Developing Web APIs for Things

Niels Olof Bouvin

Overview

- **RESTful design for WoT**
- **Real time data for the Web of Things**
- **Web based Things**
- **Gateways for Things**
- **Things in the Cloud**

Requirements for (WoT) Things

- It should be possible to access it
- It should be possible to access its constituent parts
- Representation should be tailored according to need
- Operations on things should follow predictable patterns
- It should be possible to discover the parts of a Thing as well as the operations supported by those parts

Representational State Transfer

- **By following the REST principles, it becomes easier to create maintainable, interoperable and scalable internet services**
- **REST is an architectural *style,* not a protocol**
  - so it consists of a series of guidelines, constraints, and principles

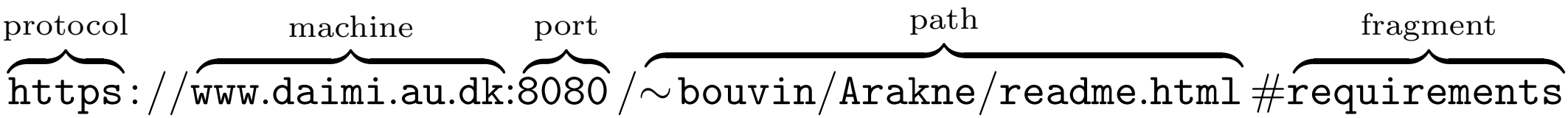
REST requirements

- **Client/Server**
  - all communication is strictly request/response
- **Uniform interfaces (i.e., APIs)**
  - following established practices enables faster learning & fewer surprises
- **Stateless (i.e., sessionless)**
  - the client holds its own state; it is communicated as needed in every request
- **Cacheable**
  - servers can indicate that data may be cacheable—this can improve performance
- **Layered architecture**
  - the above ensures that a multi-layered architecture can be used, which can greatly improve performance (e.g., through proxies, load balancers, gateways, or CDNs)

Naming

- Naming is a basic problem of computer science
- How to define a naming scheme that provides unique resources with unambiguous identifiers?
- Flat naming space: unique identifiers
  - `fb90ed86136c771386639bbb5f816eeb2a2d2a3a`
- Structured name space: often hierarchical
  - `/web/bouvin/public_html_cs/dBIoTP2PC/2017/milestones/milestone-3.html`
- Attribute-based name space: essentially a query
  - `/C=DK/O=Aarhus Universitet/OU=Computer Science`

