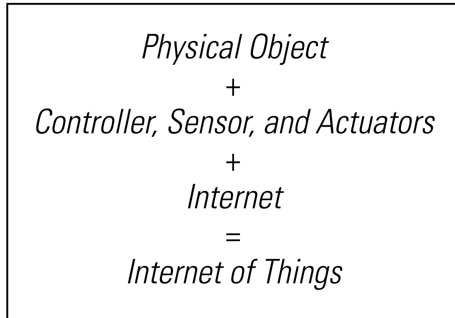
Designing the
Internet
of things
Adrian McEwen & Hakim Cassimally

7 PROTOTYPING ONLINE COMPONENTS

YOU SAW IN Chapter 2, "Design Principles for Connected Devices", how Internet of Things devices are "magical" objects: a physical thing with an embedded controller and sensors that allow it to do clever things that a merely solid or mechanical object couldn't. Even with just these two components, you can already see some magical objects, such as air fresheners that pump fragrance into the room only when they sense that someone has walked past. But the phrase "Internet of Things" does suggest that the Internet is also part of the equation.

You can easily see that each component has a critical part to play. The physical, designed object ties into the design principles (as you saw in Chapter 2) of context, glanceability, and so on. The controller and associated electronics allow it to sense and act on the real world. The Internet adds a dimension of communication. The network allows the device to inform you or others about events or to gather data and let you act on it in real time. It lets you aggregate information from disparate locations and types of sensors. Similarly, it extends your reach, so you can control or activate things from afar, and it allows the online world to bleed out into the physical realm in new and interesting ways.



The key components of the Internet of Things.

So, sensor devices which record temperature might write that data to Xively. Notification devices like Bubblino blow bubbles in response to tweets on Twitter. In fact, although you have seen in Chapter 2 that one design principle is that Things should be "first class citizens" of the Internet, they do seem to be currently tied to particular websites or services. There is a good reason for this: unlike a general-purpose device, such as a computer, tablet, or phone, the physical object is designed for a purpose and doesn't necessarily have a keyboard and screen to let it easily change its configuration.

In the near future, devices will most likely use standardized protocols to speak to each other and to other apps and computers on your local or personal network. For now, though, in most of the examples we look at, each device is tied to a single web service. Although you've looked at existing services (Xively, Twitter), you might benefit from creating your own. For a personal project, creating such a service may not be important, but if you're developing a product to sell, you will want to be in control of the service—otherwise, you may have to recall every device to reprogram any time it is discontinued, changes terms and conditions, making your use of it abusive, or changes its API, making your code stop working. In fact, even Bubblino runs via a service at `http://bubblino.com` which allows users to customise their Bubblino to search for particular words and gives Adrian the flexibility to route all the Bubblini to a different service or API if anything changes.

GETTING STARTED WITH AN API

The most important part of a web service, with regards to an Internet of Things device, is the Application Programming Interface, or API. An API is a way of accessing a service that is targeted at machines rather than people. If you think about your experience of accessing an Internet service, you might follow a number of steps. For example, to look at a friend's photo on Flickr, you might do the following:

1. Launch Chrome, Safari, or Internet Explorer.
2. Search for the Flickr website in Google and click on the link.
3. Type in your username and password and click “Login”.
4. Look at the page and click on the “Contacts” link.
5. Click on a few more links to page through the list of contacts till you see the one you want.
6. Scroll down the page, looking for the photo you want, and then click on it.

Although these actions are simple for a human, they involve a lot of looking, thinking, typing, and clicking. A computer can’t look and think in the same way. The tricky and lengthy process of following a sequence of actions and responding to each page is likely to fail the moment that Flickr slightly changes its user interface. For example, if Flickr rewords “Login” to “Sign in”, or “Contacts” to “Friends”, a human being would very likely not even notice, but a typical computer program would completely fail. Instead, a computer can very happily call defined commands such as `login` or `get picture #142857`.

MASHING UP APIS

Perhaps the data you want is already available on the Internet but in a form that doesn’t work for you? The idea of “mashing up” multiple APIs to get a result has taken off and can be used to powerful effect. For example:

- Using a mapping API to plot properties to rent or buy—for example, Google Maps to visualise properties to rent via Craigslist, or Foxtons in London showing its properties using Mapumental.
- Showing Twitter trends on a global map or in a timeline or a charting API.
- Fetching Flickr images that are related to the top headlines retrieved from *The Guardian* newspaper’s API.

Do You Need a Full API?

For a personal project, you may be best off starting by targeting an existing service, such as Twitter or Xively, as mentioned already, or Transport for London’s Rental Bike availability API, or mapme.at.

Perhaps, as with Bubblino, you will expand that service later to a simple configuration and wrapping API. But if the data you want to interact with doesn’t yet exist, this may represent an opportunity to create a new service that could be generally useful.

Some of the more visible and easy-to-use APIs want to embed your data within them—for example, the Google Maps API. This means that they are ideal to use within a web browser, but you aren't in control of the final product, and there might be limited scope for accessing them from a microcontroller.

SCRAPING

In many cases, companies or institutions have access to fantastic data but don't want to or don't have the resources or knowledge to make them available as an API. While you saw in the Flickr example above that getting a computer to pretend to be a browser and navigate it by looking for UI elements was fragile, that doesn't mean that doing so is impossible. In general, we refer to this, perhaps a little pejoratively, as "screen-scraping". Here are a few examples:

- Adrian has scraped the Ship AIS system (`www.shipais.com/`, whose data is semi-manually plotted by shipping enthusiasts) to get data about ships on the river Mersey, and this information is then tweeted by the `@merseyshipping` account (`www.mcqn.com/weblog/connecting_river_mersey_twitter`). He says of the project that it is a way to "connect the river to the Internet", so although this doesn't have an Internet-connected "thing" as such, it arguably enters into the realm of the Internet of Things.
- The Public Whip website (`www.publicwhip.org.uk/`) is made possible by using a scraper to read the Hansard transcripts of UK government sessions (released as Word documents). With the resultant data, it can produce both human- and machine-readable feeds of how our elected representatives vote.
- As well as other tools for working with data online, the ScraperWiki site (`https://scraperwiki.com`) has an excellent platform for writing scrapers, in a number of dynamic programming languages, which collate data into database tables. Effectively, it provides infrastructure for "Mechanize" scripts that you could run on your own computer or server but allows you to outsource the boring, repetitive parts to ScraperWiki. Their CEO, Francis Irving, used this for an Internet of Things project of his own. He scrapes the Liverpool council website page to find out when his recycling bin is due to be collected. His Binduino device (`https://github.com/frabcus/binduino`), an Arduino plus custom electronics, checks the result regularly and illuminates some electroluminescent wire to make his recycling bin glow when he needs to take it out.

