SAMPLE CHAPTER

building the web of things

With examples in Node.js and Raspberry Pi

Dominique D. Guinard
Vlad M. Trifa

Building the Web of Things

With examples in Node.js and Raspberry Pi

by Dominique D. Guinard
Vlad M. Trifa

Sample Chapter 2

brief contents

2 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, WebSockets, and browsers
- Working with REST APIs to consume JSON data
- Learning about the notion of web semantics
- Creating your first physical mashup

Before we dive head first into the Web of Things architecture and show how to implement 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 applications that interact with 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 to buy a real device, we connected our own device to the web so you can access it from your computer over the Web. 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. 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.

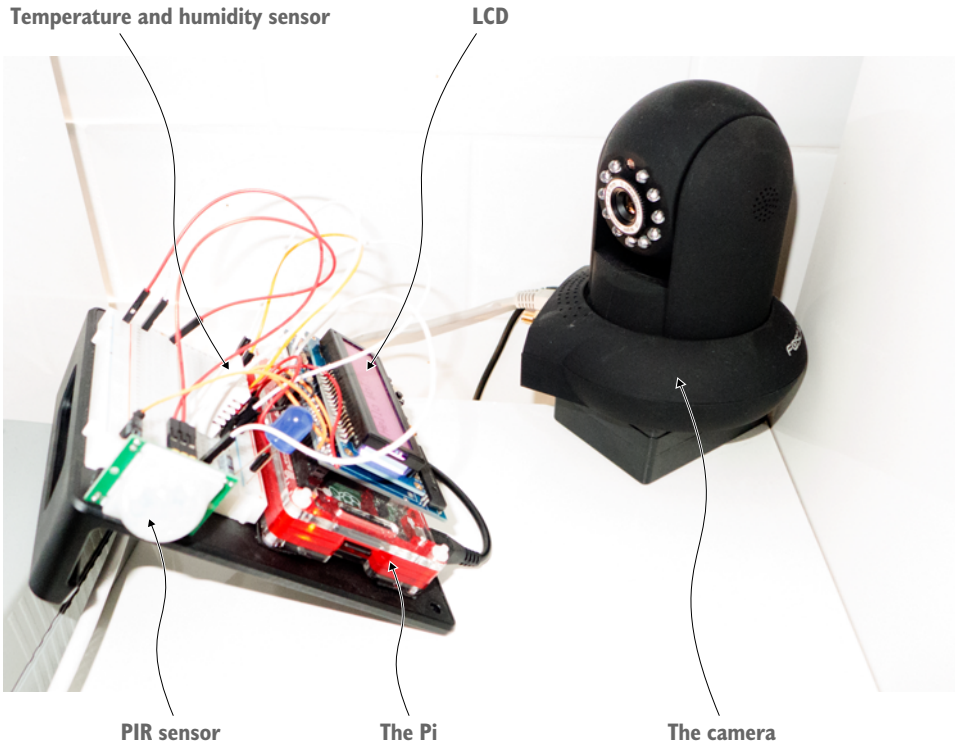


Figure 2.1 The Raspberry Pi and webcam you are accessing as they are set up in our London office

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 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 our setup, the Pi acts as a gateway to various sensors or devices attached to it, so you can interact with those resources through the Web. Gateways are described in detail in chapter 7, but

