curve(data['tipped'].ix[test_idx], preds[:,1])
auc = roc_auc_score(data['tipped'].ix[test_idx], preds[:,1])

plot(fpr,tpr)

plot(fpr,fpr)
xlabel("False positive rate")
ylabel("True positive rate")

fi = zip(feats, rf.feature_importances_)
fi.sort(key=lambda x: -x[1])
fi = pandas.DataFrame(fi,
     columns=["Feature","Importance"])
```

Plot ROC curve

Features importance

The results of running the code in listing 6.2 are shown in figure 6.8. You see a significant increase in accuracy, and it's clear that there's a signal in the dataset: some combinations of the input features are capable of predicting whether a taxi trip will yield any tips from the passenger.

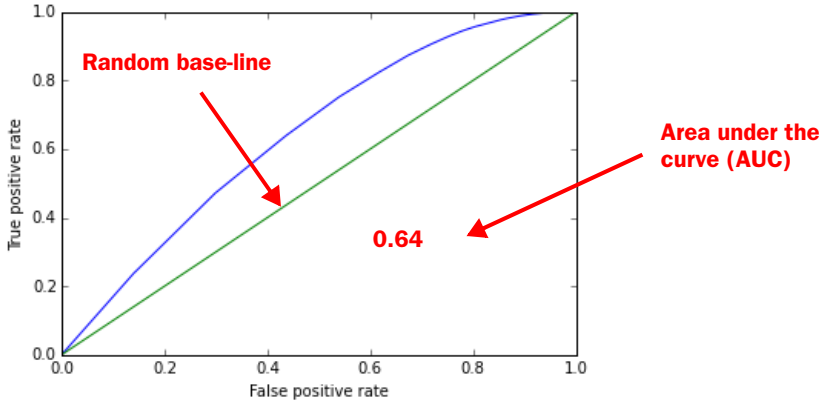


Figure 6.8 The ROC curve of the nonlinear random forest model. The AUC is significantly better: at 0.64 it's likely that there is a real signal in the dataset.

We can also use the model to gain some insight into what features were important in order to create this lift from the random baseline and successfully predict tip versus no-tip behavior. Figure 6.9 (also generated by the code in listing 6.2) shows the list of features and their relative importance for the random forest model. From this figure you can see that the location features are important, along with time, distance, and fare amount. It may be that riders in some parts of the city are less patient with slow, expensive rides, for example. We'll look more closely at the potential insights gained in section 6.2.5.

Now that we've chosen the algorithm, let us make sure we're actually using all of the raw features, including categorical columns and not just plain numerical columns.

	Feature	Importance
0	dropoff_latitude	0.165411
1	dropoff_longitude	0.163337
2	pickup_latitude	0.163068
3	pickup_longitude	0.160285
4	trip_time_in_secs	0.122214
5	trip_distance	0.112020
6	fare_amount	0.067795
7	passenger_count	0.017850
8	surcharge	0.014259
9	rate_code	0.006974
10	tolls_amount	0.004067
11	mta_tax	0.002720

Figure 6.9 important features of the random forest model. The drop-off and pickup location features seem to dominate the model.

6.2.3 Including categorical features

Without going deeper into the realm of feature engineering, we can perform some simple data preprocessing to increase the accuracy.

In chapter 2 you learned how to work with categorical features. Some ML algorithms work with categorical features directly, but we'll use the common trick of booleanizing our categorical features: creating a column of value 0 or 1 for each of the possible categories in the feature. This makes it possible for any ML algorithm to handle categorical data without changes to the algorithm itself.

The code for converting all of our categorical features is shown in the following listing.

Listing 6.3 Converting categorical columns to numerical features

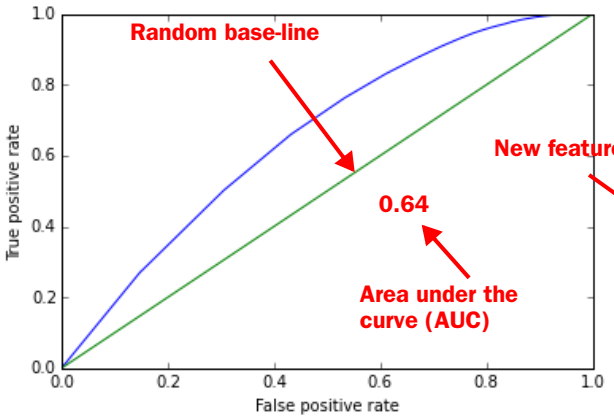
```
def cat_to_num(data):
    categories = unique(data)
    features = {}
    for cat in categories:
        binary = (data == cat)
        features["%s:%s"%(data.name, cat)] = binary.astype("int")
    return pandas.DataFrame(features)

payment_type_cats = cat_to_num(data[' payment_type'])
vendor_id_cats = cat_to_num(data['vendor_id'])
store_and_fwd_flag_cats = cat_to_num(data['store_and_fwd_flag'])
rate_code_cats = cat_to_num(data['rate_code'])

data = data.join(payment_type_cats)
data = data.join(vendor_id_cats)
data = data.join(store_and_fwd_flag_cats)
data = data.join(rate_code_cats)
```

A function for converting a categorical column to a set of numerical columns

Convert four categorical features in the dataset to numerical

Add the converted data to the full dataset used for training and testing

After creating the booleanized columns, we run the data through listing 6.2 again and obtain the ROC curve and feature importance list shown in figure 6.10.



	Feature	Importance
0	dropoff_latitude	0.163023
1	pickup_latitude	0.161114
2	dropoff_longitude	0.160988
3	pickup_longitude	0.158672
4	trip_time_in_secs	0.111172
5	trip_distance	0.106693
6	fare_amount	0.067567
7	passenger_count	0.019286
8	surcharge	0.010330
9	payment_type:NOC	0.008361
10	payment_type:CRD	0.008247

Figure 6.10 The ROC curve and feature importance list of the random forest model with all categorical variables converted to boolean (0/1) columns, one per category per feature. The new features are bringing new useful information to the table, because the AUC is seen to increase from the previous model without categorical features.

As the model performance increases, we can consider additional factors. We haven't done any real feature engineering, of course, because the data transformations applied so far are considered basic data preprocessing.

6.2.4 Including date-time features

At this point, it's time to start working with the data to produce new features, what you've previously known as feature engineering. In chapter 5, we introduced a set of date-time features transforming date and timestamps into numerical columns. You can easily imagine the time of the day or day of week to have some kind of influence on how a passenger will tip.

The code for calculating these features is presented in the following listing.

Listing 6.4 Date-time features

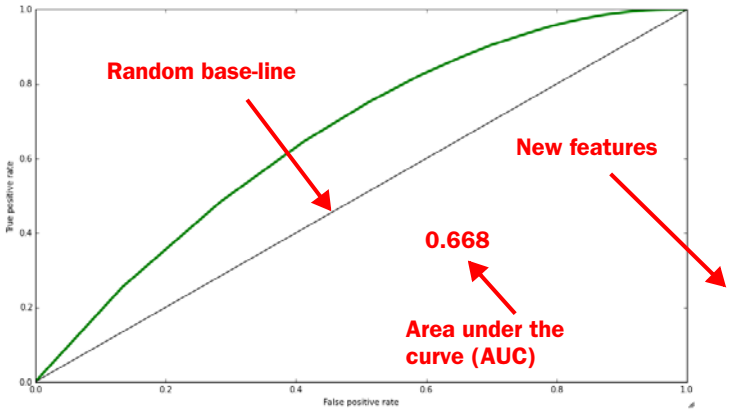
```
# Datetime features (hour of day, day of week, week of year)