LEGALITIES

Screen-scraping may break the terms and conditions of a website. For example, Google doesn't allow you to screen-scrape its search pages but does provide an API. Even if you don't think about legal sanctions, breaking the terms and conditions for a company like Google might lead to its denying you its other services, which would be at the very least inconvenient.

Other data is protected by copyright or, for example, database rights. One project we discussed for the book would be a scraper that read football fixtures and moved a "compass" to point to the relative direction that your team was playing in. However, certainly in the UK, fixtures lists are copyrighted, and the English and Scottish football leagues have sued various operators for not paying them a licensing fee for that data (`http://www.out-law.com/page-10985`). For a personal pet project, creating such a scraper shouldn't be a huge issue but might reduce the viability of a commercial product (depending on whether the licensing costs are sensible business costs or prohibitive).

Alternative sources of information often are available. For example, you could use OpenStreetMap instead of Google Maps. The UK postcode database is under Crown Copyright, but there are other, perhaps partial, crowdsourced versions.

For additional discussions about the legal aspects of data, among other things, see Chapter 9, "Business Models".

WRITING A NEW API

Assuming the data you want to play with isn't available or can't be easily mashed up or scraped using other existing tools and sources, perhaps you want to create an entirely new source of information or services. Perhaps you plan to assemble the data from free or licensed material you have and process it. Or perhaps your Internet-connected device can populate this data!

To take you through the process of building your own API, we use an example project, Clockodillo. This is an Internet of Things device that Hakim built to help him use the Pomodoro time management technique (`www.pomodorotechnique.com/`).

With the Pomodoro system you split your tasks into 25-minute chunks and use a kitchen-timer to help track how long the task takes you, and to encourage you to block out distractions during each 25-minute block.

Clockodillo explores how the Internet of Things might help with that: connecting the kitchen-timer to the Internet to make the tracking easier while keeping the simplicity of the physical twist-to-set timer for starting the clock and showing progress as it ticks down.

By the end of the chapter, you end up with the skeleton of an actual API that the timer device connects to.

Although the process of designing a web application to be used on a browser can mix up the actions that users will accomplish with the flows they will take to navigate through the application, writing the back-end API makes you think much more in terms of the data that you want to process.

As the legendary software engineer Frederick P. Brooks, Jr. wrote:

> *Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious.*
>
> —The Mythical Man-Month: Essays on Software Engineering *(Addison-Wesley, 1975)*

When you know what data you have, what actions can be taken on it, and what data will be returned, the flows of your application become simple. This is a great opportunity to think about programming without worrying (at first) about the user interface or interactions. Although this might sound very different from writing a web application, it is actually an ideal way to start: by separating the business problem from the front end, you decouple the model (core data structure) from the view (HTML/JavaScript) and controller (widgets, form interaction, and so on). If you've programmed in one of the popular MVC frameworks (Ruby on Rails, Django, Catalyst, and so on), you already know the advantage of this approach.

The best news is, if you start designing an API in this way, you can easily add a website afterwards, as you will see in the upcoming "Going Further" section.

CLOCKODILLO

As we saw earlier, Clockodillo is an Internet-connected task timer. The user can set a dial to a number of minutes, and the timer ticks down until completed. It also sends messages to an API server to let it know that a task has been started, completed, or cancelled.

A number of API interactions deal precisely with those features of the physical device:

- Start a new timer
- Change the duration of an existing timer
- Mark a timer completed
- Cancel a timer

Some interactions with a timer data structure are too complicated to be displayed on a device consisting mostly of a dial—for example, anything that might require a display or a keyboard! Those could be done through an app on your computer or phone instead.

- View and edit the timer's name/description

And, naturally, the user may want to be able to see historical data:

- Previous timers, in a list
    - Their name/description
    - Their total time and whether they were cancelled

Assuming you plan to build more than just one device, you need to have some form of identifying the device. We come back to this interesting topic when we look at scaling up to production in Chapter 10. For now, assume that each device will send some identifying token, such as a MAC address. (As you saw in Chapter 3, this is a unique code that every networked chip has.)

So the user will somehow identify himself with the server, after which all the preceding actions will relate just to a given user ID.

SECURITY

Does it look as though we're missing something? If you're jumping up and down shouting "What about security?" give yourself a pat on the back. How important security is depends a lot on how sensitive the information being passed is and whether it's in anyone's interest to compromise it. For Clockodillo, perhaps a boss might want to double-check that employees are using the timer. Or a competitor might want to check the descriptions of tasks to spy what your company is working on. Or a (more disreputable) competitor might want to disrupt and discredit the service by entering fake data.

If the service deals with health or financial information, it may be an even more attractive target. Location information is also sensitive; burglars might find it convenient to know when you are out of the house.

Security is a really important concern, so you need to bear it in mind while designing the API! But let's start off with an idea of what you want to do with it.

Task	Inputs	Outputs
1. Create a new timed task	User, Timer duration	Timer ID
2. Change duration of timed task	User, Timer ID, New duration	OK
3. Mark timer complete	User, Timer ID	OK
4. Cancel timer	User, Timer ID	OK
5. Describe the timed task	User, Timer ID, Description	OK
6. Get list of timers	User	List of Timer IDs
7. Get information about a timer	User, Timer ID	Description, Create time, Status

Obviously, the request has to pass details to identify the user, which is the problem of identity; that is, the application needs to know for which user to create the timer so that the user can retrieve information about it later.

But the application should also authenticate that request. A password is "good enough" authentication for something that isn't hypersensitive.

So, looking at the preceding list, you can see that tasks 1–4 could be requested by the physical timer device. To pass a description, display details about a list of timers, or get information about them would require more input and output capability than the timer will have!

But for tasks 1–4, how will the timer pass on the username and password? The user could configure them with a computer, via USB. But doing so is potentially complex and means that the device will need some persistent storage; this suggests more work, a more powerful microcontroller, and possibly an extra SD card storage reader for a lower-end controller.