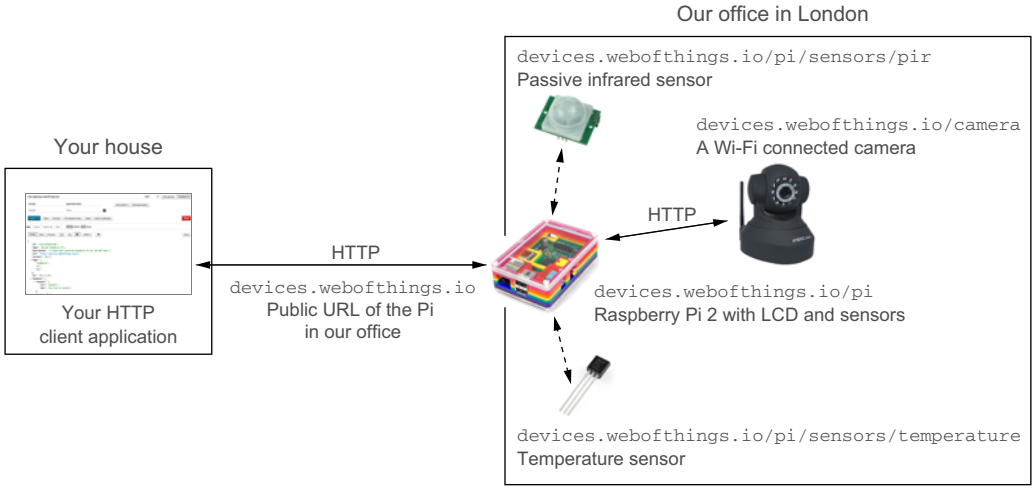


Figure 2.2 The setup of devices and sensors used in the examples of this chapter

for now just remember that the Pi runs a web server that allows you to access those resources over the Web, as shown in figure 2.2..

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 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 in it you'll find a few folders—one for each chapter. The exercises in this chapter are located in the folder

chapter2-hello-wot/client. If you're wondering about the code for the server, worry not! You'll learn how to build this in the rest of the book.

How to get the code examples in this chapter

We use the GitHub[a] service to synchronize code between our computer and our Pi. 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's plenty of information about this 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.

[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 on a few links in your browser. The first point we want to illustrate is that on the Web of Things, devices can 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 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 see 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). Enter the following URL in your browser: http://devices.webofthings.io.

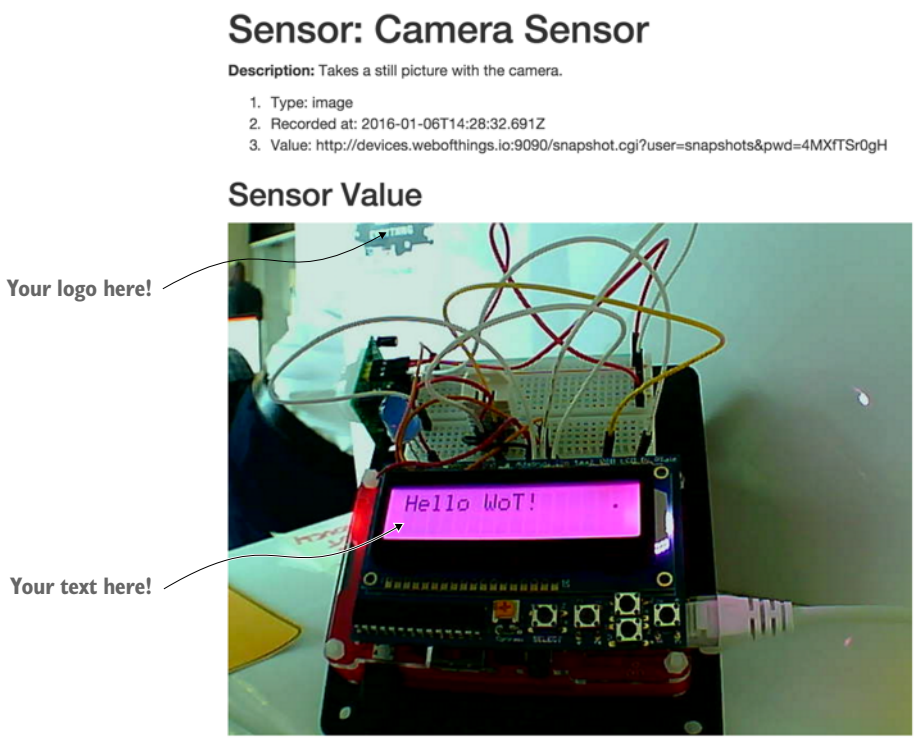


Figure 2.3 The web page of the camera used in our setup. The image is a live screenshot taken by the camera.