pickup = pandas.to_datetime(data['pickup_datetime'])
dropoff = pandas.to_datetime(data['dropoff_datetime'])
data['pickup_hour'] = pickup.apply(lambda e: e.hour)
data['pickup_day'] = pickup.apply(lambda e: e.dayofweek)
data['pickup_week'] = pickup.apply(lambda e: e.week)
data['dropoff_hour'] = dropoff.apply(lambda e: e.hour)
data['dropoff_day'] = dropoff.apply(lambda e: e.dayofweek)
data['dropoff_week'] = dropoff.apply(lambda e: e.week)
```

Convert date-time columns (text) to real dates and times

Add hour, day, and week features to pickup times

Add hour, day, and week features to dropoff times

With our date-time features, we can go ahead and build our new model. We run the data through the code in listing 6.2 once again and obtain the ROC curve and feature importance shown in figure 6.11.



	Feature	Importance
0	dropoff_latitude	1.342966e-01
1	dropoff_longitude	1.332890e-01
2	pickup_latitude	1.326898e-01
3	pickup_longitude	1.314082e-01
4	trip_time_in_secs	1.000610e-01
5	trip_distance	9.924300e-02
6	fare_amount	7.099682e-02
7	dropoff_hour	3.942205e-02
8	pickup_hour	3.924598e-02
9	pickup_day	2.417833e-02
10	dropoff_day	2.413835e-02

Figure 6.11 The ROC curve and feature importance list for the random forest model including all categorical features and additional date-time features

We've seen an interesting evolution in the accuracy of the model with additional data preprocessing and feature engineering. We're able to predict if a passenger will tip the driver with an accuracy above random. At this point, we've only looked at the improving the data in order to improve the model. There are two other ways we can try to improve this model: vary the model parameters to see if the default values are not necessarily the most optimal and increase the dataset size. In this chapter, we've been heavily subsampling the dataset in order for the algorithms to handle the dataset, even on a 16 GB memory machine. We'll talk more about scalability of methods in chapters 9 and 10, but in the meantime we'll leave it to you to work with this data to try to improve the accuracy even further.

6.2.5 Model insights

It'sinteresting to use the model to gain insight about the data from the act of building a model to predict a certain answer. From the feature importance list, we can understand which parameters have most predictive power, and we use that to look at the data in new ways. In our initial unsuccessful attempt, it was because of inspection of the feature importance list that we discovered the problem with the data. In the current working model, we can also use the list to inspire some new visualizations.

At every iteration of our model in this section, the most important features have been the pickup and drop-off location features. In figure 6.12 we're plotting the geographical distribution of drop-offs that yield tips from the passenger, as well as drop-offs from trips that don't.

Figure 6.12 shows an interesting trend of not tipping when being dropped off closer to the center of the city. Why is that? One possibility is that the traffic situation creates many slow trips and the passenger is not necessarily happy with the driver's behavior. As a non–U.S. citizen, I have another theory. This particular area of the city

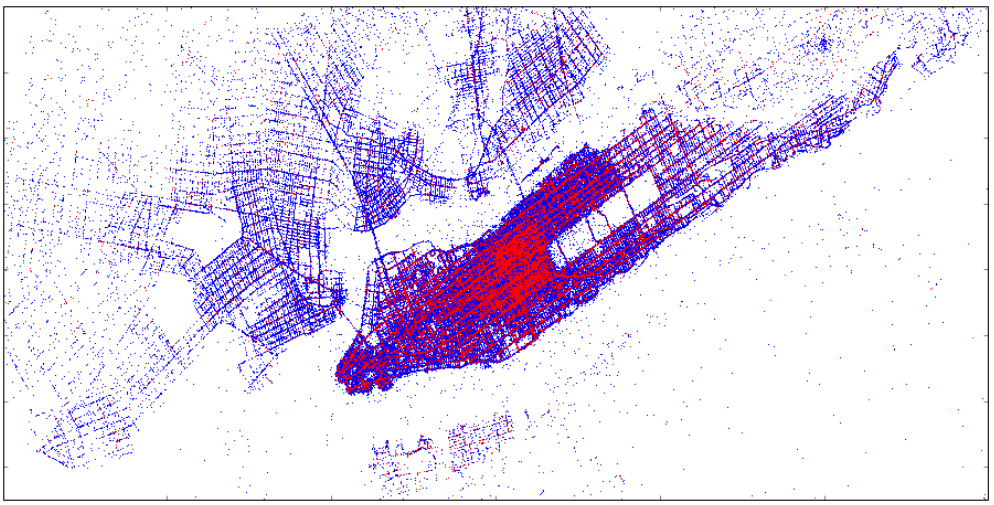


Figure 6.12 The geographical distribution of drop-offs colored by tip (blue) and no-tip (red)

has a high volume of both financial workers and tourists. We would expect the financial group to be distributed farther south on Manhattan. There's another reason why tourists are the most likely cause of this discrepancy, in my mind: many countries have vastly different rules for tipping than in the United States. Some Asian countries almost never tip, and many northern European countries tip much less and rarely in taxis. You can make many other interesting investigations based on this dataset. The point is, of course, that real-world data can often be used to say something interesting about the real world and the people generating the data.

6.3 Summary

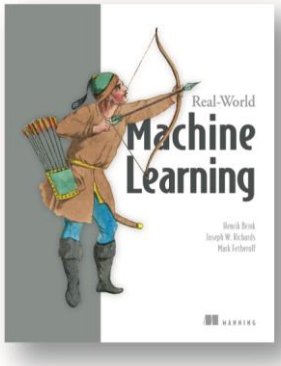
In this chapter we introduced a dataset from the real world and defined a problem suitable for the machine learning knowledge that has built up over the previous five chapters. We went through the entire ML workflow, including initial data preparation, feature engineering, and multiple iterations of model building, evaluation, and optimization. The main takeaways from the chapter are these:

- With more and more organizations producing vast amounts of data, increasing amounts of data are becoming available within organizations, if not publicly.
- Records of all taxi trips from NYC in 2013 have been released publicly. There are a lot of taxi trips in NYC in one year!
- Real-world data can be messy. Visualization and knowledge about the domain helps. Don't get caught in too-good-to-be-true scenarios and don't make premature assumptions about the data.
- Start iterating from the simplest possible model. Don't spend time on premature optimization. Gradually increase complexity.
- Make choices and move on; for example, decide on an algorithm early on. In an ideal world you'd try all combinations at all steps in the iterative process of building a model, but you'll have to fix some things in order to make progress.
- Gain insights into the model and the data in order to learn about the domain and potentially improve the model further.

Table 6.1 lists the important terms from this chapter.

Table 6.1 Vocabulary from chapter 6

Word	Definition
Open dataData made available publicly by institutions and organizations.	
FOIL	Freedom of Information Law. Also known as FOIA (Freedom of Information Act).
Too-good-to-be-true scenario	If a model is extremely accurate compared to what you would have thought, chances are there are some features in the model, or some data peculiarities, causing the model to be "cheating."
Premature assumptions	Assuming something about the data without validation, risking biasing your views on the results.



In a world where big data is the norm and near-real-time decisions are crucial, machine learning is a critical component of the data workflow. Machine learning systems can quickly crunch massive amounts of information to offer insight and make decisions in a way that matches or even surpasses human cognitive abilities. These systems use sophisticated computational and statistical tools to build models that can recognize and visualize patterns, predict outcomes, forecast values, and make recommendations. Gartner predicts that big data analytics will be a $25 billion market by 2017, and financial firms, marketing organizations, scientific facilities, and Silicon Valley startups are all demanding machine learning skills from their developers.

Real-World Machine Learning is a practical guide designed to teach working developers the art of ML project execution. Without overdosing you on academic theory and complex mathematics, it introduces the day-to-day practice of machine learning, preparing you to successfully build and deploy powerful ML systems. Using the Python language and the R statistical package, you'll start with core concepts like data acquisition and modeling, classification, and regression. You'll then move through the most important ML tasks, like model validation, optimization and feature engineering. By following numerous real-world examples, you'll learn how to anticipate and overcome common pitfalls. Along the way, you'll discover scalable and online algorithms for large and streaming data sets. Advanced readers will appreciate the in-depth discussion of enhanced ML systems through advanced data exploration and pre-processing methods.

What's inside

- Build and maintain your own ML system
- Detailed treatment of real-world use-cases
- ML workflow, practical considerations and common pitfalls
- Python and R code snippets
- Feature engineering, computational scalability, and real-time streaming ML

Code examples are in Python and R. No prior machine learning experience required.

The Compose layer is also about visualizing IoT data on the web. As you probably noticed in the previous chapter, data without visualizations isn't really useful at all. Creating compelling visualizations for IoT data is far from trivial due to the unprecedented amount of data generated by IoT devices. Don't worry though—there are a number great web tools ready to come to the rescue. In this final selected chapter, "Big data visualization" from *D3.js in Action*, you'll learn how to use one of the most impressive and scalable web visualization libraries out there: D3.js[1] .

[1] See https://d3js.org/

Chapter 11 from *D3.js in Action*
by Elijah Meeks

Big data visualization

This chapter covers

- Creating large random datasets of multiple types
- Using HTML5 canvas in conjunction with SVG to draw large datasets
- Optimizing geospatial, network, and traditional dataviz
- Working with quadtrees to enhance spatial search performance

This chapter focuses on techniques to create data visualization with large amounts of data. Because it would be impractical to include a few large datasets, we'll also touch on how to create large amounts of sample data to test your code with. You'll use several layouts that you saw earlier, such as the force-directed network layout from chapter 6 and the geospatial map from chapter 7, as well as the brush component from chapter 9, except this time you'll use it to select regions across the x- and y-axes.

This chapter touches on an exotic piece of functionality in D3: the quadtree (shown in figure 11.1). This is an advanced technique we'll use to improve interactivity

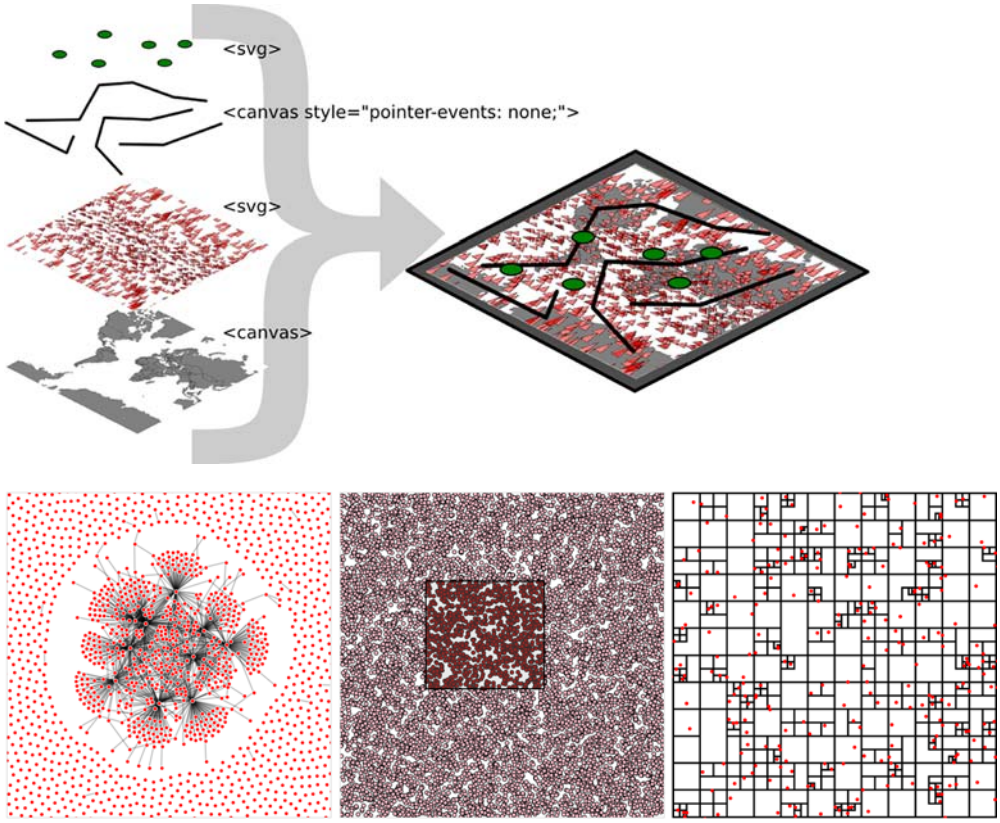


Figure 11.1 This chapter focuses on optimization techniques such as using HTML5 canvas to draw large datasets in tandem with SVG for the interactive elements. This is demonstrated with maps (section 11.1), networks (11.2), and traditional xy data (section 11.3), which uses the D3 quadtree function (section 11.3.2).

and performance. We'll also revisit HTML5 canvas throughout the chapter to see how we can use canvas in tandem with SVG to get the high performance and maintain the interactivity that SVG is so useful for.

We've worked with data throughout this book, but this time, we'll appreciably up the ante by trying to represent a thousand or more datapoints using maps, networks, and charts, which are significantly more resource-intensive than a circle pack chart, a bar chart, or a spreadsheet.

11.1 Big geodata

In chapter 7, you had only 10 cities representing the entire globe. That's not typical: when you're working with geodata, you'll often work with large datasets describing many complex shapes. Fortunately, there's built-in functionality in D3 for drawing that complex data with HTML5 canvas, which dramatically improves performance. For this chapter, we'll need to include a `<canvas>` element in our DOM.

Listing 11.1 bigdata.html

