## Contents

# 1 Accessing and Developing WoT

## 1.1 Chapter 6

### 1.1.1 REST STUFF

- The first layer is called access. This layer is aptly named Access because it covers the most fundamental piece of the WoT puzzle: how to connect a Thing to the web so that it can be accessed using standard web tools and libraries.

- REST provides a set of architectural constraints that, when applied as a whole, empha- sizes scalability of component interactions, generality of interfaces, independent deploy- ment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

- In short, if the architecture of any distributed system follows the REST constraints, that system is said to be RESTful.
- Maximises interoperability and scalability
- Five constraints: Client/server, Uniform interfaces, Stateless, Cacheable, Layered system

1. Client/server
   - Maximises decoupling, as client doesn't need to know how the server works and vice versa
   - Such a separation of concerns between data, control logic, and presentation improves scalability and portability because loose coupling means each component can exist and evolve independently.
2. Uniform interfaces
   - Loose coupling between components can be achieved only when using a uniform interface that all components in the system respect.
   - This is also essential for the Web of Things because new, unknown devices can be added to and removed from the system at any time, and interacting with them will require min- imal effort.
3. Stateless
   - The client context and state should be kept only on the client, not on the server.
   - Each request to server should contain client state, visibility (monitoring and debugging of the server), robustness (recovering from network or application failures) and scalability are improved.
4. Cacheable
   - Caching is a key element in the performance (loading time) of the web today and therefore its usability.
   - Servers can define policies as when data expires and when updates must be reloaded from the server.
5. Layered

- For example, in order to scale, you may make use of a proxy behaving like a load balancer. The sole purpose of the proxy would then be to forward incoming requests to the appropriate server instance.
- Another layer may behave like a gateway, and translate HTTP requests to other protocols.
- Similarly, there may be another layer in the architecture responsible for caching responses in order to minimize the work needed to be done by the server.

6. HATEOAS

   - Servers shouldn't keep track of each client's state because stateless applications are easier to scale. Instead, application state should be addressable via its own URL, and each resource should contain links and information about what operations are possible in each state and how to navigate across states. HATEOAS is particularly useful at the Find layer

7. Principles of the uniform interface of the web

   - Our point here is that what REST and HTTP have done for the web, they can also do for the Web of Things. As long as a Thing follows the same rules as the rest of the web—that is, shares this uniform interface—that Thing is truly part of the web. In the end, the goal of the Web of Things is this: make it possible for any physical object to be accessed via the same uniform interface as the rest of the web. This is exactly what the Access layer enables
   - Addressable resources—A resource is any concept or piece of data in an application that needs to be referenced or used. Every resource must have a unique identi- fier and should be addressable using a unique referencing mechanism. On the web, this is done by assigning every resource a unique URL.
   - Manipulation of resources through representations—Clients interact with services using multiple representations of their resources. Those representations include HTML, which is used for browsing and viewing content on the web, and JSON, which is better for machine-readable content.
   - Self-descriptive messages—Clients must use only the methods provided by the pro- tocol—GET, POST, PUT, DELETE, and HEAD

among others—and stick to their meaning as closely as possible. Responses to those operations must use only well-known response codes—HTTP status codes, such as 200, 302, 404, and 500.

- Hypermedia as the engine of the application state (HATEOAS)—Servers shouldn't keep track of each client's state because stateless applications are easier to scale. Instead, application state should be addressable via its own URL, and each resource should contain links and information about what operations are possi- ble in each state and how to navigate across states.

(a) Principle #1, adressable resources

- REST is a resource-oriented architecture (ROA)
- A resource is explicitly identified and can be individually addressed, by its URI
- A URI is a sequence of characters that unambiguously identifies an abstract or physi- cal resource. There are many possible types of URIs, but the ones we care about here are those used by HTTP to both identify and locate on a network a resource on the web, which is called the URL (Uniform Resource Locator) for that resource.
- An important and powerful consequence of this is the addressability and portability of resource identifiers: they become unique (internet- or intranet-wide)
- Hierachical naming!

(b) Principle #2, manipulation of resources through representation

- On the web, Multipurpose Internet Mail Extensions (MIME) types have been introduced as standards to describe various data for- mats transmitted over the internet, such as images, video, or audio. The MIME type for an image encoded as PNG is expressed with image/png and an MP3 audio file with audio/mp3. The Internet Assigned Numbers Authority (IANA) maintains the list of the all the official MIME media types.
- The tangible instance of a resource is called a representation, which is a standard encoding of a resource using a MIME type.
- HTTP defines a simple mechanism called content negotiation that allows clients to request a preferred data format they

want to receive from a specific service. Using the Accept header, clients can specify the format of the representation they want to receive as a response. Likewise, servers specify the format of the data they return using the Content-Type header.

- The Accept: header of an HTTP request can also contain not just one but a weighted list of media types the client understands
- MessagePack can be used to pack JSON into a binary format, to make it lighter.
- A common way of dealing with unofficial MIME types is to use the x- extension, so if you want your client to ask for MessagePack, use Content-Type: application/x-msgpack.

(c) Principle #3: self-descriptive messages

- REST emphasizes a uniform interface between components to reduce coupling between operations and their implementation. This requires every resource to support a standard, common set of operations with clearly defined semantics and behavior.
- The most commonly used among them are GET, POST, PUT, DELETE, and HEAD. Although it seems that you could do everything with just GET and POST, it's important to correctly use all four verbs to avoid bad surprises in your applications or introducing security risks.
- CRUD operations; create, read, update and delete
- HEAD is a GET, but only returns the headers
- POST should be used only to create a new instance of something that doesn't have its own URL yet
- PUT is usually modeled as an idempotent but unsafe update method. You should use PUT to update something that already exists and has its own URL, but not to create a new resource
- Unlike POST, it's idempotent because sending the same PUT message once or 10 times will have the same effect, whereas a POST would create 10 different resources.
- A bunch of error codes as well: 200, 201, 202, 401, 404, 500, 501

- CORS—ENABLING CLIENT-SIDE JAVASCRIPT TO ACCESS RESOURCES

(d) CORS

- Although accessing web resources from different origins located on various servers in any server-side application doesn't pose any problem, JavaScript applications running in web browsers can't easily access resources across origins for security reasons. What we mean by this is that a bit of client-side JavaScript code loaded from the domain apples.com won't be allowed by the browser to retrieve particular representations of resources from the domain oranges.com using particular verbs.
- This security mechanism is known as the same- origin policy and is there to ensure that a site can't load any scripts from another domain. In particular, it ensures that a site can't misuse cookies to use your credentials to log onto another site.
- Fortunately for us, a new standard mechanism called cross-origin resource sharing (CORS)9 has been developed and is well supported by most modern browsers and web servers.