One technique that is commonly used for microcontrollers is that they can send a physical ID, commonly their MAC address (this is a unique ID

assigned to every networked device or rather to their network interface). As this is unique, it can be tied to a user.

Also, you have to consider the risks in sending the identification or authentication data over the Internet—whether that's a MAC address or username and password. If you think back to the description of Internet routing in Chapter 3, you know that the package with the request can be sent by all kinds of routes. If the username and password are in "clear text", they can be read by anyone who is packet sniffing. The two main cases here are as follows:

- **Someone who is targeting a specific user and has access to that person's wired or (unencrypted) wireless network.** This attacker could read the details and use them (to create fake timers or get information about the user).
- **Someone who has access to one of the intermediate nodes.** This person won't be targeting a specific device but may be looking to see what unencrypted data passes by, to see what will be a tempting target.

Of course, your Internet of Things device may not be a tempting target. We hope, though, the word to add to the preceding sentence is "yet!" If your device becomes popular, it may have competitors who would be delighted to expose a security flaw. Although a software product can easily be updated to handle this situation, the logistics with upgrading a hardware project are much more complicated!

Even worse, if a software password is compromised, a website can easily provide a way of changing that password. But while a computer has a monitor and keyboard to make that task easy, an Internet-connected device may not. So you would need a way to configure the device to change its password—for example, a web control panel hosted on the server or on the device itself. This solution is trickier (and does require the machine to have local storage to write the new password to).

One obvious solution to the problem of sending cleartext passwords would be to encrypt the whole request, including the authentication details. For a web API, you can simply do this by targeting `https://` instead of `http://`. It doesn't require any further changes to your application code. It is easy to set up most web servers to serve HTTPS, and we do this in the sample code for this chapter.

Resolving this problem may be harder for the device. Encryption requires solving very large equations and takes CPU and memory. The current Arduino platform doesn't have an HTTPS library, for example. While more powerful microcontrollers, and no doubt future versions of Arduino will,

you can easily imagine even smaller controllers in the future that have similar limitations.

If there is any chance that your connected device could be used maliciously by an attacker, then the ability to secure the communications should be a key factor in deciding on the platform to use. It is unlikely that an attacker could glean anything from the data being gathered by an air quality monitor, for example, but if the data is a reasonable proxy for occupancy of your house or if it can control items and so on, then you need to ensure that it is secure.

If you are defining your own API, there are cryptography libraries for Arduino, so there's scope for using them for a custom form of secure communications. You need to do so carefully and with the help of a security expert, if you take this approach.

The OAuth 1.0 protocol—used by services such as Twitter to allow third-party applications to access your account without requiring your password—is a good example of providing strong authentication without using HTTPS. The content of the API call is still sent in the clear, so an attacker sniffing the network traffic would still be able to see what was happening, but he wouldn't be able to modify or replay the requests. To add encryption—preventing people from watching what is going on—with OAuth 1.0, you would still have to run it over HTTPS. The OAuth 1.0 guide has a useful discussion of the issues it addresses, at `http://hueniverse.com/oauth/guide/security/`.

As a compromise, to save complicating discussion of the API, for this example we suggest insisting on username/password over HTTPS for any requests done over the web but allowing a MAC address over HTTP for requests 1-4 that will be sent by the timer.

Here's a revised table; we also added some requests to add and check the MAC address for a user and categorised the previous requests by the type of resource they affect.

Task	Auth	Inputs	Outputs
1. Create a new timed task	MAC or User/Pass	Timer duration	Timer ID
2. Change duration of timed task	MAC or User/Pass	Timer ID, New timer duration	OK
3. Mark timer complete	MAC or User/Pass	Timer ID	OK
4. Cancel timer	MAC or User/Pass	Timer ID	OK

Task	Auth	Inputs	Outputs
5. Describe the timed task	User/Pass	Timer ID, Description	OK
6. Get list of timers	User/Pass		List of Timer IDs
7. Get information about a timer	User/Pass	Timer ID	Info about Timer
8. Register a device to a user	User/Pass	MAC address	OK
9. Get details of the user's device	User/Pass		MAC address

As you can see, the first four options can be set by the device, but more methods relate to the clients (website or native) than the device itself. This is often the case: the device provides a number of functions that its inputs and outputs are particularly well suited for, but a whole host of other functions to support the data, control authentication, and present and edit it may require a richer input device (perhaps another Thing or a general-purpose device such as a computer or phone).

IMPLEMENTING THE API

An API defines the messages that are sent from client to server and from server to client. Ultimately, you can send data in whatever format you want, but it is almost always better to use an existing standard because convenient libraries will exist for both client and server to produce and understand the required messages.

Here are a few of the most common standards that you should consider:

- **Representational State Transfer (REST):** Access a set of web URLs like `http://timer.roomofthings.com/timers/` or `http::/timer.roomofthings.com/timers/1234` using HTTP methods such as `GET` and `POST`, but also `PUT` and `DELETE`. The result is often XML or JSON but can often depend on the HTTP content-type negotiation mechanisms.
- **JSON-RPC:** Access a single web URL like `http://timer.roomofthings.com/api/`, passing a JSON string such as `{'method':'update', 'params': [{'timer-id':1234, 'description':'Writing API chapter for book'}], 'id':12}`. The return value would also be in JSON, like `{'result':'OK', 'error':null, 'id':12}`.

- **XML-RPC:** This standard is just like JSON-RPC but uses XML instead of JSON.
- **Simple Object Access Protocol (SOAP):** This standard uses XML for transport like XML-RPC but provides additional layers of functionality, which may be useful for very complicated systems.

Jason and the Remote Procedure Calls

A brief word on some of the acronyms we're throwing at you here. None of them are vital to understand, but a brief description will no doubt help.

- JavaScript Object Notation (JSON), pronounced "Jason", is a way of formatting data so that it can be easily exchanged between different systems. As the name suggests, it grew from the JavaScript programming language but these days is just as easy to work with in other languages such as Ruby and Python.

  At its core it is a series of properties, in the form:

  ```
  "property name": "property value"
  ```

  The property values can be a string, a number, a Boolean value (true or false), or another JSON object or array (a sequence of objects).

  Individual properties are separated from each other with commas, and a set of different properties can be grouped into an object with { }. Arrays (a sequence of the same sort of object) are grouped with [ ].