```
<!doctype html>
<html>
<head>
  <title>Big Data Visualization</title>
  <meta charset="utf-8" />
    <link type="text/css" rel="stylesheet" href="bigdata.css" />
</head>
<body>
<div>
<canvas height="500" width="500"></canvas>
   <div id="viz">
      <svg></svg>
</div>
</div>
<footer>
<script src="d3.v3.min.js" type="text/javascript"></script>
</footer>
</body>
</html>
```

Make sure to set the height and width attributes, not just the style attributes.

To handle our <canvas> element, as well as some of the visual elements we'll create in this chapter, we need to account for them in our CSS, as in the following listing. We want our <canvas> element to line up with our <svg> element so that we can use HTML5 canvas as a background layer to any SVG elements we create.

Listing 11.2 bigdata.css

```
 body, html {
    margin: 0;
  }
canvas {
  position: absolute;
  width: 500px;
  height: 500px;
}
svg {
  position: absolute;
  width:500px;
  height:500px;
}
path.country {
    fill: gray;
    stroke-width: 1;
    stroke: black;
    opacity: .5;
}
path.sample {
    stroke: black;
    stroke-width: 1px;
    fill: red;
    fill-opacity: .5;
}
```

In this chapter we'll draw SVG over canvas, so the canvas element needs to have the same attributes as the SVG element.

Likewise, identical settings for the SVG element

```
line.link {
      stroke-width: 1px;
      stroke: black;
      stroke-opacity: .5;
}
circle.node {
  fill: red;
  stroke: white;
  stroke-width: 1px;
}
circle.xy {
  fill: pink;
  stroke: black;
  stroke-width: 1px;
}
```

11.1.1 Creating random geodata

The first thing we need is a dataset with a thousand datapoints. Rather than using data from a pregenerated file, we'll invent it. One useful function available in D3 is `d3.range()`, which allows you to create an array of values. We'll use `d3.range()` to create an array of a thousand values. We'll then use that array to populate an array of objects with enough data to put on a network and on a map. Because we're going to put this data on a map, we need to make sure it's properly formatted geoJSON, as in the following listing, which uses the `randomCoords()` function to create triangles.

Listing 11.3 Creating sample data

```
var sampleData = d3.range(1000).map(function(d) {
   var datapoint = {};
   datapoint.id = "Sample Feature " + d;
   datapoint.type = "Feature";
   datapoint.properties = {};
   datapoint.geometry = {};
   datapoint.geometry.type = "Polygon";
   datapoint.geometry.coordinates = randomCoords();
   return datapoint;
});

function randomCoords() {
  var randX = (Math.random() * 350) - 175;
  var randY = (Math.random() * 170) - 85;
  return [[[randX - 5,randY],[randX,randY - 5],
        [randX - 10,randY - 5],[randX - 5,randY]]];
};
```

d3.range creates an array that we immediately map to an object array.

Each datapoint is an object with the necessary attributes to be placed on a map.

Draws a triangle around each random lat/long coordinate pair

After we have this data, we can throw it on a map like the one we first created in chapter 7. In the following listing we use the world.geojson file from chapter 7, so that we have some context for where the triangles are drawn.

Listing 11.4 Drawing a map with our sample data on it

```
d3.json("world.geojson", function(data) {createMap(data)});

function createMap(countries) {
  var projection = d3.geo.mercator()
       .scale(100).translate([250,250])

  var geoPath = d3.geo.path().projection(projection);
  var g = d3.select("svg").append("g");

  g.selectAll("path.country")
      .data(countries.features)
      .enter()
      .append("path")
      .attr("d", geoPath)
      .attr("class", "country");

  g.selectAll("path.sample")
      .data(sampleData)
      .enter()
      .append("path")
      .attr("d", geoPath)
      .attr("class", "sample");

};
```

Adjusts the projection and translation of the projection rather than the <g> so we can use the projection later to draw to canvas

Although our random triangles will obviously be in different places, our code should still produce something that looks like figure 11.2.

A thousand datapoints isn't very many, even on a small map like this. And in any browser that supports SVG, the data should be able to render quickly and provide you with the kind of functionality, like mouseover and click events, that you may want from your data display. But if you add zoom controls, like you see in listing 11.5 (the same zooming we had in chapter 7), then you'll notice that the performance of the

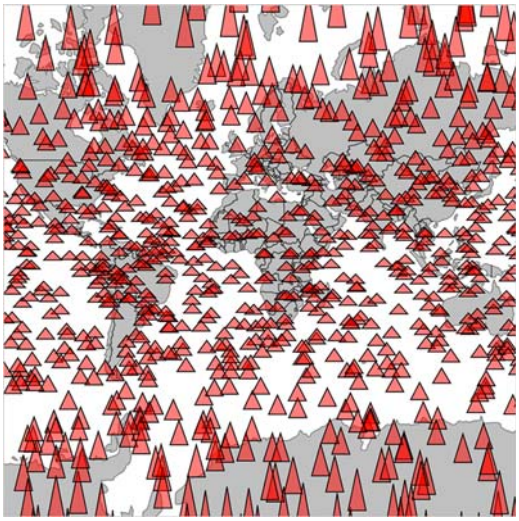


Figure 11.2 Drawing random triangles on a map entirely with SVG

Infoviz term: big data visualization

By the time you read this book, *big data* will probably sound as dated as *Pentium II*, *Rich Internet Application*, or *Buffy Cosplay*. Big data and all the excitement surrounding big data resulted from the broad availability of large datasets that were previously too large to handle. Often, big data is associated with exotic data stores like Hadoop or specialized techniques like GPU supercomputing (along with overpriced consultants).

But what constitutes *big* is in the eye of the beholder. In the domain of data visualization, the representation of big data doesn't typically mean placing thousands (or millions or trillions) of individual datapoints onscreen at once. Rather, it tends to mean demographic, topological, and other traditional statistical analysis of these massive datasets. Counterintuitively, big data visualization often takes the form of pie charts and bar charts. But when you look at traditional practice with presenting data interactively—natively—in the browser, the size of the datasets you're dealing with in this chapter really can be considered "*big*."

zooming and panning of the map isn't so great. If you expect your users to be on mobile, then optimization is still a good idea.

Listing 11.5 Adding zoom controls to a map

```
var mapZoom = d3.behavior.zoom().translate(projection.translate())
    .scale(projection.scale()).on("zoom", zoomed);
d3.select("svg").call(mapZoom);

function zoomed() {
     projection
       .translate(mapZoom.translate())
       .scale(mapZoom.scale());

     d3.selectAll("path.sample").attr("d", geoPath);
     d3.selectAll("path.country").attr("d", geoPath);
};
```

We use projection zoom in this example because it'll be easier to draw canvas elements later.

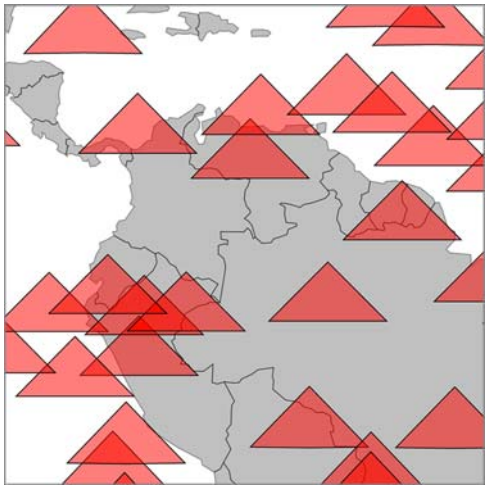


Figure 11.3 Zooming in on the sample geodata around South America

Now we can zoom into our map and pan around, as shown in figure 11.3. If you expect your users to be on browsers that handle SVG very well, like Chrome or Safari, and you don't expect to put more features on a map, then you may not even need to worry about optimization.

But what if you want to build interactive websites that work on all modern browsers? Firefox doesn't have the best SVG performance, and zooming this map in Firefox isn't a pleasant experience. If you change your `d3.range()` setting from 1000 to 5000, then even browsers that handle SVG well start to slow down.

11.1.2 Drawing geodata with canvas

One solution for optimization, which we touched on earlier, is to draw the elements with canvas instead of SVG. That's why we have a canvas element in our sample HTML page for this chapter, and why it's styled in such a way as to be directly underneath our `<svg>` element. Instead of creating SVG elements using D3's enter syntax, we use the built-in functionality in `d3.geo.path` to provide a context for HTML5 canvas. In the following listing, you can see how to use that built-in functionality with your existing dataset.

Listing 11.6 Drawing the map with canvas