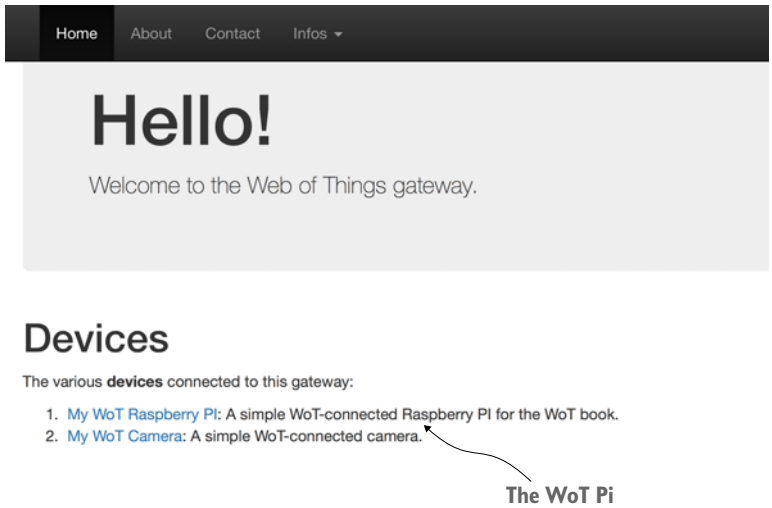


Figure 2.4 The HTML homepage of the gateway of our WoT device. The two hyperlinks at the bottom of the page allow you to access the pages of the devices connected to the gateway.

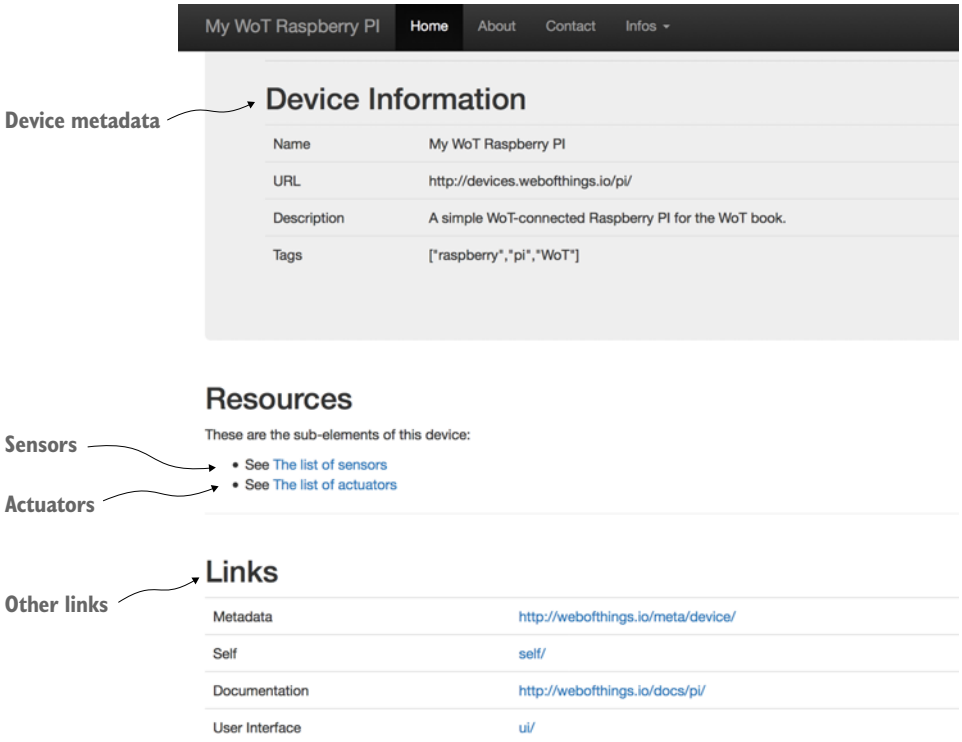


Figure 2.5 The homepage of the Raspberry Pi. Here, you can use the links at the bottom to browse and explore the various resources offered by this device; for example, its sensors and actuators.

This 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 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.

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.

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.