When a script in the browser wants to make a cross-site request, it needs to include an Origin header containing the origin domain. The server replies with an Access- Control-Allow-Origin header that contains the list of allowed origin domains (or * to allow all origin domains)

- When the browser receives the reply, it will check to see if the Access-Control- Allow-Origin corresponds to the origin, and if it does, it will allow the cross-site request.

For verbs other than GET/HEAD, or when using POST with representations other than application/x-www-form-urlencoded, multipart/form-data, or text/ plain, an additional request called preflight is needed. A preflight request is an HTTP request with the verb OPTIONS that's used by a browser to ask the target server whether it's safe to send the cross-origin request.

(e) Principle #4 : Hypermedia as the Engine of Application State

- contains two subconcepts: hypermedia and application state.
- This fourth principle is centered on the notion of hypermedia, the idea of using links as connections between related ideas.

- Links have become highly popular thanks to web browsers yet are by no means limited to human use. For example, UUIDs used to identify RFID tags are also links.
- Based on this representation of the device, you can easily follow these links to retrieve additional information about the subresources of the device
- The application state—the AS in HATEOAS—refers to a step in a process or workflow, similar to a state machine, and REST requires the engine of application state to be hypermedia driven.
- Each possible state of your device or application needs to be a RESTful resource with its own unique URL, where any client can retrieve a representation of the current state and also the possible transitions to other states. Resource state, such as the status of an LED, is kept on the server and each request is answered with a representation of the current state and with the necessary information on how to change the resource state, such as turn off the LED or open the garage door.
- In other words, applications can be stateful as long as client state is not kept on the server and state changes within an application happen by following links, which meets the self-contained-messages constraint.
- The OPTIONS verb can be used to retrieve the list of operations permitted by a resource, as well as metadata about invocations on this resource.

(f) Five-step process

- A RESTful architecture makes it possible to use HTTP as a universal protocol for web-connected devices. We described the process of web-enabling Things, which are summarized in the five main steps of the web Things design process:
- Integration strategy—Choose a pattern to integrate Things to the internet and the web, either directly or through a proxy or gateway. This will be covered in chapter 7, so we'll skip this step for now.
- Resource design—Identify the functionality or services of a Thing and organize the hierarchy of these services. This is where we apply design rule #1: address- able resources.

        - Representation design—Decide which representations will be served for each resource. The right representation will be selected by the clients, thanks to design rule #2: content negotiation.
        - Interface design—Decide which commands are possible for each service, along with which error codes. Here we apply design rule #3: self-descriptive messages.
        - Resource linking design—Decide how the different resources are linked to each other and especially how to expose those resources and links, along with the operations and parameters they can use. In this final step we use design rule #4: Hypermedia as the Engine of Application State.

8. Design rules
    (a) #2–CONTENT NEGOTIATION
        - Web Things must support JSON as their default representation.
        - Web Things support UTF8 encoding for requests and responses
        - Web Things may offer an HTML interface/representation (UI).

    (b) #3 : Self-descriptive messages
        - Web Things must support the GET, POST, PUT, and DELETE HTTP verbs.
        - Web Things must implement HTTP status codes 20x, 40x, 50x.
        - Web Things must support a GET on their root URL.
        - Web Things should support CORS

    (c) #4 : HATEOAS
        - Web Things should support browsability with links.
        - Web Things may support OPTIONS for each of its resources.

#### 1.1.2 EVENT STUFF

1. Events and stuff

    - Unfortunately, the request-response model is insufficient for a number of IoT use cases. More precisely, it doesn't match event-driven use cases where events must be communicated (pushed) to the clients as they happen.

- A client-initiated model isn't practical for applications where notifications need to be sent asynchronously by a device to clients as soon as they're produced.
- polling is one way of circumventing the problem, however it's inefficient, as the client will need to make many requests which will simply return the same response. Additionally, we might not "poll" at the exact time an event takes place.
- Most of the requests will end up with empty responses (304 Not Modified) or with the same response as long as the value observed remains unchanged.

2. Publish/subscribe

    - What's really needed on top of the request-response pattern is a model called publish/subscribe (pub/sub) that allows further decoupling between data consumers (subscribers) and producers (publishers). Publishers send messages to a central server, called a broker, that handles the routing and distribution of the messages to the various subscribers, depending on the type or content of messages.
    - A publisher can send notifications into a topic, which subscribers can have subscribed to

3. Webhooks

    - The simplest way to implement a publish-subscribe system over HTTP without break- ing the REST model is to treat every entity as both a client and a server. This way, both web Things and web applications can act as HTTP clients by initiating requests to other servers, and they can host a server that can respond to other requests at the same time. This pattern is called webhooks or HTTP callbacks and has become popular on the web for enabling different servers to talk to each other.
    - The implementation of this model is fairly simple. All we need is to implement a REST API on both the Thing and on the client, which then becomes a server as well. This means that when the Thing has an update, it POSTs it via HTTP to the client
    - Webhooks are a conceptually simple way to implement bidirectional communication between clients and servers by turning everything into a server.

- webhooks have one big drawback: because they need the subscriber to have an HTTP server to push the notification, this works only when the subscriber has a publicly accessible URL or IP address.

4. Comet

   - Comet is an umbrella term that refers to a range of techniques for circumventing the limitations of HTTP polling and webhooks by introducing event-based communication over HTTP.
   - This model enables web servers to push data back to the browser without the client requesting it explicitly. Since browsers were initially not designed with server-sent events in mind, web application developers have exploited several specification loop- holes to implement Comet-like behavior, each with different benefits and drawbacks.
   - Among them is a technique called long polling
   - With long poll- ing, a client sends a standard HTTP request to the server, but instead of receiving the response right away, the server holds the request until an event is received from the sensor, which is then injected into the response returned to the client's request that was held idle. As soon as the client receives the response, it immediately sends a new request for an update, which will be held until the next update comes from the sensor, and so on.

5. Websockets

   - WebSocket is part of the HTML5 specification. The increasing support for HTML5 in most recent web and mobile web browsers means WebSocket is becoming ubiquitously available to all web apps
   - WebSockets enables a full-duplex communication channel over a single TCP connection. In plain English, this means that it creates a permanent link between the client and the server that both the client and the server can use to send messages to each other. Unlike techniques we've seen before, such as Comet, WebSocket is standard and opens a TCP socket. This means it doesn't need to encapsulate custom, non-web content in HTTP messages or keep the connection artificially alive as is needed with Comet implementations.