```
function createMap(countries) {
  var projection = d3.geo.mercator().scale(50).translate([150,100]);
  var geoPath = d3.geo.path().projection(projection);

  var mapZoom = d3.behavior.zoom().translate(projection.translate())
             .scale(projection.scale()).on("zoom", zoomed);

  d3.select("svg").call(mapZoom);

  zoomed();

  function zoomed() {

    projection.translate(mapZoom.translate()).scale(mapZoom.scale());

    var context = d3.select("canvas").node().getContext("2d");
    context.clearRect(0,0,500,500);
    geoPath.context(context);

    context.strokeStyle = "black";
    context.fillStyle = "gray";
    context.lineWidth = "1px";
    for (var x in countries.features) {
      context.beginPath();
  geoPath(countries.features[x]);
      context.stroke()
      context.fill();
    }

    context.strokeStyle = "black";
    context.fillStyle = "rgba(255,0,0,.2)";
    context.lineWidth = "1px";
    for (var x in sampleData) {
```

Always clear the canvas before redrawing it if you're updating it.

Switches geoPath to a context generator with our canvas context

Styles settings for countries

Draws each country feature to canvas

```
        context.beginPath();
        geoPath(sampleData[x]);
        context.stroke();
        context.fill();
      }
    };
};
```

Draws each triangle to canvas

You can see some key differences between listings 11.6 and 11.5. In contrast with SVG, where you can move elements around as well as redraw them, you always have to clear and redraw the canvas to update it. Although it seems this would be slower, performance increases on all browsers, especially those that don't have the best SVG performance, because you don't need to manage hundreds or thousands of DOM elements. The graphical results, as seen in figure 11.4, demonstrate that it's hard to see the difference between SVG and canvas rendering.

11.1.3 Mixed-mode rendering techniques

The drawback with using canvas is that you can't easily provide the level of interactivity you may want for your data visualization. Typically, you draw your interactive elements with SVG and your large datasets with canvas. If we assume that the countries we're drawing aren't going to provide any interactivity, but the triangles will, then we can render the triangles as SVG and render the countries as canvas using the code in the following listing. This requires that we initialize two versions of `d3.geo.path`—one for drawing SVG and one for drawing canvas—and then we use both in our zoomed function.

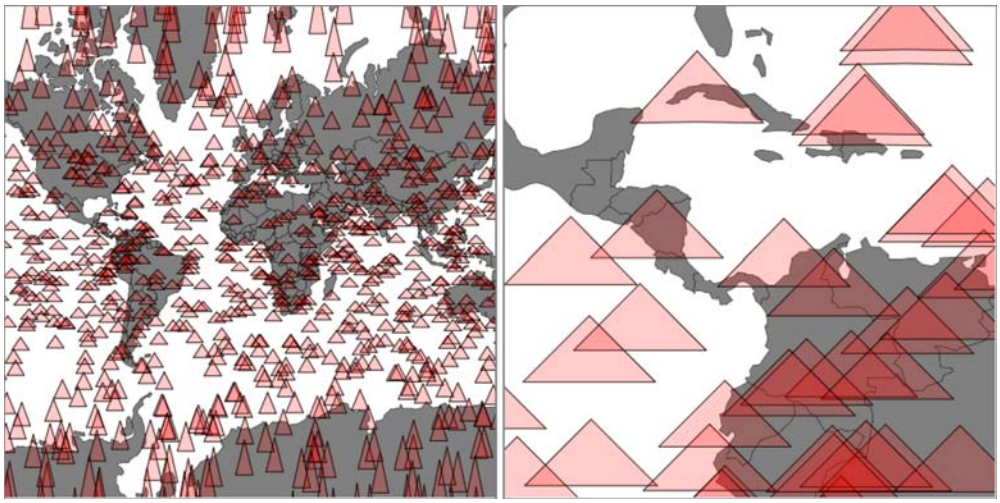


Figure 11.4 Drawing our map with canvas produces higher performance, but slightly less crisp graphics. On the left, it may seem like the triangles are as smoothly rendered as the earlier SVG triangles, but if you zoom in as we've done on the right, you can start to see clearly the slightly pixelated canvas rendering.

Listing 11.7 Rendering SVG and canvas simultaneously

```
function createMap(countries) {
    var projection = d3.geo.mercator().scale(100).translate([250,250]);
    var svgPath = d3.geo.path().projection(projection);
    var canvasPath = d3.geo.path().projection(projection);

    var mapZoom = d3.behavior.zoom()
      .translate(projection.translate())
      .scale(projection.scale())
      .on("zoom", zoomed);

    d3.select("svg").call(mapZoom);

    var g = d3.select("svg");

    g.selectAll("path.sample")
      .data(sampleData)
      .enter()
      .append("path")
      .attr("class", "sample")
      .on("mouseover", function() {d3.select(this).style("fill", "pink")});

    zoomed();

    function zoomed() {
      projection.translate(mapZoom.translate()).scale(mapZoom.scale());

      var context = d3.select("canvas").node().getContext("2d");
      context.clearRect(0,0,500,500);
      canvasPath.context(context);

      context.strokeStyle = "black";
      context.fillStyle = "gray";
      context.lineWidth = "1px";
      for (var x in countries.features) {
        context.beginPath();
        canvasPath(countries.features[x]);
        context.stroke();
        context.fill();
      }

    d3.selectAll("path.sample").attr("d", svgPath);
  };
};
```

We need to instantiate a different d3.geo.path for canvas and for SVG.

Updates the map when it's first created

Draws canvas features with canvasPath

Draws SVG features with svgPath

This allows us to maintain interactivity, such as the mouseover function on our triangles to change any triangle's color to pink when moused over. This approach maximizes performance by rendering any graphics that have no interactivity using HTML5 canvas instead of SVG. As shown in figure 11.5, the appearance produced using this method is virtually identical to that using canvas only or SVG only.

But what if you have massive numbers of elements and you really do want interactivity on all them, but you also want to give the user the ability to pan and drag? In that case, you have to embrace an extension of this mixed-mode rendering. You render in canvas whenever users are interacting in such a way that they can't interact with other elements. In other words, we need to render the triangles in canvas when the map is

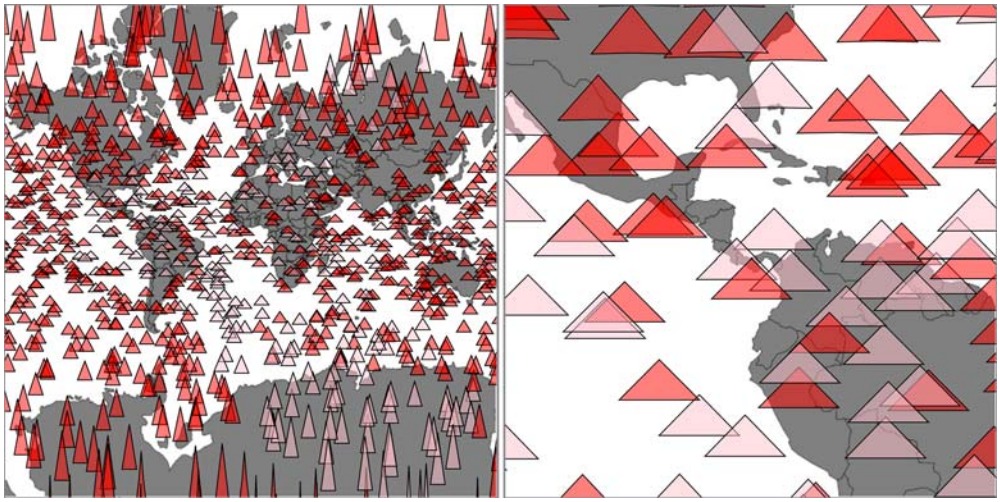


Figure 11.5 Background countries are drawn with canvas, while foreground triangles are drawn with SVG to use interactivity. SVG graphics are individual elements in the DOM and therefore amenable to having click, mouseover, and other event listeners attached to them.

being zoomed and panned, but render them in SVG when the map isn't in motion and the user is mousing over certain elements.

How do you determine when a zoom event starts and when it finishes? In the past you had to set a timer, check to see if the user was still zooming, and then redraw the elements. But, fortunately, D3 introduced a pair of new events to the zoom control: `"zoomstart"` and `"zoomend"`. These fire, as you may guess, when the zoom event begins and ends, respectively. The following listing shows how you'd initialize a zoom behavior with different functions for these different events.

Listing 11.8 Mixed rendering based on zoom interaction

```
var projection = d3.geo.mercator().scale(100).translate([250,250]);
var svgPath = d3.geo.path().projection(projection);
var canvasPath = d3.geo.path().projection(projection);

 mapZoom = d3.behavior.zoom()
    .translate(projection.translate())
    .scale(projection.scale())
    .on("zoom", zoomed)
    .on("zoomstart", zoomInitialized)
    .on("zoomend", zoomFinished);

 d3.select("svg").call(mapZoom);

 var g = d3.select("svg").append("g")

 g.selectAll("path.sample").data(sampleData)
    .enter()
    .append("path")
```

Assigns separate functions for each zoom state

```
      .attr("class", "sample")
      .on("mouseover", function() {
        d3.select(this).style("fill", "pink");
      });

  zoomFinished();
```

We have to call zoomFinished (listing 11.9) to draw the canvas countries with SVG triangles.

This allows us to restore our canvas drawing code for triangles to the `zoomed` function and to move the SVG rendering code out of the `zoomed` function and into a new `zoomFinished` function. We also need to hide the SVG triangles when zooming or panning starts by creating a `zoomInitialized` function that itself also fires the `zoomed` function (to draw the triangles we just hid, but in canvas). Finally, our `zoomFinished` function also contains the canvas drawing code necessary to only draw the countries. The different drawing strategies based on `zoom` events are shown in table 11.1.

Table 11.1 Rendering action based on `zoom` event

`zoom` event	*Countries* rendered as	Triangles rendered as
`zoomed`	Canvas	Canvas
`zoomInitialized`	Canvas	Hide SVG
`zoomFinished`	Canvas	SVG

As you can see in the following listing, this code is inefficient, but I wanted to be explicit about this functionality, because it's a bit convoluted.

Listing 11.9 Zoom functions for mixed rendering