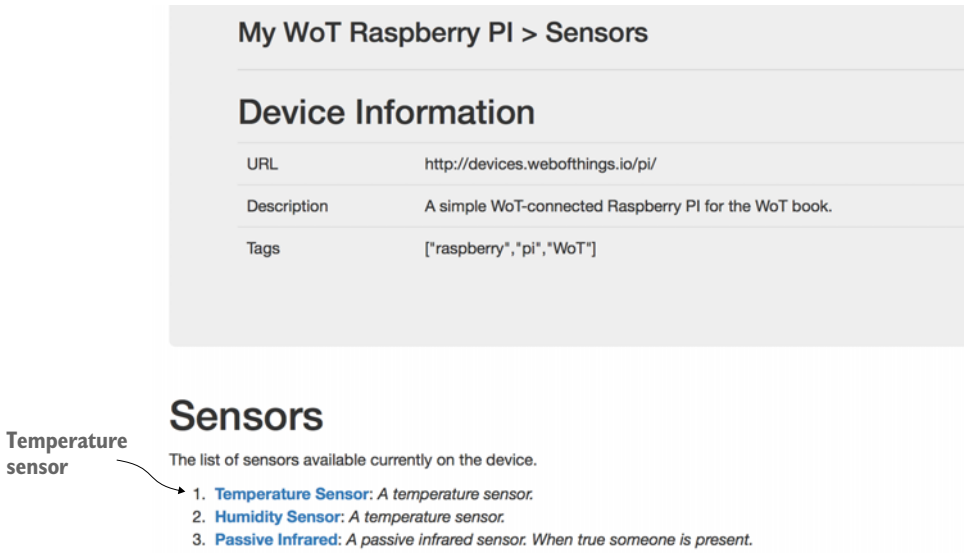


Figure 2.6 The list of sensors on the Pi. You can click each of them and see the latest known value for each.

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/` in the URL and you'll be taken directly to the Sensors page of the camera: http://devices.webofthings.io/camera/sensors; see figure 2.7.

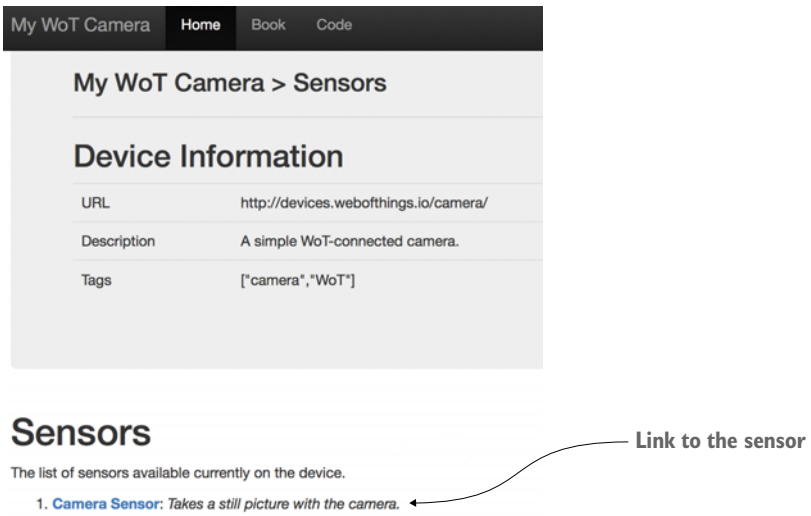


Figure 2.7 The sensors on the camera. There's only one sensor here, which is the current image.

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 sensors, go to the actuators list of the Pi and open the Actuator Details page (see 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, and 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 resources of a device (sensors, actuators, and so on) and how to interact with that device 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 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 illustrate how the web makes it easy to support both humans and applications by showing you what another device or 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 you could 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 go ahead and install it.