- A websockets starts out with a handshake: The first step is to send an HTTP call to the server with a special header asking for the protocol to be upgraded to WebSockets. If the web server sup- ports WebSockets, it will reply with a 101 Switch- ing Protocols status code, acknowledging the opening of a full-duplex TCP socket.
- Once the initial handshake takes place, the client and the server will be able to send messages back and forth over the open TCP connection; these messages are not HTTP messages but WebSockets data frames
- The overhead of each WebSockets data frame is 2 bytes, which is small compared to the 871-byte overhead of an HTTP message meta- data (headers and the like)
- the hierarchical structure of Things and their resources as URLs can be reused as-is for WebSockets.
- we can subscribe to events for a Thing's resource by using its corre- sponding URL and asking for a protocol upgrade to WebSockets. Moreover, Web- Sockets do not dictate the format of messages that are sent back and forth. This means we can happily use JSON and give messages the structure and semantics we want.
- Moreover, because WebSockets consist of an initial handshake followed by basic message framing layered over TCP, they can be directly implemented on many plat- forms supporting TCP/IP—not just web browsers. They can also be used to wrap sev- eral other internet-compatible protocols to make them web-compatible. One example is MQTT, a well-known pub/sub protocol for the IoT that can be inte- grated to the web of browsers via WebSockets
- The drawback, however, is that keeping a TCP connection permanently open can lead to an increase in battery consumption and is harder to scale than HTTP on the server side.

6. HTTP/2

    - This new version of HTTP allows multiplexing responses—that is, sending responses in parallel, This fixes the head-of-line blocking problem of HTTP/1.x where only one request can be outstanding on a TCP/IP connection at a time.

- HTTP/2 also introduces compressed headers using an efficient and low-memory compression format.
- Finally, HTTP/2 introduces the notion of server push. Concretely, this means that the server can provide content to clients without having to wait for them to send a request. In the long run, widespread adoption of server push over HTTP/2 might even remove the need for an additional protocol for push like WebSocket or webhooks.

### 1.1.3 SUMMARY

- When applied correctly, the REST architecture is an excellent substrate on which to create large-scale and flexible distributed systems.
- REST APIs are interesting and easily applicable to enable access to data and ser- vices of physical objects and other devices.
- Various mechanisms, such as content negotiation and caching of Hypermedia as the Engine of Application State (HATEOAS), can help in creating great APIs for Things.
- A five-step design process (integration strategy, resource design, representation design, interface design, and resource linking) allows anyone to create a mean- ingful REST API for Things based on industry best practices.
- The latest developments in the real-time web, such as WebSockets, allow creat- ing highly scalable, distributed, and heterogeneous real-time data processing applications. Devices that speak directly to the web can easily use web-based push messaging to stream their sensor data efficiently.
- HTTP/2 will bring a number of interesting optimizations for Things, such as multiplexing and compression.

## 1.2 Chapter 7

### 1.2.1 Connecting to the web

1. Direct Integration
   - The most straightforward integration pattern is the direct integration pattern. It can be used for devices that support HTTP

and TCP/IP and can therefore expose a web API directly. This pattern is particularly useful when a device can directly connect to the internet; for example, it uses Wi-Fi or Ethernet

2. Gateway Integration

   - Second, we explore the gateway integra- tion pattern, where resource-constrained devices can use non-web protocols to talk to a more powerful device (the gateway), which then exposes a REST API for those non-web devices. This pattern is particularly useful for devices that can't connect directly to the internet; for example, they support only Bluetooth or ZigBee or they have limited resources and can't serve HTTP requests directly.

3. Cloud Integration

   - Third, the cloud integration pattern allows a powerful and scalable web platform to act as a gateway. This is useful for any device that can connect to a cloud server over the internet, regardless of whether it uses HTTP or not, and that needs more capability than it would be able to offer alone.

### 1.2.2 Five step process

1. Integration strategy—Choose a pattern to integrate Things to the internet and the web. The patterns are presented in this chapter.
2. Resource design—Identify the functionality or services of a Thing, and organize the hierarchy of these services.
3. Representation design—Decide which representations will be served for each resource.
4. Interface design—Decide which commands are possible for each service, along with which error codes.
5. Resource linking design—Decide how the different resources are linked to each other.

1. Direct integration

   - the direct integration pattern is the perfect choice when the device isn't battery powered and when direct access from clients such as mobile web apps is required.

- the resource design. You first need to consider the physical resources on your device and map them into REST resources.
- The next step of the design process is the representation design. REST is agnostic of a par- ticular format or representation of the data. We mentioned that JSON is a must to guarantee interoperability, but it isn't the only interesting data representation available.
- a modular way based on the middleware pattern.
- In essence, a middleware can execute code that changes the request or response objects and can then decide to respond to the client or call the next middleware in the stack using the next() function.
- The core of this implementation is using the Object.observe() function.9 This allows you to asynchronously observe the changes happening to an object by registering a callback to be invoked whenever a change in the observed object is detected.

2. Gateway integration pattern

   - Gateway integration pattern. In this case, the web Thing can't directly offer a web API because the device might not support HTTP directly. An application gateway is working as a proxy for the Thing by offering a web API in the Thing's name. This API could be hosted on the router in the case of Bluetooth or on another device that exposes the web Thing API; for example, via CoAP.
   - The direct integration pattern worked well because your Pi was not battery powered, had access to a decent bandwidth (Wi-Fi/Ethernet), and had more than enough RAM and storage for Node. But not all devices are so lucky. Native sup- port for HTTP/WS or even TCP/IP isn't always possible or even desirable. For batterypowered devices, Wi-Fi or Ethernet is often too much of a power drag, so they need to rely on low-power protocols such as ZigBee or Bluetooth instead. Does it mean those devices can't be part of the Web of Things? Certainly not.
   - Such devices can also be part of the Web of Things as long as there' s an intermedi- ary somewhere that can expose the device's functionality through a WoT API like the one we described previously. These intermediaries are called application gateways (we'll

call them WoT gateways hereafter), and they can talk to Things using any non-web application protocols and then translate those into a clean REST WoT API that any HTTP client can use.

- They can add a layer of security or authentication, aggregate and store data temporarily, expose semantic descriptions for Things that don't have any, and so on.
- CoAP is a service layer protocol that is intended for use in resource-constrained internet devices, such as wireless sensor network nodes. CoAP is designed to easily translate to HTTP for simplified integration with the web
- CoAP is an interesting protocol based on REST, but because it isn't HTTP and uses UDP instead of TCP, a gateway that translates CoAP messages from/to HTTP is needed
- It's therefore ideal for device-to-device communi- cation over low-power radio communication, but you can't talk to a CoAP device from a JavaScript application in your browser without installing a special plugin or browser extension. Let's fix this by using your Pi as a WoT gateway to CoAP devices.
- By proxying, the gateway essentially just send a request to the CoAP device whenever the gateway receives a request and it'll return the value to the requester, once it receives a value from the CoAP device.