```
function zoomed() {
  projection.translate(mapZoom.translate()).scale(mapZoom.scale());

  var context = d3.select("canvas").node().getContext("2d");
  context.clearRect(0,0,500,500);
  canvasPath.context(context);

  context.strokeStyle = "black";
  context.fillStyle = "gray";
  context.lineWidth = "1px";
  for (var x in countries.features) {
    context.beginPath();
    canvasPath(countries.features[x]);
    context.stroke()
    context.fill();
  }

  context.strokeStyle = "black";
  context.fillStyle = "rgba(255,0,0,.2)";
  context.lineWidth = 1;
  for (var x in sampleData) {
    context.beginPath();
```

Draws all elements as canvas during zooming

```
    canvasPath(sampleData[x]);
    context.stroke()
    context.fill();
  }
};

function zoomInitialized() {
  d3.selectAll("path.sample")
    .style("display", "none");
  zoomed();
};

function zoomFinished() {
  var context = d3.select("canvas").node().getContext("2d");
  context.clearRect(0,0,500,500);
  canvasPath.context(context)

  context.strokeStyle = "black";
  context.fillStyle = "gray";
  context.lineWidth = "1px";
  for (var x in countries.features) {
    context.beginPath();
    canvasPath(countries.features[x]);
    context.stroke()
    context.fill();
  }

  d3.selectAll("path.sample")
    .style("display", "block")
    .attr("d", svgPath);
};
```

Hides SVG elements when zooming starts

Calls zoomed to draw with canvas the SVG triangles we just hid

Only draws countries with canvas at the end of the zoom

Shows SVG elements when zoom ends

Sets the new position of SVG elements

As a result of this new code, we have a map that uses canvas rendering when users zoom and pan, but SVG rendering when the map is fixed in place and users have the ability to click, mouse over, or otherwise interact with the graphical elements. It's the best of both worlds. The only drawback of this approach is that we have to invest more time making sure our `<canvas>` element and our `<svg>` element line up perfectly, and that our opacity, fill colors, and so on are close enough matches that it's not jarring to the user to see the different modes. I haven't done this in the previous code, so that you can see that the two modes are in operation at the same time, and that's reflected in the difference between the two graphical outputs in figure 11.6.

The kind of pixel-perfect alignment necessary to make the transition from one mode to another, as well as the fastidious color matching also required, isn't something I have the space to explain in this book, but you'll need to do both to make the best interactive information visualization. If you look closely at figure 11.6, you'll notice that the canvas element (on the right) is a pixel or so shifted up and to the left, and that's without testing it in other browsers that may have different default settings for `<canvas>` or `<svg>` or both.

Finally, using canvas and SVG drawing simultaneously may present a difficulty. Say we want to draw a canvas layer over an SVG layer because we want the canvas layer to

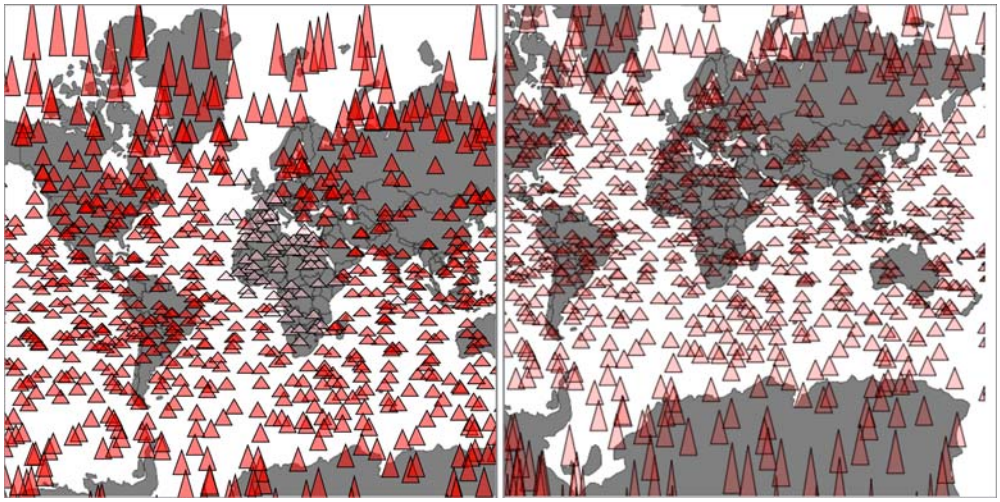


Figure 11.6 The same randomly generated triangles rendered in SVG while the map isn't being zoomed or panned (left) and in canvas while the map is being zoomed or panned (right). Notice that only the SVG triangles have different fill values based on user interaction, because that isn't factored into the canvas drawing code for the triangles on the right.

appear above *some* of our SVG elements visually but below other SVG elements, and we want interactivity on all them. In that case we'd need to sandwich our canvas layer between our SVG layers and set the `pointer-events` style of our canvas layer, as shown in figure 11.7.

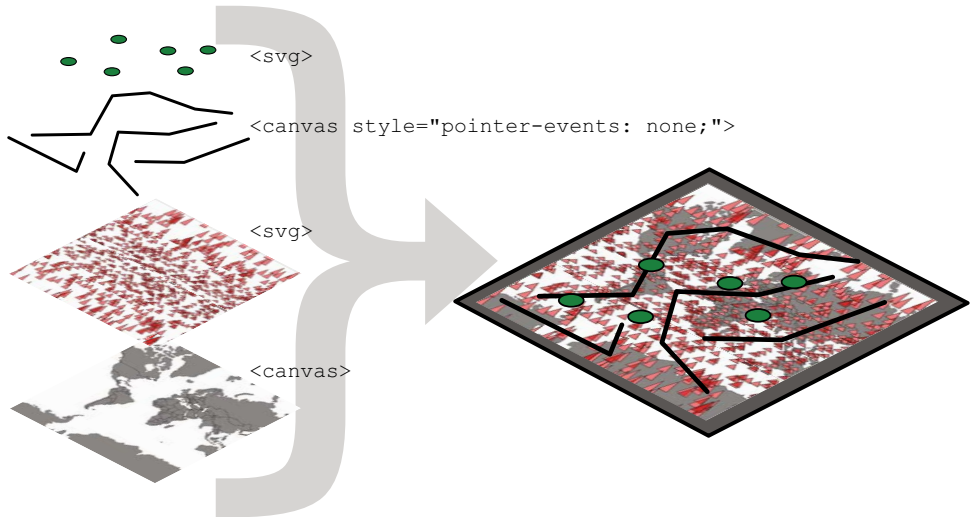


Figure 11.7 Placing interactive SVG elements below a `<canvas>` element requires that you set its `pointer-events` style to `"none"`, even if it has a transparent background, in order to register click events on the `<svg>` element underneath it.

If you add further alternating layers of interactivity but with graphical placement above and below, then you can end up making a `<canvas>` and `<svg>` layer cake in your DOM that's hard to manage and also hard to mentally conceptualize.

11.2 Big network data

It's great that `d3.geo.path` has built-in functionality for drawing geodata to canvas, but what about other types of data visualization? One of the most performance-intensive layouts is the force-directed layout that we dealt with in chapter 6. The layout calculates new positions for each node in your network at every tick. When you use SVG, you need to redraw the network constantly. When I first started working with force-directed layouts in D3, I found that any network with more than 100 nodes was too slow to prove useful. That was a problem because larger networks could still have structure that would benefit from interactivity and animation that needed SVG.

In my own work, I looked at how different small D3 applications hosted on gist.github.com share common D3 functions. D3 coders can understand how different information visualization methods use D3 functions commonly associated with other types of information visualization. You can explore this network along with how D3 Meetup users describe themselves at http://emeeks.github.io/introspect/block_block.html.

To explore these connections, I needed to have a method for dealing with over a thousand different examples and thousands of connections between them. You can see some of this network in figure 11.8. I wanted to show how this network changed based

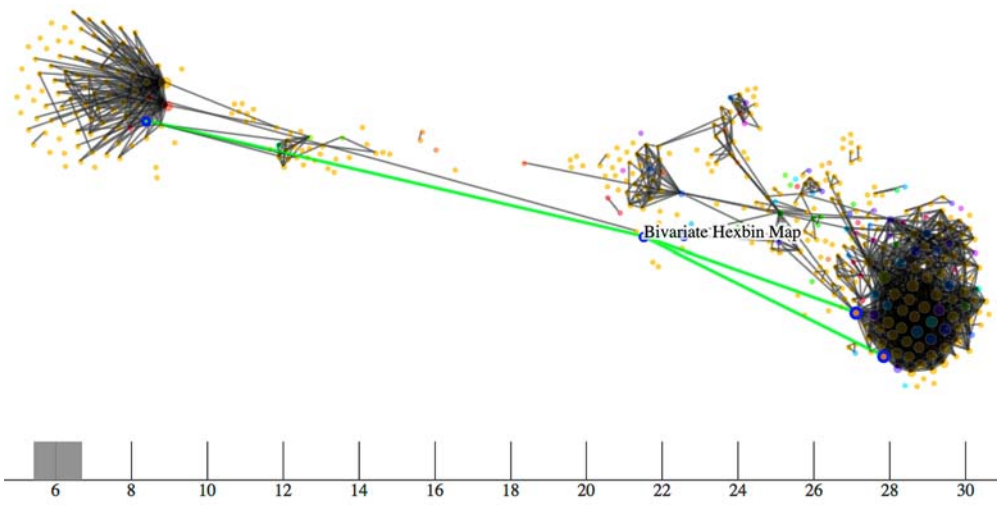


Figure 11.8 A network of D3 examples hosted on gist.github.com that connects different examples to each other by shared functions. Here you can see that the example "Bivariate Hexbin Map" by Mike Bostock (http://bl.ocks.org/mbostock/4330486) shares functions in common with three different examples: "Metropolitan Unemployment," "Marey's Trains II," and "GitHub Users Worldwide." The brush and axis component allows you to filter the network by the number of connections from one block to another.

on a threshold of shared functions, and I also wanted to provide users with the capacity to click each example to get more details, so I couldn't draw the network using canvas. Instead, I needed to draw the network using the same mixed-rendering method we looked at to draw all those triangles on a map. But in this case I used canvas for the network edges and SVG for the network nodes.

Using bl.ocks.org

Although D3 is suitable for building large, complex interactive applications, you often make a smal, single-use interactive data visualization that can live on a single page with limited resources. For these small applications, it's common in the D3 community to host the code on gist.github.com, which is the part of GitHub designed for small applications. If you host your D3 code as a gist, and it's formatted to have an index.html, then you can use bl.ocks.org to share your work with others.

To make your gist work on bl.ocks.org, you need to have the data files and libraries hosted in the gist or accessible through it. Then you can take the alphanumeric identifier of your gist and append it to bl.ocks.org/username/ to serve a working copy for sharing. So, for instance, I have a gist at https://gist.github.com/emeeks/0a4d7cd56e027023bf78 that demonstrates how to do the mixed rendering of a force-directed layout like I described in this chapter. As a result, I can point people to http://bl.ocks.org/emeeks/0a4d7cd56e027023bf78 and they can see the code itself as well as the animated network in action.

Doing this kind of mixed rendering with networks isn't as easy as it is with maps. That's because there's no built-in method to render regular data to canvas as with `d3.geo.path`. If you want to create a similar large network that combines canvas and SVG rendering, you have to build the function manually. First, though, you need data. This time, instead of sample geodata, listing 11.10 shows how to create sample network data.

Building sample network data is easy: you can create an array of nodes and an array of random links between those nodes. But building a sample network that's not an undifferentiated mass is a little bit harder. In listing 11.10 you can see my slightly sophisticated network generator. It operates on the principle that a few nodes are very popular and most nodes aren't (we've known about this principle of networks since grade school). This does a decent job of creating a network with 3000 nodes and 1000 edges that doesn't look quite like a giant hairball.

Listing 11.10 Generating random network data