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

[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/

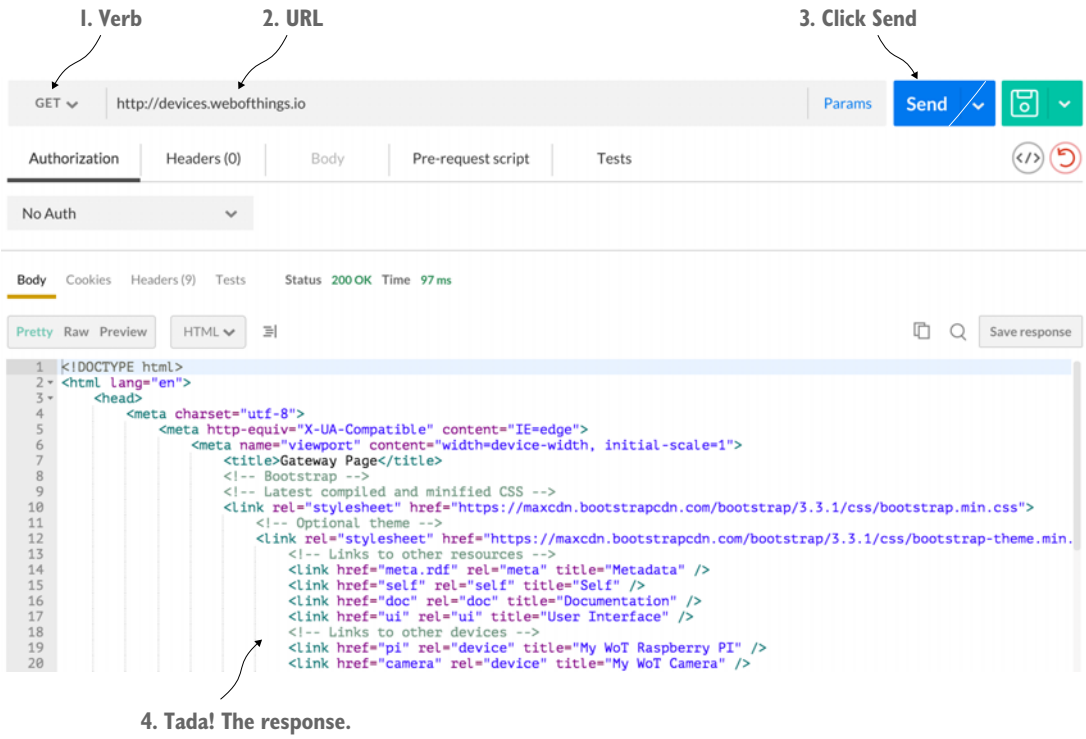


Figure 2.8 Getting the root page of the gateway using the Postman web client. The request is an HTTP GET (1) on the URL of the gateway (2). The response body will contain an HTML document (4).

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 you'll 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 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 telling the HTTP server, "Hey, if you can, please return me

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 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 the web page you've retrieved before, but this time with only the content and no visual elements (that is, the HTML code).

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

```
{
  "pi": {
    "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"
      }
    }
  },
  "camera": {
    [ ... description of the camera object... ]
  }
}
```

In this JSON document, you can see two first-level elements (`pi` and `camera`) that represent the two devices attached to the gateway, as well as a few details about them, such as their URL, name, ID, and 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 so it points back 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.

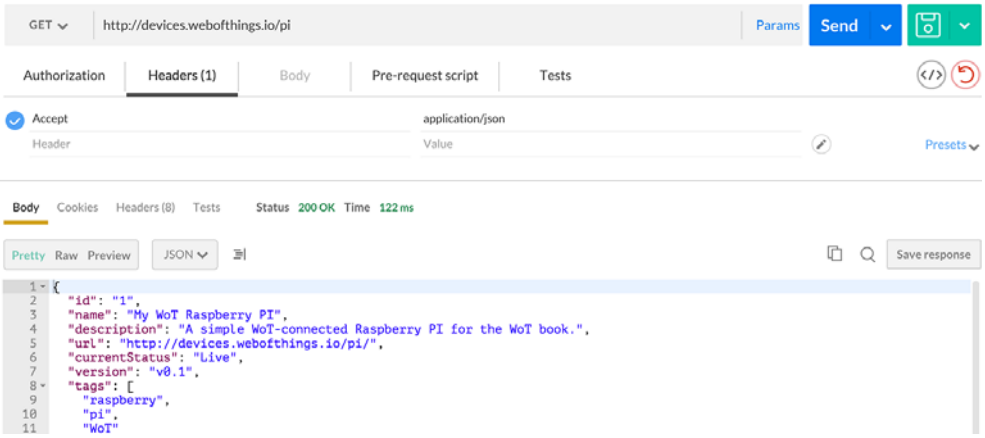


Figure 2.10 Getting the JSON representation of the Raspberry Pi. The JSON payload contains metadata about the device as well as links to its sub-resources.

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, 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.

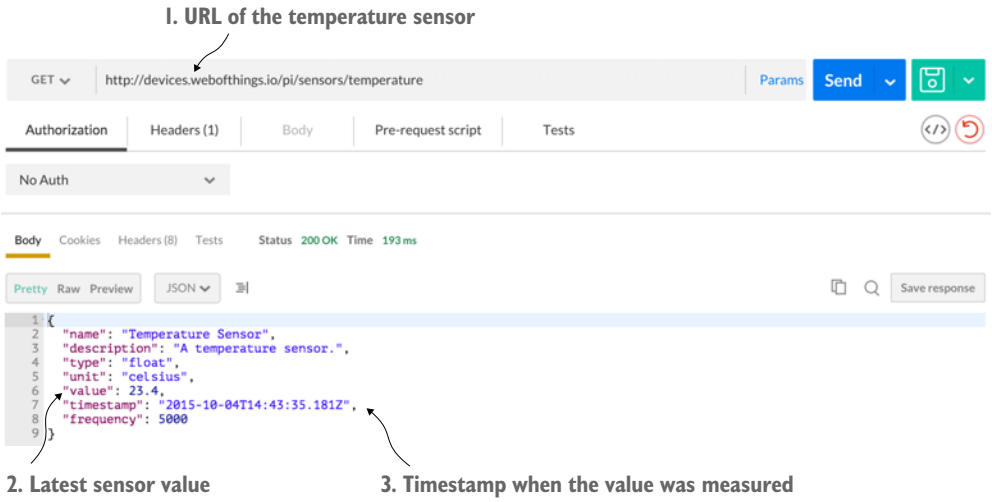


Figure 2.11 Retrieve the temperature sensor object from the Raspberry Pi. You can see the latest reading (23.4 degrees Celsius) and when it took place (at 14:43 on October 4, 2015).