(a) Summary

   - For some devices, it might not make sense to support HTTP or WebSockets directly, or it might not even be possible, such as when they have very limited resources like memory or processing, when they can't connect to the internet directly (such as your Bluetooth activity tracker), or when they're battery-powered. Those devices will use more optimized communication or application protocols and thus will need to rely on a more powerful gateway that connects them to the Web of Things, such as your mobile phone to upload the data from your Bluetooth bracelet, by bridging/translating various protocols. Here we implemented a simple gateway from scratch using Express, but you could also use other open source alternatives such as OpenHab13 or The Thing System.

3. Cloud Integration pattern

    - Cloud integration pattern. In this pattern, the Thing can't directly offer a Web API. But a cloud service acts as a powerful application gateway, offering many more features in the name of the Thing. In this particular example, the web Thing connects via MQTT to a cloud service, which exposes the web Thing API via HTTP and the WebSockets API. Cloud services can also offer many additional features such as unlimited data storage, user management, data visualization, stream processing, support for many concurrent requests, and more.
    - Using a cloud server has several advantages. First, because it doesn't have the physical constraints of devices and gateways, it's much more scalable and can process and store a virtually unlimited amount of data. This also allows a cloud platform to support many protocols at the same time, handle protocol translation efficiently, and act as a scalable intermediary that can support many more concurrent clients than an IoT device could.
    - Second, those platforms can have many features that might take consid- erable time to build from scratch, from industry-grade security, to specialized analytics capabilities, to flexible data visualization tools and user and access management
    - Third, because those platforms are natively connected to the web, data and services from your devices can be easily integrated into third-party systems to extend your devices.

### 1.2.3 Summary

- There are three main integration patterns for connecting Things to the web: direct, gateway, and cloud.
- Regardless of the pattern you choose, you'll have to work through the following steps: resource design, representation design, and interface design.
- Direct integration allows local access to the web API of a Thing. You tried this by building an API for your Pi using the Express Node framework.
- The resource design step in Express was implemented using routes, each route representing the path to the resources of your Pi.

- We used the idea of middleware to implement support for different representa- tions— for example, JSON, MessagePack, and HTML—in the representation design step.
- The interface design step was implemented using HTTP verbs on routes as well as by integrating a WebSockets server using the ws Node module.
- Gateway integration allows integrating Things without web APIs (or not sup- porting web or even internet protocols) to the WoT by providing an API for them. You tried this by integrating a CoAP device via a gateway on your cloud.
- Cloud integration uses servers on the web to act as shadows or proxies for Things. They augment the API of Things with such features as scalability, analy- tics, and security. You tried this by using the EVRYTHNG cloud.

## 1.3 Chapter 8

- Having a single and common data model that all web Things can share would further increase interoperability and ease of integration by making it possible for applications and services to interact without the need to tailor the application manually for each specific device.
- The ability to easily discover and understand any entity of the Web of Things—what it is and what it does—is called findability.
- How to achieve such a level of interoperability—making web Things findable—is the purpose of the second layer
- The goal of the Find layer is to offer a uniform data model that all web Things can use to expose their metadata using only web standards and best practices.
- Metadata means the description of a web Thing, including the URL, name, current location, and status, and of the services it offers, such as sensors, actuators, com- mands, and properties
- this is useful for discovering web Things as they get con- nected to a local network or to the web. Second, it allows applications, services, and other web Things to search for and find new devices without installing a driver for that Thing

### 1.3.1 Findability problem

- For a Thing to be interacted with using HTTP and WebSocket requests, there are three fundamental problems

  1. How do we know where to send the requests, such as root URL/resources of a web Thing?
  2. How do we know what requests to send and how; for example, verbs and the format of payloads?
  3. How do we know the meaning of requests we send and responses we get, that is, semantics?

- The bootstrap problem. This problem is concerned with how the initial link between two entities on the Web of Things can be established.

- Lets assume the Thing can be found, how is it interacted with, if it exposes a UI at the root of its URL? In this case, a clean and user-centric web interface can solve problem 3 because humans would be able to read and understand how to do this.

- Problem 2 also would be taken care of by the web page, which would hardcode which request to send to which endpoint.

- But what if the heater has no user interface, only a RESTful API?1 Because Lena is an experienced front-end developer and never watches TV, she decides to build a sim- ple JavaScript app to control the heater. Now she faces the second problem: even though she knows the URL of the heater, how can she find out the structure of the heater API? What resources (endpoints) are available? Which verbs can she send to which resource? How can she specify the temperature she wants to set? How does she know if those parameters need to be in Celsius or Fahrenheit degrees?

### 1.3.2 Discovering Things

- The bootstrap problem deals with two scopes:

  1. first, how to find web Things that are physically nearby—for example, within the same local network
  2. second, how to find web Things that are not in the same local network—for example, find devices over the web.

1. Network discovery

    - In a computer network, the ability to automatically discover new participants is common.
    - In your LAN at home, as soon as a device connects to the network, it automatically gets an IP address using DHCP
    - Once the device has an IP address, it can then broadcast data packets that can be caught by other machines on the same network.
    - a broadcast or multicast of a message means that this message isn't sent to a particular IP address but rather to a group of addresses (multicast) or to everyone (broadcast), which is done over UDP.
    - This announcement process is called a network discovery protocol, and it allows devices and applications to find each other in local networks. This process is commonly used by various discovery protocols such as multicast Domain Name System (mDNS), Digital Living Network Alliance (DLNA), and Universal Plug and Play (UPnP).
    - Most internet-connected TVs and media players can use DLNA to discover network-attached storage (NAS)
    - your laptop can find and configure printers on your network with minimal effort thanks to network-level discovery protocols such as Apple Bonjour that are built into iOS and OSX.

    (a) mDNS

    - In mDNS, clients can discover new devices on a network by listening for mDNS mes- sages such as the one in the following listing. The client populates the local DNS tables as messages come in, so, once discovered, the new service—here a web page of a printer—can be used via its local IP address or via a URI usually ending with the .local domain. In this example, it would be `http://evt-bw-brother.local`.
    - The limitation of mDNS, and of most network-level discovery protocols, is that the network-level information can't be directly accessed from the web.