  For example, an array of two objects — each with a name and an age — would look like this:

  ```
  [
      { "name": "Object 1", "age": 34 },
      { "name": "Second object", "age": 45 }
  ]
  ```

  For full details see the JSON website, `http://json.org/`.

- Remote Procedure Call (RPC) is a term to describe ways of calling programming code which isn't on the same computer as the code you are writing. The web APIs we have been discussing so far are a form of RPC. However, because the "web" part of that description better explains how the remote communication is done, the RPC moniker tends not to be used.

- Extensible Markup Language (XML), for the purposes we discuss in this book, can be thought of as an alternative to JSON. It uses < > to demark its elements and tends to be much more verbose than the equivalent JSON would be. As a result, it is less well suited for resource-constrained systems such as Internet of Things devices, so we recommend using JSON if you have the choice. XML has a common parentage with HTML, so if you're familiar with that, XML won't be unfamiliar. XML is defined by the World Wide Web Consortium (W3C), who also look after HTML, CSS, and other web standards. The XML section of their website (`www.w3.org/standards/xml/`) is a good starting point to learn more.

We recommend using REST, but you may have reasons to use another standard. For example, if you are trying to replicate the interface of another XML-RPC service, or you are already familiar with SOAP from other projects, that may well trump the other factors.

For this chapter, we use a REST API because it is popular, well supported, and simple to interact with for a limited microcontroller. The design considerations we describe mostly apply for all the standards, however.

REST has some disadvantages, too. For example, some of the HTTP methods aren't well supported by every client or, indeed, server. In particular, web browsers only natively support `GET` and `POST`, which can complicate things when interacting with REST from a web page.

There is also a lot of disagreement over best practices. REST experts may not always look at the most pragmatic solution favourably. This book isn't a resource on REST as such but aims to provide a flavour of how to use it; we try to point out where we're making a decision out of expediency.

In REST, you attach each resource to a URL and act on that. For example, to interact with a timer, you might speak to /timers/1234 for a timer with ID 1234. To create an entirely new timer, you would talk to /timers. As you can see, you can use different "methods" depending on whether you want to `GET` a resource, `POST` it onto the server in the first place, `PUT` an update to it, or `DELETE` it from the server.

Authorization and Session Management

In the previous table, we suggested passing username and password each time. That isn't really a good idea. If an attacker compromised the transaction, then she would have access to both. It is often a much better idea to perform a single login and then send some kind of token back with subsequent requests. This approach could be limited in terms of time or of session. In the case of REST, we are trying to use HTTP functionality as richly as possible. It turns out that "some kind of token" is a session cookie. Most servers and clients handle cookies automatically, so on subsequent requests only the cookie needs to be checked. Although that sounds like good news, at present, the Arduino HttpClient libraries don't support cookies. This issue will no doubt be resolved soon or worked around (by parsing and setting the HTTP headers manually). But for this timer example, you can continue to pass the MAC address for every request.

So the REST API will finally look like this:

Resource URL	Method	Auth	Parameters	Outputs
1. /timers	POST	MAC or Cookie	Timer duration	Timer ID
2. /timers/:id/duration	PUT	MAC or Cookie	Timer duration	OK
3. /timers/:id/complete	PUT	MAC or Cookie		OK
4. /timers/:id	DELETE	MAC or Cookie		OK
5. /timers/:id/description	PUT	Cookie	Description	OK
6. /timers	GET	Cookie		List of Timer IDs
7. /timers/:id	GET	Cookie		Info about Timer
8. /user/device	PUT	Cookie	MAC address	OK
9. /user/device	GET	Cookie		MAC address
10. /login	POST	User/Pass	User/Pass	Cookie + OK
11. /user	POST		User/Pass	Cookie + OK

All the preceding work is vital to build an idea about how your Internet of Things device and service will interact. Actually, programming it is beyond the scope of this book, although we do present a rough partial implementation as an example. There is no single "best" solution for writing the code, and a lot of the choices will depend on your programming specializations, or if you are hiring a developer to do the back-end work, whether you can get someone who is good.

Following are some of the parameters you should consider when deciding on a platform for your web back end:

- What do you already know (if you are planning to develop the code yourself)?
- What is the local/Internet recruiting market like (if you are planning to outsource)?

- Is the language thriving? Is it actively developed? Does it have a healthy community (or commercial support)? Is there a rich ecosystem of libraries available?

We are deliberately not mentioning variables such as "power" or "speed". Almost any language that fulfils the most important criteria here is powerful enough to get the job done. When writing a web app, you probably care more about speed of development, robustness, and maintainability than power or speed. If you scale up enough that you have to rewrite your infrastructure in Erlang or critical subsystems in C++, that is a good problem to have!

If you are dipping your toes into web programming and don't have a firm idea about what platform to use already, you might want to consider a dynamic language, such as Ruby, Perl, Python, JavaScript (Node.js), or PHP. They are relatively simple to learn, well supported by web hosts, and have a host of libraries to help get the code written.

Those of you with experience of the Microsoft developer ecosystem may want to use C# or ASP.net. If you have skills with the JVM, Java or Scala would be a fine choice. If you're a functional programmer, Clojure, Erlang, or Haskell will get the job done.

Next, we look at an example of the back-end code in Perl, using the Dancer framework. This "lightweight" web framework uses a mindset similar to that of Ruby's Sinatra. We cover only the most interesting parts here, but you can look at the full example at `https://github.com/osfameron/aBookOfThings-examples/`, along with other code discussed in the book. (All the code on this site is open source, so feel free to fork it and make contributions, perhaps translations of the code, into your favourite programming language.)

Back-end Code in Perl

Perl has advantages and disadvantages just like all the other languages we mention. It is actively developed, has a great ecosystem of libraries on CPAN, and is powerful enough to serve large sites if the system is architected well. Its disadvantages (some areas have thorny syntax, the job market is patchy in some locations) are overweighed, in our case, by the fact that it's what Hakim knows best.