STEP 4—GET DETAILS OF A SINGLE SENSOR

Finally, you'll get the details of a specific sensor, so 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.

You will get detailed information about the temperature sensor, in particular the latest value that was read (the value field). If you only want to retrieve the sensor value, you can append /value to the URL of the sensor to retrieve it, which also work for other sensors:

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

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 different encoding formats (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. This is one example of how simple—yet powerful—web technologies can be. Thanks to immensely popular web standards such as HTTP and URLs, it becomes straightforward to interact with the real world from any web browser. You'll learn much more about these concepts 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.[3] This page 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[4] 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

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

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

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

When the response arrives, this function is called.

[3] 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.

[4] 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 look for the console that appears below where you can see the HTML code of the current page. 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 the current value of the sensor—for example, a graph of all readings in the last hour or week. 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,[5] a nice and lightweight JavaScript library for displaying all sorts of charts and graphs. See our annotated code sample in the next listing.

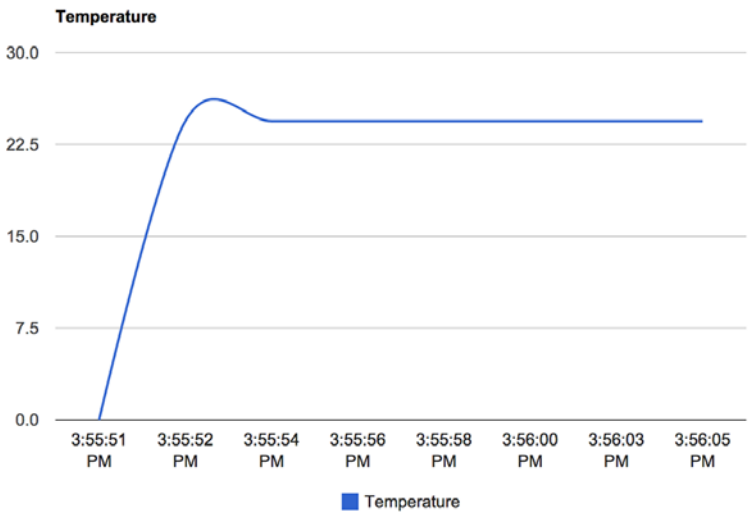


Figure 2.12 This graph gets a new value every few seconds from the device and is updated automatically.

[5] 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 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 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. Have a look at the new code shown in the next listing.

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 WebSocket 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 will be given 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 on what you did in 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 were able 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 that 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 be 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: making things from the real world programmable and universally accessible by bringing them closer to the masses of web developers, where a lot of today's innovations are happening.

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, and in chapter 7 we'll discover how to use gateways to bring other protocols and systems closer to the 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 send some 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 specifies how long the text will be displayed on our LCD. Use Postman to get the JSON object that describes the display actuator by

entering the URL shown in the last paragraph, 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 was 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. Go ahead; open this page, and you'll see the latest picture of the camera you saw in figure 2.3 (to see the latest image, 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