    (b) Network discovery on the web

- Because HTTP is an Application layer protocol, it doesn't know a thing about what's underneath—the network protocols used to shuffle HTTP requests around.
- The real question here is why the configu- ration and status of a router is only available through a web page for humans and not accessible via a REST API. Put simply, why don't all routers also offer a secure API where its configuration can be seen and changed by others' devices and applications in your network?
- Providing such an API is easy to do. For example, you can install an open-source operating system for routers such as OpenWrt and modify the software to expose the IP addresses assigned by the DHCP server of the router as a JSON document.
- This way, you use the existing HTTP server of your router to create an API that exposes the IP addresses of all the devices in your network. This makes sense because almost all networked devices today, from printers to routers, already come with a web user inter- face. Other devices and applications can then retrieve the list of IP addresses in the network via a simple HTTP call (step 2 in figure 8.3) and then retrieve the metadata of each device in the network by using their IP address (step 3 of figure 8.3).

(c) Resource discovery on the web

- Although network discovery does the job locally, it doesn't propagate beyond the boundaries of local networks.
- how do we find new Things when they connect, how do we understand the services they offer, and can we search for the right Things and their data in composite applications?
- On the web, new resources (pages) are discovered through hyperlinks. Search engines periodically parse all the pages in their database to find outgoing links to other pages. As soon as a link to a page not yet indexed is found, that new page is parsed and added to directory. This process is known as web crawling.

(d) Crawling

- From the root HTML page of the web Thing, the crawler can find the sub-resources, such as sensors and actuators, by discovering outgoing links and can then create a resource tree of the web Thing and all its resources. The crawler then uses the HTTP OPTIONS method to retrieve all verbs supported for each resource of the web Thing. Finally, the crawler uses content negotiation to understand which format is available for each resource.

(e) HATEOAS and web linking

- The simple way of crawling, of basically looping through links found is a good start, but it also has several limitations. First, all links are treated equally because there's no notion of the nature of a link; the link to the user interface and the link to the actuator resource look the same—they're just URLs.
- Additionally, it requires the web Thing to offer an HTML interface, which might be too heavy for resource-constrained devices. Finally, it also means that a client needs to both understand HTML and JSON to work with our web Things.
- A better solution for discovering the resources of any REST API is to use the HATEOAS principle to describe relationships between the various resources of a web Thing.
- A simple method to implement HATEOAS with REST APIs is to use the mechanism of web linking defined in RFC 5988. The idea is that the response to any HTTP request to a resource always contains a set of links to related resources—for example, the previous, next, or last page that contains the results of a search. These would be contained in the LINK header.
- encoding the links as HTTP headers introduces a more general framework to define relationships between resources outside the representation of the resource—directly at the HTTP level.
- When doing an HTTP GET on any Web Thing, the response should include a Link header that contains links to related resources. In particular, you should be able to get information about the device, its resources (API endpoints), and the documentation of the API using only Link headers.
- The URL of each resource is contained between angle brackets (<URL>) and the type of the link is denoted by rel="X", where X is the type of the rela- tion.

(f) New HATEOAS rel link things

- REL="MODEL" : This is a link to a Web Thing Model resource; see section 8.3.1.
- REL="TYPE" : This is a link to a resource that contains additional metadata about this web Thing.
- REL="HELP" : This relationship type is a link to the documentation, which means that a GET to devices.webofthings.io/help would return the documentation for the API in a human-friendly (HTML) or machine-readable (JSON) format.
- REL="UI" : This relationship type is a link to a graphical user interface (GUI) for interacting with the web Thing.

### 1.3.3 Describing web Things

- knowing only the root URL is insufficient to interact with the Web Thing API because we still need to solve the sec- ond problem mentioned at the beginning of this chapter: how can an application know which payloads to send to which resources of a web Thing?
- how can we formally describe the API offered by any web Thing?
- The simplest solution is to provide a written documentation for the API of your web Thing so that developers can use it (1 and 2 in figure 8.4).
- This approach, however, is insufficient to automatically find new devices, understand what they are, and what services they offer.
- In addition, manual implementation of the payloads is more error-prone because the developer needs to ensure that all the requests they send are valid
- By using a unique data model to define formally the API of any web Thing (the Web Thing Model), we'll have a powerful basis to describe not only the metadata but also the operations of any web Thing in a standard way (cases 3 and 4 of figure 8.4).
- This is the cornerstone of the Web of Things: creating a model to describe physical Things with the right balance between expressiveness—how flexible the model is—and usability— how easy it is to describe any web Thing with that model.

1. Introducing the Web Thing model

    - Once we find a web Thing and understand its API structure, we still need a method to describe what that device is and does. In other words, we need a conceptual model of a web Thing that can describe the resources of a web Thing using a set of well-known concepts.
    - In the previous chapters, we showed how to organize the resources of a web Thing using the /sensors and /actuators end points. But this works only for devices that actually have sensors and actuators, not for complex objects and scenarios that are com- mon in the real world that can't be mapped to actuators or sensors. To achieve this, the core model of the Web of Things must be easily applicable for any entity in the real world, ranging from packages in a truck, to collectible card games, to orange juice bot- tles. This section provides exactly such a model, which is called the Web Thing Model.

    (a) Entities
    - the Web of Things is composed of web Things.
    - A web Thing is a digital representation of a physical object—a Thing—accessible on the web. Think of it like this: your Facebook profile is a digital representation of yourself, so a web Thing is the "Facebook profile" of a physical object.
    - The web Thing is a web resource that can be hosted directly on the device, if it can connect to the web, or on an intermediate in the network such as a gateway or a cloud service that bridges non-web devices to the web.
    - All web Things should have the following resources:
        i. Model—A web Thing always has a set of metadata that defines various aspects about it such as its name, description, or configurations.
        ii. Properties—A property is a variable of a web Thing. Properties represent the internal state of a web Thing. Clients can subscribe to properties to receive a notification message when specific conditions are met; for example, the value of one or more properties changed.
        iii. Actions—An action is a function offered by a web Thing. Clients can invoke a function on a web Thing by sending

an action to the web Thing. Examples of actions are "open" or "close" for a garage door, "enable" or "disable" for a smoke alarm, and "scan" or "check in" for a bottle of soda or a place. The direc- tion of an action is from the client to the web Thing.

    iv. Things—A web Thing can be a gateway to other devices that don't have an inter- net connection. This resource contains all the web Things that are proxied by this web Thing. This is mainly used by clouds or gateways because they can proxy other devices.