Uniform Resource Locators

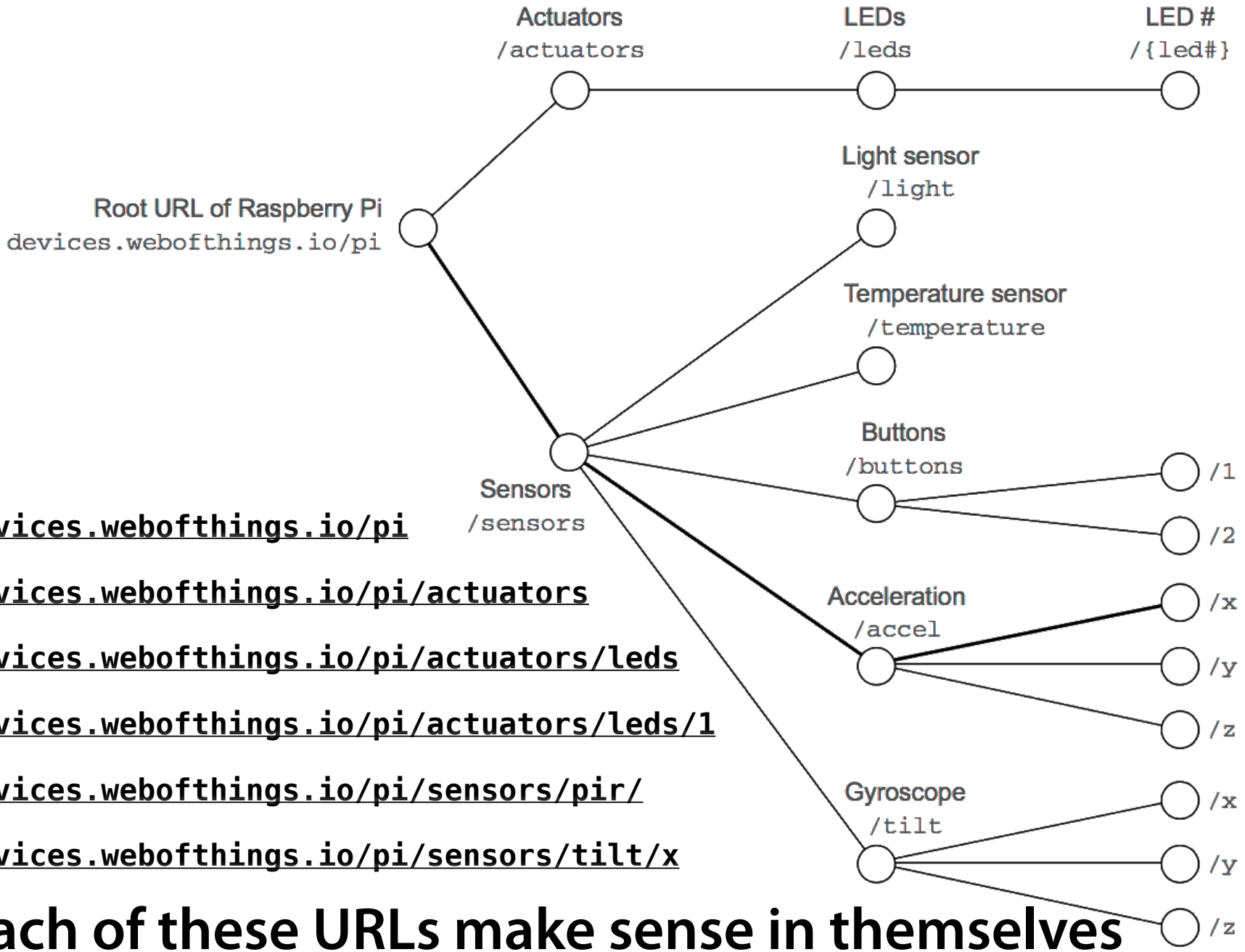


- A twice structured naming scheme
- Originally, path often mapped to folders and files
  - `https://users-cs.au.dk/bouvin/dBIoTP2PC/2017/milestones/milestone-3.html`
  - `/web/bouvin/public_html_cs/dBIoTP2PC/2017/milestones/milestone-3.html`
- The path of the URL is now interpreted in a far more flexible manner, exploiting the hierarchical nature to represent not only arbitrary resources, but also their constituent parts
- URLs have thus become part of an API

Consistency in naming

- **Interoperability implies predictability—APIs should be structured compliant with established practices so other developers can quickly adapt and adopt**
- **Simple is better than complex**
- **Resources are *identifiers*, not *operations***
  - http://devices.webofthings.io/pi/sensors/temperature/
  - http://devices.webofthings.io/pi/sensors/light
  - http://devices.webofthings.io/pi/actuators/leds/1/

The strength of hierarchical names



- devices.webofthings.io/pi
- devices.webofthings.io/pi/actuators
- devices.webofthings.io/pi/actuators/leds
- devices.webofthings.io/pi/actuators/leds/1
- devices.webofthings.io/pi/sensors/pir/
- devices.webofthings.io/pi/sensors/tilt/x

- **Each of these URLs make sense in themselves**

Resources are identifiers, not methods

- **Names should be descriptive**
- **Names should be nouns, not verbs**
  - it is the thing you wish to operate on, not the operation
- **If there are several resources of the same type, they are addressed first as aggregate, second as individual**
  - devices.webofthings.io/pi/actuators/leds
  - devices.webofthings.io/pi/actuators/leds/1

Design consequences for WoT Things

- A Thing is a web server
- A Thing should use good security practices
- A Thing should have a root resource
- A Thing should make its properties accessible through a hierarchical naming scheme

Resources and their representations

- **A resource is any kind of data that needs to be addressed**
  - it must be given a unique URL as just outlined
- **Resources are presented and manipulated through their representation, which does not have to be what is actually kept at the server**
  - e.g., a row in a SQL database or a measurement from a sensor could be represented as HTML, JSON, or otherwise
  - it should be whatever the client and the server can understand
- **This makes REST APIs very flexible**

Design consequences for WoT Things

- **A Thing should use JSON as the default representation**
  - naming scheme should be camelCase: lastValue rather than last-value
- **A Thing should use UTF8 for text encoding**
- **A Thing might provide HTML representations of their resources for easier inspection and control**

Operating on resources

- Just as a simple naming scheme simplifies interoperability, restricting the types of possible operations to a fixed set makes it easier to write clients and servers
- Following the rules for each methods ensures consistent behaviour with other systems

Operating on resources

- **HTTP supports four main methods:**
- GET
  - retrieve the state of a resource — don't modify it
- POST
  - create *new* resource, do not specify identifier
- PUT
  - update *existing* resource, or create new resource with an identifier
- DELETE
  - remove a resource

GET

- **Retrieve the resource in desired representation**
- **Safe and idempotent**
  - does not modify the resource; can be repeated over and over expecting similar result
  - **HEAD** is the same, but returns only the headers: handy, if checking for updates

Request:

```
GET /pi/sensors/temperature/value
Accept: application/json
Host: devices.webofthings.io
```

Response:

```
200 OK HTTP/1.1
Content-Type: application/json
{"temperature" : 37}
```

POST

- **Creates a *new* resource**
- **Unsafe and non-idempotent**
  - changes status on server; repeated calls will have different results