After a small amount of boilerplate, the code mostly consists of handlers for the different API calls. Each handler declares the HTTP verb (`GET`, `POST`, `PUT`, `DELETE`) and the route that it handles. Parameters can be passed within the route, marked by a colon (`:id`, for example), or as part of the HTTP request.

```
#1 Create a new timed task
post "/timers.:format" => sub {
    my $user = require_user;
        # 'require_user' wants a session cookie
        # OR a valid MAC address.

    my $duration = param 'duration'
        or return status_bad_request('No duration passed');

    my $timer = schema->resultset('Timer')->create({
        user_id  => $user->id,
        duration => $duration,
        status   => 'O', # open
    });
    return status_created({
        status=>'ok',
        id => $timer->id,
    });
};

#2 Change duration of timed task
put "/timers/:id/duration.:format" => {
    my $user = require_user;
    my $duration = param 'duration'
        or return update_complete;
    my $timer = require_open_timer;
        # a timer is open if it's in 'O' status

    ## NB: the following calculation has to extend the time
    ## as of now
    my $start_datetime = $timer->start_datetime;
    my $new_end_time = DateTime->now->add(
        minutes => $duration );
    my $total_duration = ($new_end_time - $start_datetime)
       ->in_units('minutes');

    $timer->update({ duration => $total_duration });

    return status_ok({
        ok => 1,
        message => 'Timer length updated',
```

```
    });
};

#3 Mark timer complete
put "/timers/:id/complete.:format" => sub {
    my $user = require_user;
    my $timer = require_open_timer;

    $timer->update({ status => 'C' });

    return status_ok({
        ok => 1,
        message => 'Timer marked complete',
    });
};

#4 Cancel timer
del "/timers/:id.:format" => sub {
    my $user = require_user;
    my $timer = require_timer;

    $timer->update({ status => 'D' });

    return status_ok({
        status => 'ok',
    });
};

#5 Describe the timed task
put "/timers/:id/description.:format" => sub {
    my $user = require_session;
        # 'require_user' demands a session cookie!
    my $timer = require_open_timer;
    my $description = param 'description';

    $timer->update({ description => $description });

    return status_ok({
        ok => 1,
        message => 'Description updated',
    });
};

#6 Get list of timers
get "/timers.:format" => sub {
    my $user = require_session;
```

```
    return status_ok({
        status => 'ok',
        timers => [ map $_->serialize, $user->timers ],
    });
};

#7 Get information about a timer
get "/timers/:id.:format" => sub {
    my $user = require_session;
    my $timer = require_timer;

    return status_ok({
        status => 'ok',
        timer => $timer->serialize,
    });
};

#8 TODO Set the user's device MAC address

#9 TODO Get the user's device MAC address

#10 Login
post "/login.:format" => sub {
    my $username = param 'user';
    my $password = param 'pass';

    my $user = schema->resultset('User')->find({
       email => $username });

    if ($user && $user->check_password($password)) {
        session user_id => $user->id;
        return status_ok({
            status=>'ok',
            message=>'Login OK',
        });
    }
    else {
        return status_bad_request("Bad username or password");
    }
};

#11 Register the user
post "/user.:format" => sub {
    my $username = param 'user';
    my $password = param 'pass';

    if (schema->resultset('User')->find({ email => $username }))
    {
```

```
        return status_bad_request("Duplicate user");
    }
    else {
        my $user = schema->resultset('User')->create({
            email    => $username,
            password => $password,
        });
        return status_created({
            status=>'ok',
            id => $user->id,
        });
    }
};
```

Note how all the requests end with `.:format`. This means that you could post to `http://api.roomofthings.com/timers.json` to get the result back from the server in JSON format or to `http://api.roomofthings.com/timers.txt` to get back a simple string, optimised for easy parsing by a microcontroller.

This code calls some functions defined by Dancer and Dancer::Plugin::REST, such as `status_ok` and `param`. The other things we had to write, omitted from the preceding listing, are as follows:

- The database definition (simply creating two tables, `users` and `timers`) and the code to connect them to Perl using `DBIx::Class` (an ORM layer, similar to ActiveRecord or LINQ)
- Some utility functions for user management: `require_user` and `require_session` (which distinguish between the "MAC or Cookie" and "Cookie" cases)
- Similar utility functions `require_timer` and `require_open_timer` to get the timer object from the database
- Basic configuration of Dancer/PSGI, to make the application easy to test and deploy

USING CURL TO TEST

While you're developing the API, and afterwards, to test it and show it off, you need to have a way to interact with it. You could create the client to interface with it at the same time (either an application on the web or computer, or the code to make your Internet of Things project connect to it). In this case, while we were developing Clockodillo, the API was ready long before the physical device. Luckily, many tools can interact with APIs, and one very useful one is curl, a command-line tool for transferring all kinds of data, including HTTP.

You can easily issue `GET` requests by simply calling `curl http://timer.roomofthings.com/timers.json`, for example. But, of course, the API is protected with logins. Luckily, curl takes this in its stride! Here is an example of interacting with it on a development server:

```
# the -F flag makes curl POST the request
$ curl http://localhost:3000/user.json \
       -F user=hakim -F pass=secret
{
   "status" : "ok",
   "id" : 2
}
```

curl simply makes an HTTP request and prints out the result to a terminal. Because the command line requests JSON, the result comes back in that format, with a dictionary of `status` and `id` values.

Here are some more examples:

```
# Check that login rejects a bad password
$ curl http://localhost:3000/login.json \
       -F user=hakim -F password=wrong
{
   "error" : "Bad username or password"
}

# save login session to our "cookie jar"
$ curl http://localhost:3000/login.json -c cookie.jar \
       -F user=hakim -F pass=secret
{
   "status" : "ok",
   "message" : "Login OK"
}

# use that cookie to login, and create a 25 minute timer
$ curl http://localhost:3000/timers.json -b cookie.jar \
       -F duration=25
{
    "status" : "ok",
    "id" : 1
}

# change the request to a PUT
$ curl http://localhost:3000/timer/1/duration.json \
       -X PUT -b cookie.jar -F duration=12
{
   "ok" : 1,
   "message" : "Timer length updated"
```

```
}

# GET the information about that timer

$ curl http://localhost:3000/timers/1.json -b cookie.jar
{
   "status" : "ok",
   "timer" : {
      "start_datetime" : "2012-05-21 19:30:40",
      "status" : "O",
      "id" : 1,
      "user_id" : 1,
      "duration" : 12,
      "description" : null,
      "end_datetime" : null
   }
}

# DELETE (cancel) it
$ curl http://localhost:3000/timer/1.json \
       -X DELETE -b cookie.jar
{
   "status" : "ok"
}

# GET it again (note how the status is now 'D' as it's deleted)
$ curl -b cookie.jar http://localhost:3000/timers/1.json
{
   "status" : "ok",
   "timer" : {
      "start_datetime" : "2012-05-21 19:30:40",
      "status" : "D",
      "id" : 1,
      "user_id" : 1,
      "duration" : 12,
      "description" : null,
      "end_datetime" : null
   }
}
```

Although the preceding examples may look a little arcane if you aren't familiar with code, we hope they are understandable enough for you to get a feel for what is happening. They exercise the main methods in the API and show the sorts of basic sanity tests that you would perform to make sure your code was functioning in the way you anticipated. That lets you go on to develop the device code knowing that the service it will be talking to has a reasonable foundation.