i. Metadata
   - In the Web Thing Model, all web Things must have some associated metadata to describe what they are. This is a set of basic fields about a web Thing, including its identifiers, name, description, and tags, and also the set of resources it has, such as the actions and properties. A GET on the root URL of any web Thing always returns the metadata using this format, which is JSON by default

ii. Properties
   - Web Things can also have properties. A property is a collection of data values that relate to some aspect of the web Thing. Typically, you'd use properties to model any dynamic time series of data that a web Thing exposes, such as the current and past states of the web Thing or its sensor values—for example, the temperature or humidity sensor readings.

iii. Actions
   - Actions are another important type of resources of a web Thing because they represent the various commands that can be sent to that web Thing.
   - In theory, you could also use properties to change the status of a web Thing, but this can be a prob- lem when both an application and the web Thing itself want to edit the same property.
   - The actions object of the Web Thing Model has an object called resources, which contains all the types of actions (commands) supported by this web Thing.
   - Actions are sent to a web Thing with a POST to the URL

        of the action {WT}/actions/{id}, where id is the ID of the action

    iv. Things
        - a web Thing can act as a gateway between the web and devices that aren't connected to the internet. In this case, the gateway can expose the resources—properties, actions, and metadata—of those non-web Things using the web Thing.
        - The web Thing then acts as an Application-layer gateway for those non-web Things as it converts incoming HTTP requests for the devices into the various protocols or interfaces they support natively. For example, if your WoT Pi has a Bluetooth dongle, it can find and bridge Bluetooth devices nearby and expose them as web Things.
        - The resource that contains all the web Things proxied by a web Thing gateway is {WT}/things, and performing a GET on that resource will return the list of all web Things currently available

2. The WoT pie model

    - A new tree structure, fitting the discussed model, where the different sensors end up in /properties, setLedState ends up in /actions, we have no /things and /model is the metadata as well as all sensor data, their properties, the actions, everything.
    - Following the model allows for dynamically creating routes and such, as all information is maintained in the model of the Thing, /model, /properties, /actions, /things.

3. Summary

    - In this section, we introduced the Web Thing Model, a simple JSON-based data model for a web Thing and its resources. We also showed how to implement this model using Node.js and run it on a Raspberry Pi. We showed that this model is quite easy to understand and use, and yet is sufficiently flexible to represent all sorts of devices and products using a set of properties and actions. The goal is to propose a uniform way to describe web Things and their capabilities so that any HTTP client can find web Things and interact with them. This is sufficient for most use cases, and this model has all you need to be able to generate user interfaces for web Things automatically.

### 1.3.4 The Semantic Web of Things (Ontologies)

- In an ideal world, search engines and any other applications on the web could also understand the Web Thing Model. Given the root URL of a web Thing, any applica- tion could retrieve its JSON model and understand what the web Thing is and how to interact with it.
- The question now is how to expose the Web Thing Model using an existing web standard so that the resources are described in a way that means some- thing to other clients. The answer lies in the notion of the Semantic Web and, more precisely, the notion of linked data that we introduce in this section.
- Semantic Web refers to an extension of the web that promotes common data formats to facilitate meaningful data exchange between machines. Thanks to a set of stan- dards defined by the World Wide Web Consortium (W3C), web pages can offer a stan- dardized way to express relationships among them so that machines can understand the meaning and content of those pages. In other words, the Semantic Web makes it easier to find, share, reuse, and process information from any content on the web thanks to a common and extensible data description and interchange format.

1. Linked Data and RDFa
    - The HTML specification alone doesn't define a shared vocabulary that allows you to describe in a standard and non-ambiguous manner the elements on a page and what they relate to.

    (a) Linked Data
    - Enter the vision of linked data, which is a set of best practices for publishing and connecting structured data on the web, so that web resources can be interlinked in a way that allows computers to automatically understand the type and data of each resource.
    - This vision has been strongly driven by complex and heavy standards and tools centered on the Resource Description Framework (RDF)

- Although powerful and expressive, RDF would be overkill for most simple scenarios, and this is why a simpler method to structure con- tent on the web is desirable.
- RDFa emerged as a lighter version of RDF that can be embedded into HTML code
- Most search engines can use these annotations to generate better search listings and make it easier to find your websites.
- using RDFa to describe the metadata of a web Thing will make that web Thing findable and search- able by Google.

(b) RFDa

- vocab defines the vocabulary used for that element, in this case the Web of Things Model vocabulary defined previously.
- property defines the various fields of the model such as name, ID, or descrip- tion.
- typeof defines the type of those elements in relation to the vocabulary of the element.
- This allows other applications to parse the HTML representation of the device and automatically understand which resources are available and how they work.

(c) JSON-LD

- JSON-LD is an interesting and lightweight semantic annotation format for linked data that, unlike RDFa and Microdata, is based on JSON.29 It's a simple way to semanti- cally augment JSON documents by adding context information and hyperlinks for describing the semantics of the different elements of a JSON objects.

(d) Micro-summary

- This simple example already illustrates the essence of JSON-LD it gives a context to the content of a JSON document. As a consequence, all clients that understand the `http://schema.org/Product` context will be able to automatically process this informa- tion in a meaningful way. This is the case with search engines, for example. Google and Yahoo! process JSON-LD payloads using the Product schema to render special search results; as soon as it gets indexed, our Pi will be known by Google and Yahoo! as a Raspberry Pi product. This means that the more semantic data we add to our

Pi, the more findable it will become. As an example, try adding a location to your Pi using the Place schema,33 and it will eventually become findable by location.

We could also use this approach to create more specific schemas on top of the Web Thing Model; for instance, an agreed-upon schema for the data and functions a wash- ing machine or smart lock offers. This would facilitate discovery and enable automatic integration with more and more web clients.

### 1.3.5 Summary

- The ability to find nearby devices and services is essential in the Web of Things and is known as the bootstrap problem. Several protocols can help in discover- ing the root URL of Things, such as mDNS/Bonjour, QR codes or NFC tags.
- The last step of the web Things design process, resource linking design (also known as HATEOAS in REST terms), can be implemented using the web linking mechanism in HTTP headers.
- Beyond finding the root URL and sub-resources, client applications also need a mechanism to discover and understand what data or services a web Thing offers.
- The services of Things can be modeled as properties (variables), ac- tions (func- tions), and links. The Web Thing Model offers a simple, flexible, fully web-com- patible, and extensible data model to describe the details of any web Thing. This model is simple to adapt for your devices and easy to use for your products and applications.
- The Web Thing Model can be extended with more specific semantic descriptions such as those based on JSON-LD and available from the Schema.org repository.