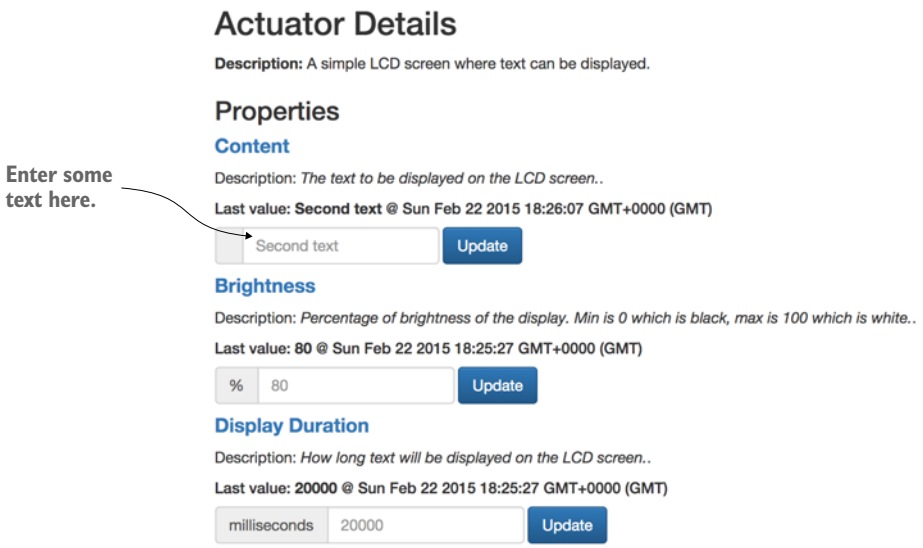


Figure 2.13 The details of the LCD actuator, with the various properties that you can set, for example, the text that should be displayed next on the device

you 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.

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 on the LCD screen of the Pi in our office. Please keep it courteous, and because the API of our Pi is open to the public, we decline all responsibility 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 what text to display. This method works well for a basic website. Unfortunately, what you don't see behind the scenes is that web browsers do not submit (nor do they make it possible to submit) data to the server using a JSON payload body (as you could easily do with Postman previously) 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, 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, shown in the following listing.

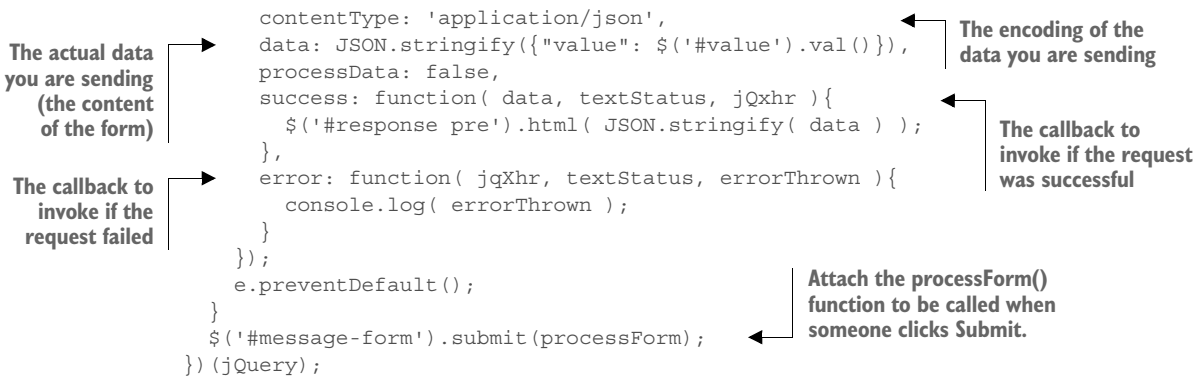
Listing 2.6 Send an HTTP POST with JSON payload from a form

```
(function($){ function processForm(e){
  $.ajax({
    url: 'http://devices.webofthings.io/pi/actuators/display/content/',
    dataType: 'json',
    method: 'POST',
```

The URL the request will be sent to

The format of the data you expect to get

The HTTP verb this request will send




```
      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);
```

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, 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 data 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 over the web 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 the 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 happen to know (by some magic trick) the root URL of all Web of Things–compatible devices around you, how can 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, 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 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). Click Browse This Device.

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,

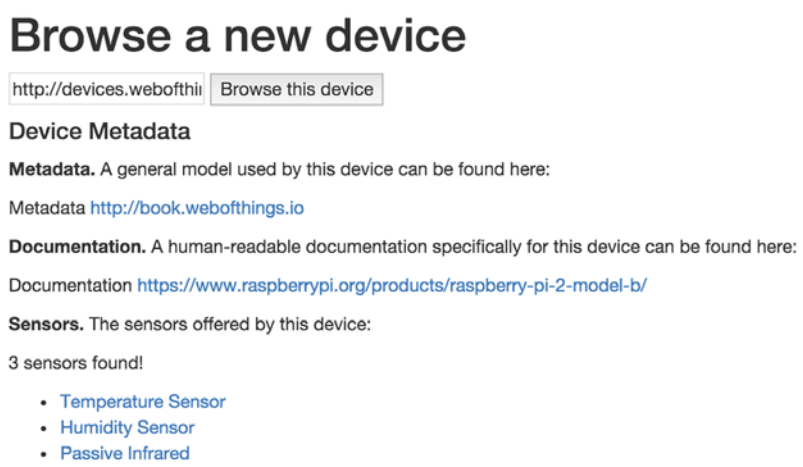


Figure 2.15 A mini-browser that parses your device metadata and displays the results

along with link to the documentation for this device. First, let's look at the HTML code to display the report, as shown in the next listing.