```
var linkScale = d3.scale.linear()
        .domain([0,.9,.95,1]).range([0,10,100,1000]);

var sampleNodes = d3.range(3000).map(function(d) {
  var datapoint = {};
  datapoint.id = "Sample Node " + d;
  return datapoint;
})
```

This scale makes 90% of the links to 1% of the nodes.

```
var sampleLinks = [];
  var y = 0;
  while (y < 1000) {
    var randomSource = Math.floor(Math.random() * 1000);
    var randomTarget = Math.floor(linkScale(Math.random()));
    var linkObject = {source: sampleNodes[randomSource], target:
        sampleNodes[randomTarget]}
    if (randomSource != randomTarget) {
        sampleLinks.push(linkObject);
    }
    y++;
}
```

The source of each link is purely random.

The target is weighted toward popular nodes.

Don't keep any links that have the same source as target.

With this generator in place, we can instantiate our typical force-directed layout using the code in the following listing, and create a few lines and circles with it.

Listing 11.11 Force-directed layout

```
var force = d3.layout.force()
   .size([500,500])
   .gravity(.5)
   .nodes(sampleNodes)
   .links(sampleLinks)
   .on("tick", forceTick);

d3.select("svg")
   .selectAll("line.link")
   .data(sampleLinks)
   .enter()
   .append("line")
   .attr("class", "link");

d3.select("svg").selectAll("circle.node")
   .data(sampleNodes)
   .enter()
   .append("circle")
   .attr("r", 3)
   .attr("class", "node");

force.start();

function forceTick() {
  d3.selectAll("line.link")
     .attr("x1", function(d) {return d.source.x})
     .attr("y1", function(d) {return d.source.y})
     .attr("x2", function(d) {return d.target.x})
     .attr("y2", function(d) {return d.target.y});

  d3.selectAll("circle.node")
     .attr("cx", function(d) {return d.x})
     .attr("cy", function(d) {return d.y});
};
```

This is all vanilla force-directed layout code like in chapter 6.

For our initial implementation, we render everything in SVG and update the SVG on every tick.

This code should be familiar to you if you've read chapter 6. Generation of random networks is a complex and well-described practice. This random generator isn't going to win any awards, but it does produce a recognizable structure. Typical results are

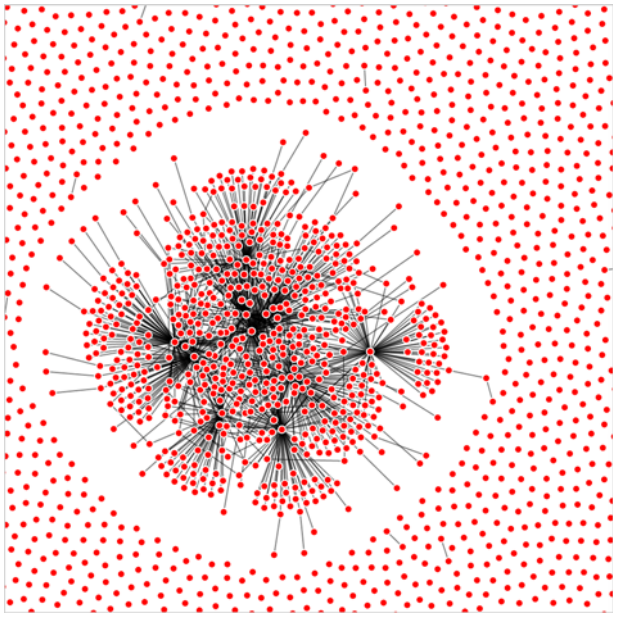


Figure 11.9 A randomly generated network with 3000 nodes and 1000 edges

shown in figure 11.9. What's lost in the static image is the slow and jerky rendering, even on a fast computer using a browser that handles SVG well.

When I first started working with these networks, I thought the main cause of slowdown was calculating the myriad positions for each node on every tick. After all, node position is based on a simulation of competing forces caused by nodes pushing and edges pulling, and something like this, with thousands of components, seems heavy duty. That's not what's taxing the browser, though. Instead, it's the management of so many DOM elements. You can get rid of many of those DOM elements by replacing the SVG lines with canvas lines. Let's change our code so that it doesn't create any SVG `<line>` elements for the links and instead modify our `forceTick` function to draw those links with canvas.

Listing 11.12 Mixed rendering network drawing

```
function forceTick() {
  var context = d3.select("canvas").node()
            .getContext("2d");
  context.clearRect(0,0,500,500);

  context.lineWidth = 1;
  context.strokeStyle = "rgba(0, 0, 0, 0.5)";

  sampleLinks.forEach(function (link) {
     context.beginPath();
     context.moveTo(link.source.x,link.source.y)
     context.lineTo(link.target.x,link.target.y)
     context.stroke();
  });
```

Remember: you always need to clear your canvas.

Draws links as 50% transparent black

Starts each line at the link source coordinates

Draws each link to the link target coordinates

```
    d3.selectAll("circle.node")
      .attr("cx", function(d) {return d.x})
      .attr("cy", function(d) {return d.y});
};
```

Draws nodes as SVG

The rendering of the network is similar in appearance, as you can see in figure 11.10, but the performance improves dramatically. Using canvas, I can draw 10,000 link networks with performance high enough to have animation and interactivity. The canvas drawing code can be a bit cumbersome (it's like the old LOGO drawing code), but the performance makes it more than worth it.

We could use the same method as with the earlier maps to use canvas during animated periods and SVG when the network is fixed. But we'll move on and look at another method for dealing with large amounts of data: quadtrees.

11.3 Optimizing xy data selection with quadtrees

When you're working with a large dataset, one issue is optimizing search and selection of elements in a region. Let's say you're working with a set of data with xy coordinates (anything that's laid out on a plane or screen). You've seen enough examples in this book to know that this may be a scatterplot, points on a map, or any of a number of different graphical representations of data. When you have data like this, you often want to know what datapoints fall in a particular selected region. This is referred to as *spatial search* (and notice that "spatial" in this case doesn't refer to geographic, but rather space in a more generic sense). The quadtree functionality is a spatial version of `d3.nest`, which we used in chapter 5 and chapter 8 and will use again in chapter 12 (available online only) to create hierarchical data. Following the

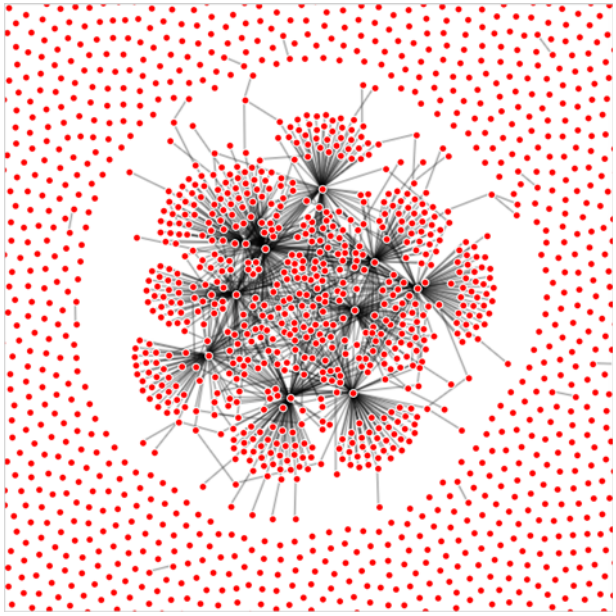


Figure 11.10 A large network drawn with SVG nodes and canvas links

theme of this chapter, we'll get started by creating a big dataset of random points and render them in SVG.

11.3.1 Generating random xy data

Our third random data generator doesn't require nearly as much work as the first two did. In the following listing, all we do is create 3000 points with random x and y coordinates.

Listing 11.13 xy data generator