## 1.4 Chapter 9

- In most cases, Internet of Things deployments involve a group of de- vices that com- municate with each other or with various applications within closed networks— rarely over open networks such as the inter- net. It would be fair to call such deploy- ments the "intranets of Things" because they're essentially isolated, private net- works that only a few entities can access. But the real power of the Web of Things lies in opening up these lonely silos and facilitating interconnection between devices and applications at a large scale.

- when it comes to public data such as data.gov initiatives, real-time traffic/weather/pollution conditions in a city, or a group of sensors deployed in a jungle or a volcano, it would be great to ensure that the general public or researchers anywhere in the world could access that data. This would enable anyone to create new innovative applications with it and possibly gener- ate substantial economic, environmental, and social value.
- How to share this data in secure and flexible way is what Layer 3 provides,
- The Share layer of the Web of Things. This layer focuses on how devices and their resources must be secured so that they can only be accessed by authorized users and applications.
- First, we'll show how Layer 3 of the WoT architecture covers the security of Things: how to ensure that only authorized parties can access a given resource. Then we'll show how to use existing trusted systems to allow sharing physical resources via the web.

### 1.4.1 Securing Things

- Ultimately, every security breach hurts the entire web because it erodes the overall trust of users in technology.
- Security in the Web of Things is even more critical than in the web. Because web Things are physical objects that will be deployed everywhere in the real world, the risks associated with IoT attacks can be catastrophic.
- Digitally augmented devices allow collecting fine-grained information about people, when they took their last insulin shot, their last jog and where they ran. It can also be used to remote control cars, houses and the like.
- the majority of IoT solutions don't comply with even the most basic security best practices; think clear-text passwords and communications, invalid certificates, old software versions with exploitable bugs, and so on.

1. Securing the IoT has three major problems

    - First, we must consider how to encrypt the communications between two enti- ties (for example, between an app and a web Thing) so that a malicious inter- ceptor—a "man in the middle"—can't access the data being transmitted in clear text. This is referred to as securing the channel
    - Second, we must find a way to ensure that when a client talks to a host, it can ensure that the host is really "himself"
    - Third, we must ensure that the correct access control is in place. We need to set up a method to control which user can access what resource of what server or Thing and when and then to ensure that the user is really who they claim to be.

2. Encryption 101

    - encryption is an essential ingredient for any secure system.
    - Without encryption, any attempt to secure a Thing will be in vain because attackers can sniff the communication and understand the security mechanisms that were put in place.

    (a) Symmetric Encryption

    - The oldest form of encoding a message is symmetric encryption. The idea is that the sender and receiver share a secret key that can be used to both encode and decode a message in a specific way

    (b) Assymetric Encryption

    - another method called asymmetric encryption has become popular because it doesn't require a secret to be shared between parties. This method uses two related keys, one public and the other private (secret)

3. Web Security with TLS: The S of HTTPS

    - Fortunately , there are standard protocols for securely encrypting data between clients and servers on the web.
    - The best known protocol for this is Secure Sockets Layer (SSL)
    - SSL 3.0 has a lot of vulnerabilities (Heartbleed and the like). These events inked the death of this proto- col, which was replaced by the much more secure but conceptually similar Transport Layer Security (TLS)

(a) TLS 101

- Despite its name, TLS is an Application layer protocol (see chapter 5). TLS not only secures HTTP (HTTPS) communication but is also the basis of secure WebSocket (WSS) and secure MQTT (MQTTS)
- First, it helps the client ensure that the server is who it says it is; this is the SSL/TLS authentication. Second, it guarantees that the data sent over the communication channel can't be read by any- one other than the client and the server involved in the transaction (also known as SSL/TLS encryption).
- The client, such as a mobile app, tells the server, such as a web Thing, which protocols and encryption algorithms it supports. This is somewhat similar to the content negotiation process we described in chapter 6.
- The server sends the public part of its certificate to the client. The goal here is for the client to make sure it knows who the server is. All web clients have a list of certificates they trust.12 In the case of your Pi, you can find them in /etc/ssl/certs. SSL certificates form a trust chain, meaning that if a client doesn't trust certificate S1 that the server sends back, but it trusts certificate S2 that was used to sign S1, the web client can accept S1 as well.
- The rest of the process generates a key from the public certificates. This key is then used to encrypt the data going back and forth between the server and the client in a secure manner. Because this process is dynamic, only the client and the server know how to decrypt the data they exchange during this session. This means the data is now securely encrypted: if an attacker manages to capture data packets, they will remain meaningless.

(b) Beyond Self-signed certificates

- Clearly, having to deal with all these security exceptions isn't nice, but these excep- tions exist for a reason: to warn clients that part of the security usually covered by SSL/ TLS can't be guaranteed with the certificate you generated. Basically, although the encryption of messages will work with a self-signed certificate (the one you created with the previous command), the authenticity of the server (the Pi) can't be guaran- teed. In consequence, the chain of trust is broken—problem 2

  - In an IoT context, this means that attackers could pretend to be the Thing you think you're talk- ing to.
  - The common way to generate certificates that guarantee the authenticity of the server is to get them from a well-known and trusted certificate authority (CA). There exists an amount of these; LetsEncrypt, Symantec and GeoTrust.

### 1.4.2 Authentication and access control

- Once we encrypt the communication between Things and clients as shown in the pre- vious section, we want to enable only some applications to access it.

- First, this means that the Things—or a gateway to which Things are connected—need to be able to know the sender of each request (identification).

- Second, devices need to trust that the sender really is who they claim to be (authentication)

- Third, the devices also need to know if they should accept or reject each request depending on the identity of this sender and which request has been sent (authorization).

1. Access control with REST and API tokens

   - Server-based authentication is used when we use our username/password to log into a website, we initiate a secure session with the server that's stored for a limited time in the server application's memory or in a local browser cookie.
   - server-based authentication is usually stateful because the state of the client is stored on the server. But as you saw in chapter 6, HTTP is a stateless protocol; therefore, using a server-based authentication method goes against this principle and poses certain problems. First, the performance and scalability of the overall systems are limited because each session must be stored in memory and over- head increases when there are many authenticated users. Second, this authentication method poses certain security risks—for example, cross-site request forgery.
   - alternative method called token-based authentication has become popular and is used by most web APIs.

- Because this token is added to the headers or query parameters of each HTTP request sent to the server, all interactions remain stateless.
- API tokens shouldn't be valid forever. API tokens, just like passwords, should change regularly.