GOING FURTHER

The preceding sketch is missing a few tweaks before it can become a production-ready API. The timer duration changing is rudimentary. The code doesn't handle the case in which the timer should already have expired by the time the user tries to change it. Perhaps the data structure should also be expanded to store more history about a single timer (for example, if the user changes the time repeatedly, the server could store each change rather than only the total duration).

This example also has a number of architectural features that we didn't examine at all.

API Rate Limiting

If the service becomes popular, managing the number of connections to the site becomes critical. Setting a maximum number of calls per day or per hour or per minute might be useful. You could do this by setting a counter for each period that you want to limit. Then the authentication process could simply increment these counters and fail if the count is above a defined threshold. The counters could be reset to 0 in a scheduled cron job.

While a software application can easily warn users that their usage limit has been exceeded and they should try later, if a physical device suddenly fails, users might assume that it is broken! Solutions to this problem might include simply not applying the limit to calls made by a device.

OAuth for Authenticating with Other Services

While OAuth may not (currently) be the best solution for connecting with a microcontroller (at present, there are no accepted libraries for Arduino), there is no reason why the back-end service should not accept OAuth to allow hooks to services like Twitter, music discovery site last.fm, or the web automation of If This Then That.

Interaction via HTML

The API currently serialises the output only in JSON, XML, and Text formats. You might also want to connect from a web browser. When we first looked at the API design, we split up tasks into those that the device could do and then the rest. The latter could easily be done in a browser-based application. Of course, the users won't want to make raw API calls, and the flows taken to carry out an action may well be slightly different, but the basic data being

manipulated is the same: The calls we've looked at would form the heart of the web application, just as they do the experience with the physical device.

Note that not every Internet of Things product needs a browser application. Perhaps the API is all you need, with maybe a static home page containing some documentation about how to call the API.

In the case of Clockodillo, however, we do want to have a set of web pages to interact with: Users should be able to look at their timers, assign descriptions, and so on.

It would be easy to set up specific route handlers just for the HTML application, for example:

```
post '/login.html' => sub { ... }
```

But a more elegant approach might be to use exactly the same code as before but with an HTML option. Instead of returning a JSON string, this code might inject the data into an HTML template. Simply calling, for example, /timers/1234.html instead of /timers/1234.json would then get a view of that data targeted at a human rather than a connected device.

Drawbacks

Although web browsers do speak HTTP, they don't commonly communicate in all the methods that we've discussed. In particular, they tend to support the following:

- `GET`: Used to open pages and click on links to other pages. You can link to `http://timer.roomofthings.com/timers/1234.html` and get the HTML version of the API call (using the "get_timer" template).
- `POST`: Used when submitting a form or to upload files. To post a timer, you could create a web form like the following:

  ```
  <form method="POST" action="/timers.html">
      <input type="text" name="duration">
      <input type="submit" value="Create a new timer!">
  </form>
  ```

  This form calls the `POST` action and returns the appropriate HTML.

But what about the lovingly crafted `PUT` and `DELETE` methods? Web browsers don't commonly support those…but never fear! One option is to make these calls in JavaScript, which can indeed support them. Another is to

"tunnel" the requests through a `POST`. There is a convention in Perl to use a field called `x-tunneled-method`, which you could implement like this:

```
<form method="POST"
    action="/timer.html?x-tunneled-method=DELETE">
    <input type="hidden" name="id" value="1234">
    <input type="submit" value="Cancel this timer!">
</form>
```

Now you just need to convince your web framework to accept this `POST` as if it were actually a `DELETE`. In the example app using Dancer, we use the module `Plack::Middleware::MethodOverride` to do this in a single line (`https://metacpan.org/module/Plack::Middleware::MethodOverride`). Other frameworks will have similar extensions.

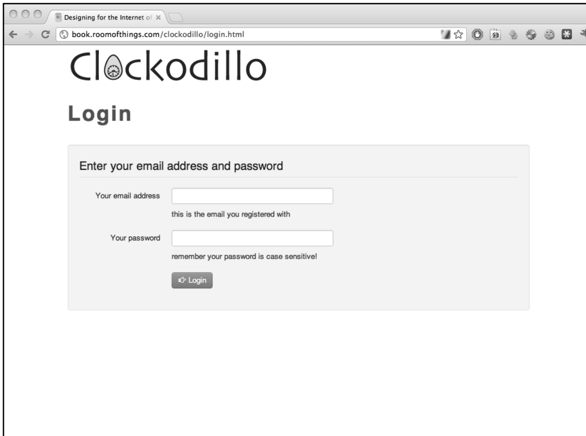
Alternatively, you could write the web application in an entirely different code base and interact with the main service through the API. This can be a winning combination because it forces the human-facing code to use (and therefore exercise) the same API that the device uses, increasing the amount of testing it receives and preventing the device-facing code from being neglected. Whether you decide to follow that path would depend very much on your team's skill set.

Designing a Web Application for Humans

However you choose to implement it, as well as the text-based API we've spent most of the chapter working on, you can easily also have an elegant and well-designed application for humans to interact with.

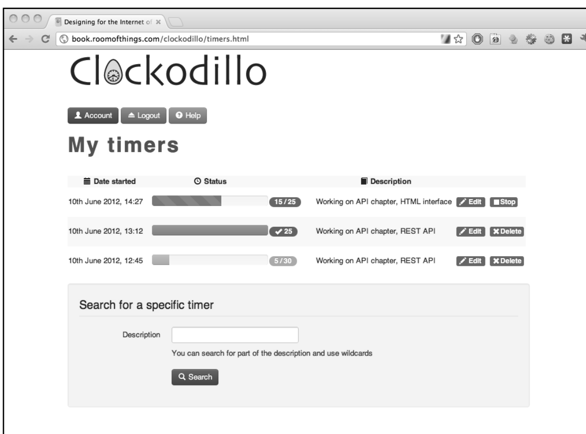
Because numerous excellent books are available on designing a web application, we don't look at this topic in any great detail, but you might be interested in looking at some examples to think about the design process.