```
sampleData = d3.range(3000).map(function(d) {
  var datapoint = {};
  datapoint.id = "Sample Node " + d;
  datapoint.x = Math.random() * 500;
  datapoint.y = Math.random() * 500;

  return datapoint;
})

d3.select("svg").selectAll("circle")
  .data(sampleData)
  .enter()
  .append("circle")
  .attr("class", "xy")
  .attr("r", 3)
  .attr("cx", function(d) {return d.x})
  .attr("cy", function(d) {return d.y});
```

Because we know the fixed size of our canvas, we can hardwire this.

As you may expect, the result of this code, shown in figure 11.11, is a bunch of pink circles scattered randomly all over our canvas.

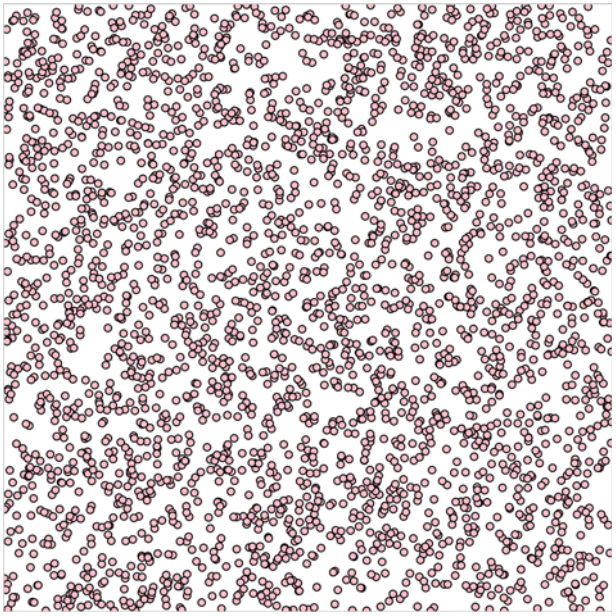


Figure 11.11 3000 randomly placed points represented by pink SVG `<circle>` elements

11.3.2 xy brushing

Now we'll create a brush to select some of these points. Recall when we used a brush in chapter 9 that we only allowed brushing along the x-axis. This time, we allow brushing along both x- and y-axes. Then we can drag a rectangle over any part of the canvas. In listing 11.14, you can see how quick and easy it is to add a brush to our canvas. We'll also add a function to highlight any circles in the brushed region. In this example we use `d3.scale.identity` for our `.x()` and `.y()` selectors. All `d3.scale.identity` does is create a scale where the domain and range are exactly the same. It's useful for times like these when the function operates with a scale but your scale domain directly matches the range of your graphical area.

Listing 11.14 xy brushing

```
var brush = d3.svg.brush()
  .x(d3.scale.identity().domain([0, 500]))
  .y(d3.scale.identity().domain([0, 500]))
  .on("brush", brushed);

d3.select("svg").call(brush)

function brushed() {
  var e = brush.extent();
  d3.selectAll("circle")
  .style("fill", function (d) {
    if (d.x >= e[0][0] && d.x <= e[1][0]
              && d.y >= e[0][1] && d.y <= e[1][1])
    {
      return "darkred";
    }
    else {
      return "pink";
    }
  });
};
```

Because we aren't going to adjust scale settings, we can define them inline.

Tests to see if the data is in our selected area

Colors the points in the selected area dark red

Colors the points outside the selected area pink

With this brushing code, we can now see the circles in the brushed region, as shown in figure 11.12.

This works, but it's terribly inefficient. It checks every point on the canvas without using any mechanism to ignore points that might be well outside the selection area. Finding points within a prescribed area is an old problem that has been well explored. One of the tools available to solve that problem quickly and easily is a quadtree. You may ask, what is a quadtree and what should I use it for?

A *quadtree* is a method for optimizing spatial search by dividing a plane into a series of quadrants. You then divide each of those quadrants into quadrants, until every point on that plane falls in its own quadrant. By dividing the xy plane like this, you nest the points you'll be searching in such a way that you can easily ignore entire quadrants of data without testing the entire dataset.

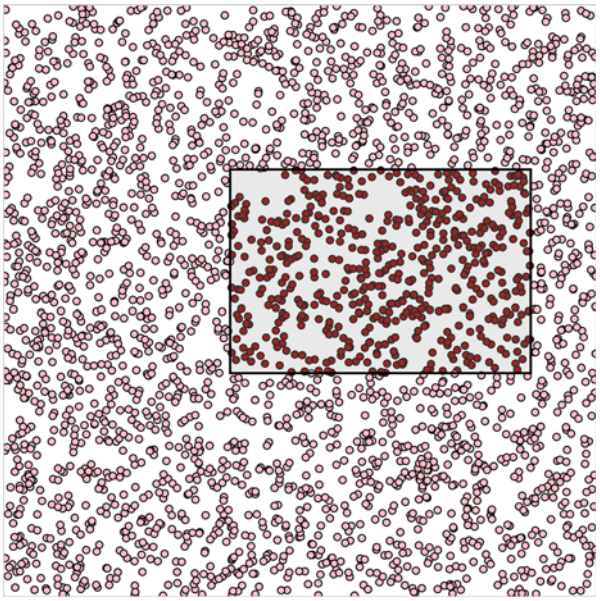


Figure 11.12 Highlighting points in a selected region

Another way to explain a quadtree is to show it. That's what this information visualization stuff is for, right? Figure 11.13 shows the quadrants that a quadtree produces based on a set of point data.

Creating a quadtree with xy data of the kind we have in our dataset is easy, as you can see in listing 11.15. We set the x and y accessors like we do with layouts and other D3 functions.

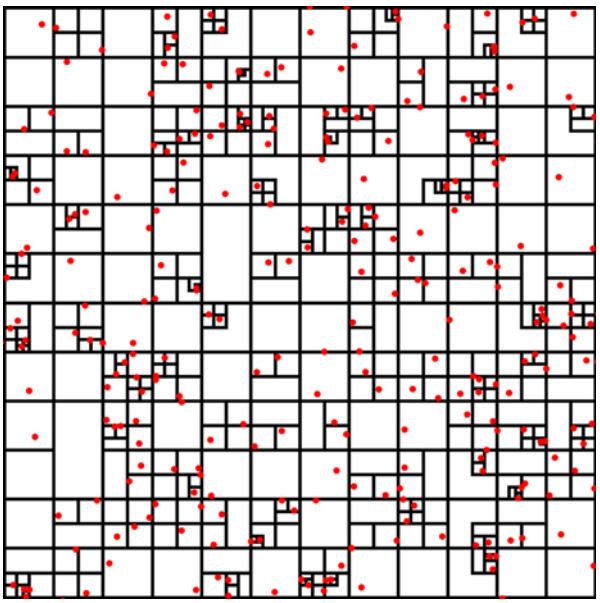


Figure 11.13 A quadtree for points shown in red with quadrant regions stroked in black. Notice how clusters of points correspond to subdivision of regions of the quadtree. Every point falls in only one region, but each region is nested in several levels of parent regions.

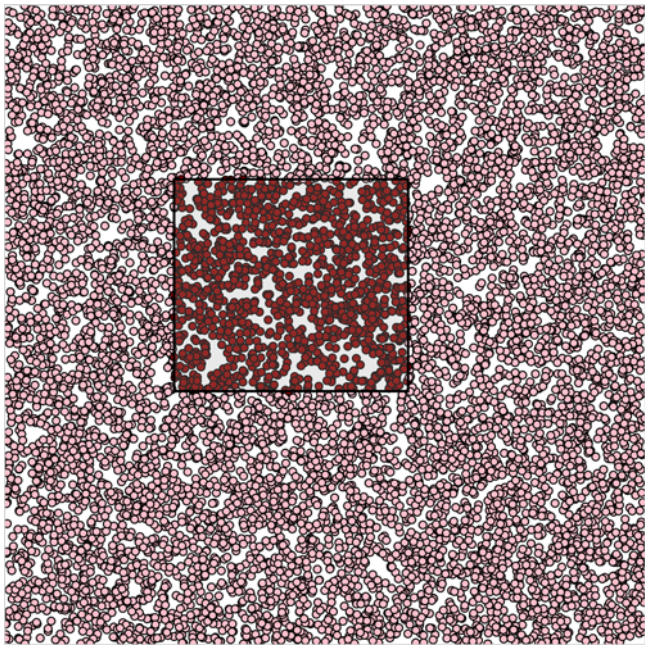


Figure 11.14 Quadtree-optimized selection used with a dataset of 10,000 points

Listing 11.15 Creating a quadtree from xy data

```
var quadtree = d3.geom.quadtree()
    .extent([[0,0], [500,500]])
    .x(function(d) {return d.x})
    .y(function(d) {return d.y});

var quadIndex = quadtree(sampleData);
```

Accessors pointed at our data's xy format

We need to define the bounding box of a quadtree as an array of upper-left and lower-right points.

After creating a quadtree, we create the index by passing our dataset to it.

After you create a quadtree and use it to create a quadtree index dataset like we did with `quadIndex`, you can use that dataset's `.visit()` function for quadtree-optimized searching. The `.visit()` functionality replaces your test in a new brush function, as shown in listing 11.16. First, I'll show you how to make it work in listing 11.16. Then, I'll show you that it *does* work in figure 11.14, and I'll explain *how* it works in detail. This isn't the usual order of things, I realize, but with a quadtree, it makes more sense if you see the code before analyzing its exact functionality.

Listing 11.16 Quadtree-optimized xy brush selection

```
function brushed() {
  var e = brush.extent();

  d3.selectAll("circle")
     .style("fill", "pink")
     .each(function(d) {d.selected = false})
```

Sets all circles to pink, and gives each a selected attribute to designate which are in our selection