2. OAuth: a web authorization framework

    - OAuth is an open standard for authorization and is essentially a mechanism for a web or mobile app to delegate the authentication of a user to a third-party trusted service; for example, Facebook, LinkedIn, or Google.
    - OAuth dynamically generates access tokens using only web protocols.
    - OUath allows sharing resources and token sharing between applications.
    - In short, OAuth standardizes how to authenticate users, generate tokens with an expiration date, regenerate tokens, and provide access to resources in a secure and standard manner over the web.
    - At the end of the token exchange process, the application will know who the user is and will be able to access resources on the resource server on behalf of the user. The application can then also renew the token before it expires using an optional refresh token or by running the authorization process again.
    - OAuth delegated authentication and access flow. The application asks the user if they want to give it access to resources on a third-party trusted service (resource server). If the user accepts, an authorization grant code is generated. This code can be exchanged for an access token with the authorization server. To make sure the authorization server knows the application, the application has to send an app ID and app secret along with the authorization grant code. The access token can then be used to access protected resources within a certain scope from the resource server.
    - Implementing an OAuth server on a Linux-based embedded device such as the Pi or the Intel Edison isn't hard because the protocol isn't really heavy. But maintaining the list of all applications,

users, and their access scope on each Thing is clearly not going to work and scale for the IoT.

(a) OAuth Roles

- A typical OAuth scenario involves four roles
    i. A resource owner—This is the user who wants to authorize an application to access one of their trusted accounts; for example, your Facebook account.
    ii. The resource server—Is the server providing access to the resources the user wants to share? In essence, this is a web API accepting OAuth tokens as credentials.
    iii. The authorization server—This is the OAuth server managing authorizations to

access the resources. It's a web server offering an OAuth API to authenticate and authorize users. In some cases, the resource server and the authorization server can be the same, such as in the case of Facebook.

i. The application—This is the web or mobile application that wants to access the resources of the user. To keep the trust chain, the application has to be known by the authorization server in advance and has to authenticate itself using a secret token, which is an API key known only by the authorization server and the application.

### 1.4.3 The Social Web of Things

- Using OAuth to manage access control to Things is tempting, but not if each Thing has to maintain its own list of users and application. This is where the gateway integration pattern can be used.
- use the notion of delegated authentication offered by OAuth, which allows you to use the accounts you already have with OAuth providers you trust, such as Facebook, Twitter, or LinkedIn.
- The Social Web of Things is usually what covers the sharing of access to devices via existing social network relationships.

1. A Social Web of Things authentication proxy

- The idea of the Social Web of Things is to create an authentication proxy that controls access to all Things it proxies by identifying users of client applications using trusted third-party services.
- Again, we have four actors: a Thing, a user using a client application, an authenti- cation proxy, and a social network (or any other service with an OAuth server). The client app can use the authentication proxy and the social network to access resources on the Thing. This concept can be implemented in three phases:
  (a) The first phase is the Thing proxy trust. The goal here is to ensure that the proxy can access resources on the Thing securely. If the Thing is protected by an API token (device token), it could be as simple as storing this token on the proxy. If the Thing is also an OAuth server, this step follows an OAuth authentication flow, as shown in figure 9.6. Regardless of the method used to authenticate, after this phase the auth proxy has a secret that lets it access the resources of the Thing.
  (b) The second phase is the delegated authentication step. Here, the user in the client app authenticates via an OAuth authorization server as in figure 9.6. The authentication proxy uses the access token returned by the authorization server to identify the user of the client app and checks to see if the user is authorized to access the Thing. If so, the proxy returns the access token or generates a new one to the client app.
  (c) The last phase is the proxied access step. Once the client app has a token, it can use it to access the resources of the Thing through the authentication proxy. If the token is valid, the authentication proxy will forward the request to the Thing using the secret (device token) it got in phase 1 and send the response back to the client app.
- All communication is encrypted using TLS
- Social Web of Things authentication proxy: the auth proxy first establishes a secret with the Thing over a secure channel. Then, a client app requests access to a resource via the auth proxy. It authenticates itself via an OAuth server (here Facebook) and gets back an access token. This token is then used to access resources on the Thing via the auth proxy. For instance, the /temp resource is requested by the client app and given access via the auth proxy

forwarding the request to the Thing and relaying the response to the client app.

2. Leveraging Social Networks

    - This is the very idea of the Social Web of Things: instead of creating abstract access control lists, we can reuse existing social structures as a basis for sharing our Things. Because social networks increasingly reflect our social relationships, we can reuse that knowledge to share access to our Things with friends via Facebook, or work colleagues via LinkedIn.

3. Implementing Access Control Lists

    - In essence, you need to create an access control list (ACL). There are various ways to implement ACLs, such as by storing them in the local database.

4. Proxying Resources of Things

    - Finally, you need to implement the actual proxying: once a request is deemed valid by the middleware, you need to contact the Thing that serves this resource and proxy the results back to the client.

### 1.4.4 Beyond book

- But just as HTTP might be too heavy for resource-limited devices, security pro- tocols such as TLS and their underlying cypher suites are too heavy for the most resource-constrained devices. This is why lighter-weight versions of TLS are being developed, such as DTLS,26 which is similar to TLS but runs on top of UDP instead of TCP and also has a smaller memory footprint

- device democracy.27 In this model, devices become more autonomous and favor peer-to-peer interactions over centralized cloud services. Security is ensured using a blockchain mechanism: similar to the way bitcoin transactions are validated by a number of independent parties in the bitcoin network, devices could all participate in making the IoT secure.

### 1.4.5 Summary

- You must cover four basic principles to secure IoT systems: encrypted commu- nication, server authentication, client authentication, and access control.
- Encrypted communication ensures attackers can't read the content of mes- sages. It uses encryption mechanisms based on symmetric or asymmetric keys.
- You should use TLS to encrypt messages on the web. TLS is based on asymmetric keys: a public key and a private server key.
- Server authentication ensures attackers can't pretend to be the server. On the web, this is achieved by using SSL (TLS) certificates. The delivery of these certif- icates is controlled through a chain of trust where only trusted parties called certificate authorities can deliver certificates to identify web servers.
- Instead of buying certificates from a trusted third party, you can create self- signed TLS certificates on a Raspberry Pi. The drawback is that web browsers will flag the communication as unsecure because they don't have the CA certifi- cate in their trust store.
- You can achieve client authentication using simple API tokens. Tokens should rotate on a regular basis and should be generated using crypto secure random algorithms so that their sequence can't be guessed.
- The OAuth protocol can be used to generate API tokens in a dynamic, standard, and secure manner and is supported by many embedded Linux devices such as the Raspberry Pi.
- The delegated authentication mechanism of OAuth relies on other OAuth pro- viders to authenticate users and create API tokens. As an example, a user of a Thing can be identified using Facebook via OAuth.
- You can implement access control for Things to reflect your social contacts by creating an authentication proxy using OAuth for clients' authentication and contacts from social networks.