Listing 2.7 A basic device browser

```
<form id="message-form">
    <input type="text" id="host" name="host" value="http://devices.webofth-
   ings.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 in the following listing.

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 links 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.

Loop through all 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);
```

Display the list in the HTML.

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 code:

```
"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 would 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 (such as documentation) 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 unlock the full potential of the Web of Things, we must be able to define all the details about an object using a single data model with clear semantics that all machines and applications can understand without any room for ambiguity. 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

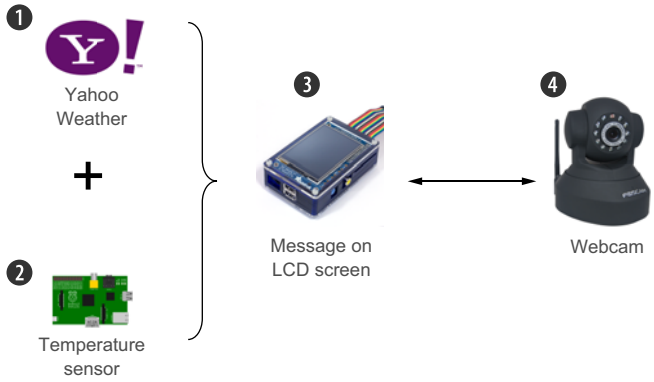


Figure 2.16 A physical mashup application. First (1), you retrieve the local temperature from Yahoo Weather and then the remote temperature from the sensor attached to our Pi (2). You compare it with the temperature in London and send the results to an LCD screen (3). When the screen displays the text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

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. 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 see 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 illlustration of this process.

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 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));
      });
    });
  }
```

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.

```
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');
});
```

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 did in section 2.3.1.

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 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, 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 or WebSocket 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.

We hope you enjoyed this first encounter with the Web of Things enough to read the ensuing chapters and learn how to implement these concepts on your own device. In the next chapters, we'll look at how to implement JavaScript on devices and we'll provide a short and sweet introduction to Node.js. Then, we'll look into configuring your own device and making it fit for the Web of Things. We'll show you how to create and deploy a Node.js application on a Raspberry Pi device, and you'll be able to create your first web-connected device and adapt these examples for your own use case.

WEB DEVELOPMENT/HARDWARE

building the web of things

Dominique D. Guinard
Vlad M. Trifa

Because the Internet of Things is new, there still is no universal application protocol. Fortunately, the IoT can take advantage of the web, where IoT protocols connect applications thanks to universal and open APIs.

Building the Web of Things is a comprehensive technical guide to using cutting-edge web technologies to build the IoT. This step-by-step book teaches you how to use web protocols to connect real-world devices to the web, including the Semantic and Social Webs. Along the way you'll gain vital concepts as you follow instructions for making Web of Things devices. By the end, you'll have the practical skills you need to implement your own web-connected products and services.

What's Inside

- Introduction to IoT protocols and devices
- Implement standard REST APIs with Node.js on embedded systems
- Integrate protocols like MQTT and CoAP to the Web of Things
- Build a web-based smart home with HTTP and WebSocket
- Compose physical mashups with EVRYTHNG, Node-RED, and IFTTT

For both seasoned programmers and those with only basic programming skills.

Dominique Guinard and ***Vlad Trifa*** pioneered the Web of Things and cofounded EVRYTHNG, a large-scale IoT cloud powering billions of Web Things.

"A fantastic set of ideas and a great addition to the IoT toolkit."
—Mike Kuniavsky
Innovation Services at PARC

"IoT needs an application layer, and leveraging the web is the right thing to do! This terrific book will show you how to get there in a few weeks."
—Sanjay Sarma
AutoID Labs, MIT

"Dom and Vlad are thought leaders in IoT, focused on how to achieve results in practice."
—Andy Chew, Cisco UK

"A complex subject covered in detail from beginning to end ... very readable too!"
—Steve Grey-Wilson
Thingworx, A PTC Business

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/building-the-web-of-things

ISBN-13: 978-1-61729-268-2
ISBN-10: 1-61729-268-0
53499
9 781617 292682

MANNING US $34.99 | Can $40.99 (including eBook)