For example, the following figure shows a static login page, to be served by `GET`. The API didn't even specify a `GET` action, as it was superfluous for a computer. This page is entirely for the convenience of a human. All the labels like "Your email address" and the help text like "Remember your password is case sensitive" are purely there to guide the user. The logo, as well as proving that we are really not designers, is there as a branding and visual look and feel for the site.



The human-facing Clockodillo login page.

That's a simple example, but the following figure shows an even more extreme change. The list of timers, instead of being a JSON string containing a raw data structure, is highly formatted. The dates are formatted for humans. The duration of the timer and the status (in progress, completed, abandoned) are visualised with colours, progress bars, and a duration "badge". The page also links to other actions: The "Edit" button opens a page that allows access to the actions that change description, and so on. The menu bar at the top links to other functions to help users flow through the tasks they want to carry out.



The human-facing list of timers.

Finally, as we were preparing this mockup, we added a "Search for a specific timer" input. This hadn't even occurred to us when preparing the API (the timer device doesn't need it) but seemed obvious as soon as we thought about things from the viewpoint of a human user. Looking at your product from both contrasting perspectives (machine and human) will make it stronger and better designed.

REAL-TIME REACTIONS

We've looked at a traditional sort of API, where you make an HTTP request to the server and receive a response. This method has some disadvantages if you want a very responsive system. To establish an HTTP request requires several round-trips to the server. There is the TCP "three-step handshake" consisting of a `SYN` (synchronise) request from the client, a `SYN-ACK` from the server to "acknowledge" the request, and finally an `ACK` from the client. Although this process can be near instantaneous, it could also take a noticeable amount of time.

The time taken to establish the connection may or may not matter. Any of the most powerful boards is able to run the connection in the background and respond to it when it's completed. For a bare-bones board such as the Arduino, the current Ethernet/HTTP shields and libraries tend to block during the connection, which means that during that time, the microcontroller can't easily do any other processing (although it is possible to work around this issue using hardware interrupts, doing so poses certain restrictions and complications). Because the connection is usually made on a "breakout board" with its own processor, there is no reason that the connection couldn't happen in parallel, without blocking the main thread, so this restriction may well be lifted in future.

If you want to perform an action the instant that something happens on your board, you may have to factor in the connection time. If the server has to perform an action immediately, that "immediately" could be nearly a minute later, depending on the connection time. For example, with the task timer example, you might want to register the exact start time from when the user released the dial, but you would actually register that time plus the time of connection.

We look at two options here: polling and the so-called "Comet" technologies. And then, in the section on non-HTTP protocols, MQTT, XMPP, and CoAP offer alternative solutions.

POLLING

If you want the device or another client to respond immediately, how do you do that? You don't know when the event you want to respond to will happen, so you can't make the request to coincide with the data becoming available. Consider these two cases:

- The WhereDial should start to turn to "Work" the moment that the user has checked into his office.
- The moment that the task timer starts, the client on the user's computer should respond, offering the opportunity to type a description of the task.

The traditional way of handling this situation using HTTP API requests was to make requests at regular intervals. This is called *polling*. You might make a call every minute to check whether new data is available for you. However, this means that you can't start to respond until the poll returns. So this might mean a delay of (in this example) one minute plus the time to establish the HTTP connection. You could make this quicker, polling every 10 seconds, for example. But this would put load on the following:

- **The server:** If the device takes off, and there are thousands of devices, each of them polling regularly, you will have to scale up to that load.
- **The client:** This is especially important if, as per the earlier Arduino example, the microcontroller blocks during each connect!

COMET

Comet is an umbrella name for a set of technologies developed to get around the inefficiencies of polling. As with many technologies, many of them were developed before the "brand" of Comet was invented; however, having a name to express the ideas is useful to help discuss and exchange ideas and push the technology forward.

Long Polling (Unidirectional)

The first important development was "long polling", which starts off with the client making a polling request as usual. However, unlike a normal poll request, in which the server immediately responds with an answer, even if that answer is "nothing to report", the long poll waits until there is something to say. This means that the server must regularly send a keep-alive to the client to prevent the Internet of Things device or web page from concluding that the server has simply timed out.

Long polling would be ideal for the case of WhereDial: the dial requests to know when the next change of a user's location will be. As soon as WhereDial receives the request, it moves the dial and issues a new long poll request. Of course, if the connection drops (for example, if the server stops sending keep-alive messages), the client can also make a new request.

However, it isn't ideal for the task timer, with which you may want to send messages from the timer quickly, as well as receive them from the server. Although you can send a message, you have to establish a connection to do so. Hence, you can think of long polling as unidirectional.

Multipart XMLHttpRequest (MXHR) (Unidirectional)

When building web applications, it is common to use a JavaScript API called `XMLHttpRequest` to communicate with the web server without requiring a full new page load. From the web server's point of view, these requests are no different from any other HTTP request, but because the intended recipient is some client-side code, conventions and support libraries (both client- and server-side) have developed to address this method of interaction specifically.

Many browsers support a `multipart/x-mixed-replace` content type, which allows the server to send subsequent versions of a document via XHR. Note that XMLHttpRequest is a misnomer because there's no requirement to actually use XML at all. Using this content type is perhaps more sophisticated if you want to be able to receive multiple messages from the server.

It is perfectly possible to simply long poll and create a new request on breaking the old one, but this means that you might miss a message while you're establishing the connection. In the example of WhereDial, this is unlikely; you're unlikely to change location first to Home and then to Work in quick succession. However, for an Internet of Things device such as Adrian's Xively meter, which tries to show the state of a Xively feed in real time, being able to respond to changes from the server almost immediately is the essential purpose of the device.

HTML5 WebSockets (Bidirectional)

In Chapter 3 you saw how the HTTP protocol used in web services sits atop the TCP protocol. Traditionally, the API used to talk directly to the TCP layer is known as the *sockets API*. When the web community was looking to provide similar capabilities at the HTTP layer, they called the solution *WebSockets*.