```
quadIndex.visit(function(node,x1,y1,x2,y2) {
  if (node.point) {
    if (node.point.x >= e[0][0] && node.point.x <= e[1][0]
        &&node.point.y >= e[0][1] && node.point.y <= e[1][1]) {
      node.point.selected = true;
    }
  }
  return x1 > e[1][0] || y1 > e[1][1] || x2 < e[0][0] || y2 < e[0][1];
})

d3.selectAll("circle")
    .filter(function(d) {
        return d.selected;
    })
    .style("fill", "darkred");
};
```

Calls .visit()

Checks each point to see if it's inside our brush extent and sets selected to true if it is

Checks each node to see if it's a point or a container

Checks to see if this area of the quadtree falls outside our selection

Shows which points were selected

The results are impressive and much faster. In figure 11.14, I increased the number of points to 10,000 and still got good performance. (But if you're dealing with datasets that large, I recommend switching to canvas, because forcing the browser to manage all those SVG elements is going to slow things down.)

How does it work? When you run the `visit` function, you get access to each node in the quadtree, from the most generalized to the more specific. With each node, which we access in listing 11.16 as `node`, you also get the bounds of that node (`x1`, `y1`, `x2`, `y2`). Because nodes in a quadtree can either be the bounding areas or the actual points that generated the quadtree, you have to test if the node is a point and, if it is, you can then test if it's in your brush bounds like we did in our earlier example. The final piece of the `visit` function is where it gets its power, but it's also the most difficult to follow, as you can see in figure 11.15.

The `visit` function looks at every node in a quadtree, unless `visit` returns `true`, in which case it stops searching that particular quadrant and all its child nodes. So you test to see if the node you're looking at (represented as the bounds `x1,y1,x2,y2`) is entirely outside the bounds of your selection area (represented as the bounds `e[0][0]`, `e[0][1]`, `e[1][0]`, `e[1][1]`). You create this test to see if the top of the selection is below the bottom of the node's bounds; if the bottom of the selection is above the top of the node's bounds; if the left side of the selection is to the right of the right side of the node's bounds; or if the right side of the selection is to the left of the left side of the node's bounds. That may seem a bit hard to follow (and sure takes up more time as a sentence than it does as a piece of code), but that's how it works.

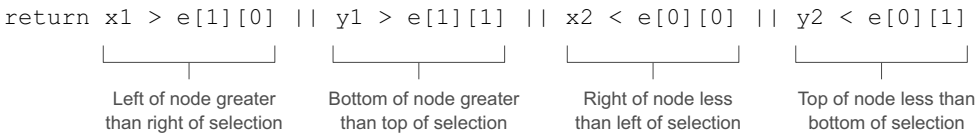


Figure 11.15 The test to see if a quadtree node is outside a brush selection involves four tests to see if it is above, left, right, or below the selection area. If it passes `true` for any of these tests, then the quadtree will stop searching any child nodes.

You can use that `visit` function to do more than optimized search. I've used it to cluster nearby points on a map (http://bl.ocks.org/emeeks/066e20c1ce5008f884eb) and also to draw the bounds of the quadtree in figure 11.13.

11.4 More optimization techniques

You can improve the performance of the data visualization of large datasets in many other ways. Here are three that should give you immediate returns: avoid general opacity, avoid general selections, and precalculate positions.

11.4.1 Avoid general opacity

Whenever possible, use fill-opacity and stroke-opacity or RGBA color references rather than the element opacity style. General element opacity, the kind of setting you get when you use `"style: opacity"`, can slow down rendering. When you use specific fill or stroke opacity, it forces you to pay more attention to where and how you're using opacity.

So instead of

```
d3.selectAll(elements).style("fill", "red").style("opacity", .5)
```

do this:

```
d3.selectAll(elements).style("fill", "red").style("fill-opacity", .5)
```

11.4.2 Avoid general selections

Although it's convenient to select all elements and apply conditional behavior across those elements, you should try to use `selection.filter` with your selections to reduce the number of calls to the DOM. If you look at the code in listing 11.16, you'll see this general selection that clears the selected attribute for all the circles and sets the fill of all the circles to pink:

```
d3.selectAll("circle")
.style("fill", "pink")
.each(function(d) {d.selected = false})
```

Instead, clear the attribute and set the fill color of only those circles that are currently set to the selection. This limits the number of costly DOM calls:

```
d3.selectAll("circle")
.filter(function(d) {return d.selected})
.style("fill", "pink")
.each(function(d) {d.selected = false})
```

If you adjust the code in that example, the performance is further improved. Remember that manipulating DOM elements, even if it's changing a setting like fill, can cause the greatest performance hit.

11.4.3 Precalculate positions

You can also precalculate positions and then apply transitions. If you have a complex algorithm that determines an element's new position, first go through the data array and calculate the new position. Then append the new position as data to the datapoint of the element. After you've done all your calculations, select and apply a transition based on the calculated new position. When you're calculating complex new positions and applying those calculated positions to a transition of a large selection of elements, you can overwhelm the browser and see jerky animations.

So, instead of

```
d3.selectAll(elements)
.transition()
.duration(1000)
.attr("x", newComplexPosition);
```

do this:

```
d3.selectAll(elements)
.each(function(d) {d.newX = newComplexPosition(d)});

d3.selectAll(elements)
.transition()
.duration(1000)
.attr("x", function(d) {return d.newX});
```

11.5 Summary

In this chapter, we looked at a few ways to deal with large datasets, and by necessity touched on methods for generating those datasets. Specifically, we looked at

- Generating random geodata
- Using the `.context` function of `d3.geo.path` to draw map features using canvas
- Using zoom's start and end functionality to render elements in canvas or SVG
- Generating random network data
- Drawing network lines in canvas
- Generating random xy data
- Creating an xy brush
- Highlighting selected features
- Building a quadtree
- Using a quadtree for optimized spatial search

D3.js is a JavaScript library that allows data to be represented graphically on a web page. Because it uses the broadly supported SVG standard, D3 allows you to create scalable graphs for any modern browser. You start with a structure, dataset, or algorithm and programmatically generate static, interactive, or animated images that responsively scale to any screen.

D3.js in Action introduces you to the most powerful web data visualization library available and shows you how to use it to build interactive graphics and data-driven applications. You'll start with dozens of practical use cases that align with different types of charts, networks, and maps using D3's out-of-the-box layouts. Then, you'll explore practical techniques for content design, animation, and representation of dynamic data—including interactive graphics and live streaming data.

What's inside

- Interacting with vector graphics
- Expressive data visualization
- Creating rich mapping applications
- Prepping your data
- Complete data-driven web apps in D3

Readers need basic HTML, CSS, and JavaScript skills. No experience with D3 or SVG is required.

index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

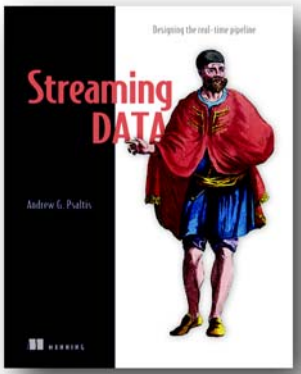
Save 50% on these selected books—eBook, pBook, and MEAP. Just enter ***apwebiot50*** *in the Promotional Code box when you check out. Only at manning.com.*

We hope you enjoyed this sneak peek at what the web can do for the Internet of Things. We hope you understand why the Web of Things is a powerful concept that can unleash the power of the IoT and why you should start acquiring the necessary skillset today. If you enjoyed the journey, we encourage you to get the full books, starting with ours: Building the Web of Things!

Building the Web of Things

by Dominique D. Guinard and Vlad M. Trifa

ISBN: 9781617292682
375 pages
$34.99
March 2016

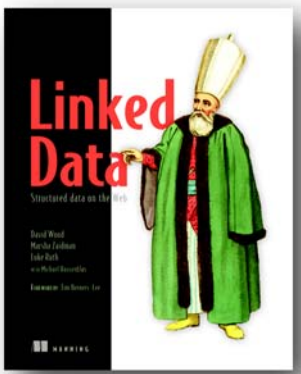


Streaming Data

Designing the real-time pipeline

by Andrew G. Psaltis

ISBN: 9781617292286
300 pages
$49.99
March 2016

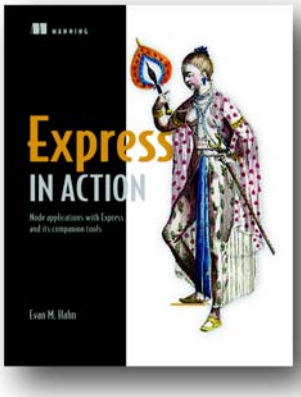


Linked Data

Structured data on the Web

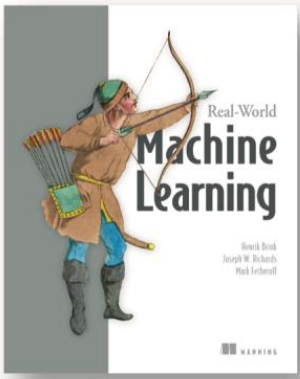
by David Wood, Marsha Zaidman, Luke Ruth, and Michael Hausenblas

ISBN: 9781617290398
336 pages
$49.99
February 2012



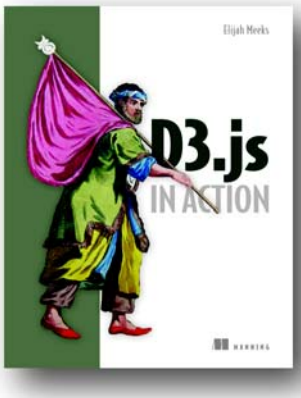
Express in Action
Node applications with Express and its companion tools
by Evan M. Hahn

ISBN: 9781617292422
245 pages
$39.99
January 2016



Real-World Machine Learning
by Henrik Brink, Joseph W. Richards, and Mark Fetherolf

ISBN: 9781617291920
400 pages
$49.99
April 2016



D3.js in Action
by Elijah Meeks

ISBN: 9781617292118
352 pages
$44.99
February 2015