Although WebSockets are currently a working draft in the HTML5 spec, they seem to have traction in modern browsers, servers, and other clients.

For example, there is a (partial) implementation for the Arduino platform (`https://github.com/krohling/ArduinoWebsocketClient`).

WebSockets have the benefit of being bidirectional. You can consider them like a full Unix socket handle that the client can write requests to and read responses from.

This might well be the ideal technology for the task timer. After a socket is established, the timer can simply send information down it about tasks being started, modified, or cancelled, and can read information about changes made in software, too.

> *Because WebSockets are new and push the HTTP protocol in a slightly unorthodox direction, they are known to have some issues with proxy servers. This situation should change as the proxies currently broken in this respect are fixed to be aware of WebSockets. This may be an issue with your system's architecture; see the later section on "Scaling".*

Implementations

The options described in the preceding section seemed to us to have most traction currently; however, as a fast-changing area with no absolute consensus as yet, the actual details of transports and limitations are bound to change. It is worth paying attention to these transports as they develop. The Wikipedia page on Comet (`http://en.wikipedia.org/wiki/Comet_(programming)`) is a useful starting point for tracking the current state of play.

Let's look at support for these techniques on the three main platforms that you may need to consider for an Internet of Things application: the browser web app (if applicable), the microcontroller itself, and the server application.

On the browser side, it is often possible to abstract the actual transport using a library which chooses which method to connect to the server. For example, it might use WebSockets if available; otherwise, it will fall back to MXHR or long polling. This capability is useful because each web browser currently has varying levels of support for the different techniques. There are well-known Comet libraries for jQuery and for Dojo.

In addition, many web servers have abstractions to support Comet techniques. `Web::Hippie::Pipe` provides a unified bidirectional abstraction for Perl web servers such as Twiggy, again using WebSockets if available,

otherwise MXHR or long polling. You can find similar abstractions for `node.js` (JavaScript), `Thin` (Rails), `jetty` (Java), and so on.

There are also libraries for the microcontroller; however, they tend to support only one scheme. For example, several dedicated WebSockets libraries are available for Arduino. In fact, the fallback to different methods of interchanging data aren't really needed on the Arduino. Unlike the case of a desktop web app, with Arduino you don't have to worry about the users having different browsers because you'll be providing the firmware for the device.

Scaling

An important consideration is that all these Comet techniques require the client to have a long-term connection with the server. For a single client, this is trivial. But if there are many clients, the server has to maintain a connection with each of them. If you run a server with multiple threads or processes, you effectively have an instance of the server for each client. As each thread or process will consume system resources, such as memory, this doesn't scale to many clients.

Instead, you might want to use an asynchronous web server, which looks at each client connection in turn and services it when there is new input or output. If the server can service each client quickly, this approach can scale up to tens of thousands of clients easily. There is a problem that each process on a typical Unix server has a maximum number of sockets, so you are restricted to that number of simultaneous clients. This, of course, is a good problem to have! When you hit that wall, you can look at load-balancing and other techniques that a good systems team will be able to apply to scale up the load.

You also might be able to let your front-end proxy (Varnish or similar) do some of the juggling of persistent client connections.

OTHER PROTOCOLS

As you have seen, although HTTP is an extremely popular protocol on the Internet, it isn't ideally suited to all situations. Rather than work around its limitations with one of the preceding solutions, another option—if you have control of both ends of the connection—is to use a different protocol completely.

There are plenty of protocols to choose from, but we will give a brief rundown of some of the options better suited to Internet of Things applications.

MQ TELEMETRY TRANSPORT

MQTT (`http://mqtt.org`) is a lightweight messaging protocol, designed specifically for scenarios where network bandwidth is limited or a small code footprint is desired. It was developed initially by IBM but has since been published as an open standard, and a number of implementations, both open and closed source, are available, together with libraries for many different languages.

Rather than the client/server model of HTTP, MQTT uses a publish/subscribe mechanism for exchanging messages via a *message broker*. Rather than send messages to a pre-defined set of recipients, senders publish messages to a specific *topic* on the message broker. Recipients subscribe to whichever topics interest them, and whenever a new message is published on that topic, the message broker delivers it to all interested recipients. This makes it much easier to do one-to-many messaging, and also breaks the tight coupling between the client and server that exists in HTTP.

A sister protocol, MQTT for Sensors (MQTT-S), is also available for extremely constrained platforms or networks where TCP isn't available, allowing MQTT's reach to extend to sensor networks such as ZigBee.

EXTENSIBLE MESSAGING AND PRESENCE PROTOCOL

Another messaging solution is the Extensible Messaging and Presence Protocol, or XMPP (`http://xmpp.org`). XMPP grew from the Jabber instant messaging system and so has broad support as a general protocol on the Internet. This is both a blessing and a curse: it is well understood and widely deployed, but because it wasn't designed explicitly for use in embedded applications, it uses XML to format the messages. This choice of XML makes the messaging relatively verbose, which could preclude it as an option for RAM-constrained microcontrollers.

CONSTRAINED APPLICATION PROTOCOL

The Constrained Application Protocol (CoAP) is designed to solve the same classes of problems as HTTP but, like MQTT-S, for networks without TCP. There are proposals for running CoAP over UDP, SMS mobile phone messaging, and integration with 6LoWPAN. CoAP draws many of its design features from HTTP and has a defined mechanism to proxies to allow mapping from one protocol to the other. At the time of this writing, the protocol is going through final stages of becoming a defined standard, with

the work being coordinated by the Internet Engineering Task Force Constrained RESTful Environments Working Group.

SUMMARY

This chapter took a good look at the network side of the Internet of Things. We looked at how to interact with existing services, either through published APIs or via web scraping, and then worked through an example to see how to create something completely new if the need arose.

Together with the previous two chapters, you will now have a good feel for the breadth of work required to build an entire Internet of Things prototype. There is more work to do to take it into production, but you will see that in later chapters. If you are just planning to build something to make your own life easier or more fun, you should be well placed to get cracking.

The next chapter takes us back to the device side of the equation, with a more detailed exploration of the techniques you will need to write code for an embedded system. It explains some of the ways that embedded coding differs from standard desktop or server programming, with tips on how to approach it and how to investigate when things don't quite go to plan.