Request:

```
POST /pi/display/messages HTTP/1.1
Host: devices.webofthings.io
Content-Type: application/json
{"content":"Hello World!”,"duration":30}
```

Response:

```
201 Created HTTP/1.1
Location: devices.webofthings.io/pi/display/messages/2210
```

  - **Location** points to the newly created resource

PUT

- **Updates an existing resource**
- **Unsafe and idempotent**
  - changes status on server; repeated calls will have the same result

Request:

```
PUT /pi/actuators/leds/4 HTTP/1.1
Host: devices.webofthings.io
Content-Type: application/json
{"red" : 0, "green" : 128, "blue" : 128}
```

Response:

```
200 OK HTTP/1.1
```

PATCH

- **Updates part of an existing resource**
- **Unsafe and idempotent**
  - changes status on server; repeated calls will have the same result

Request:

```
PUT /pi/actuators/leds/4 HTTP/1.1
Host: devices.webofthings.io
Content-Type: application/json
{"blue" : 128}
```

Response:

```
200 OK HTTP/1.1
```

DELETE

- **Deletes an existing resource**
- **Unsafe and idempotent**
  - changes status on server; repeated calls will have the same result

Request:

```
DELETE /rules/24 HTTP/1.1
Host: devices.webofthings.io
```

Response:

```
200 OK HTTP/1.1
```

OPTION

- **Lists the permitted operations on a resource**
- **Safe and idempotent**
  - No changes on server; repeated calls will have the same result

Request:

```
OPTIONS pi/sensors/humidity/ HTTP/1.1
Host: devices.webofthings.io
```

Response:

```
204 No Content HTTP/1.1
Content-Length: 0
Allow: GET, OPTIONS
Accept-Ranges: bytes
```

A selection of HTTP status codes

- **`200 OK`**
  - Standard response for successful HTTP requests.
- **`201 Created`**
  - The request has been fulfilled and resulted in a new resource being created
- **`202 Accepted`**
  - The request has been accepted, but is not yet fulfilled
- **`301 Moved Permanently`**
  - This and all future requests should be directed to the given URI
- **`404 Not Found`**
  - The requested resource could not be found
- **`500 Internal Server Error`**
  - The server has experienced an error, and could not fulfil the request

Cross-Origin Resource Sharing

- **JavaScript in the browser can by default only access its own origin server**
  - this makes excellent general security sense, but is inconvenient for accessing APIs
- **CORS addresses this:**

Request:

```
GET /pi/sensors/temperature HTTP/1.1
Host: devices.webofthings.io
Origin: http://localhost:8090
```

Response:

```
200 OK HTTP/1.1
Access-Control-Allow-Origin: "*"
```

- **If the A-C-A-O header matches, the browser permits it**

Design consequences for WoT Things

- A Thing should support standard HTTP methods
- A Thing should always support GET on its root
- A Thing should use standard HTTP error codes
- A Thing should support CORS, so it can be integrated

HATEOAS

- Hypermedia as the Engine of Application State
- The Web is hypermedia, and we can enable the discovery of the API through the use of links
- Each layer should provide links to the layers above and below, as well as links to possible actions
  - just as a well-structured (HTML) Web page should provide all relevant links

```
<html><body>
  <h1 class="device-name">Raspberry Pi</h1>
  <a href="http://devices.webofthings.io/pi" class="self">Root URL</a> of
   this device.
  View the list of <a href="sensors/" class="sensors">sensors</a> and
  <a href="actuators/" class="actuators">actuators</a> on this device.
  You can also view all <a href="links/" class="links">links</a> available on this
device.  Or read the <a href="about/" class="help">documentation</a>.
</body></html>
```

‘links’ yields this JSON fragment

```
{
  “links”:{
    "sensors": {
      "link": "http://devices.webofthings.io/pi/sensors/",
      "title": "List of Sensors"
    },
    "actions": {
      "link": " http://devices.webofthings.io/pi/actions/",
      "title": "List of actions"
    },
    "meta": {
      "link": "http://w3c.org/schemas/webofthings/",
      "title": "Metadata"
    },
    "self": {
      "link": " http://devices.webofthings.io/pi/",
      "title": "Self"
    },
    "help": {
      "link": "http://webofthings.io/docs/pi/",
      "title": "Documentation”
    },
    "ui": {
      "link": " http://devices.webofthings.io/pi/",
      "title": "User Interface"
    }
  }
}
```

- **Easy to parse with JavaScript in order to go deeper**

Design consequences for WoT Things

- **A Thing should always offer links to related resources, enabling discovery by human and computer**
  - this can aid the developer, and make clients more flexible
  - fully automated APIs are probably a little optimistic, but within defined areas this should be able to work
- **A Thing can by supporting OPTIONS ease the discovery of what can be done with its resources**

Overview

- RESTful design for WoT
- **Real time data for the Web of Things**
- **Web based Things**
- **Gateways for Things**
- **Things in the Cloud**

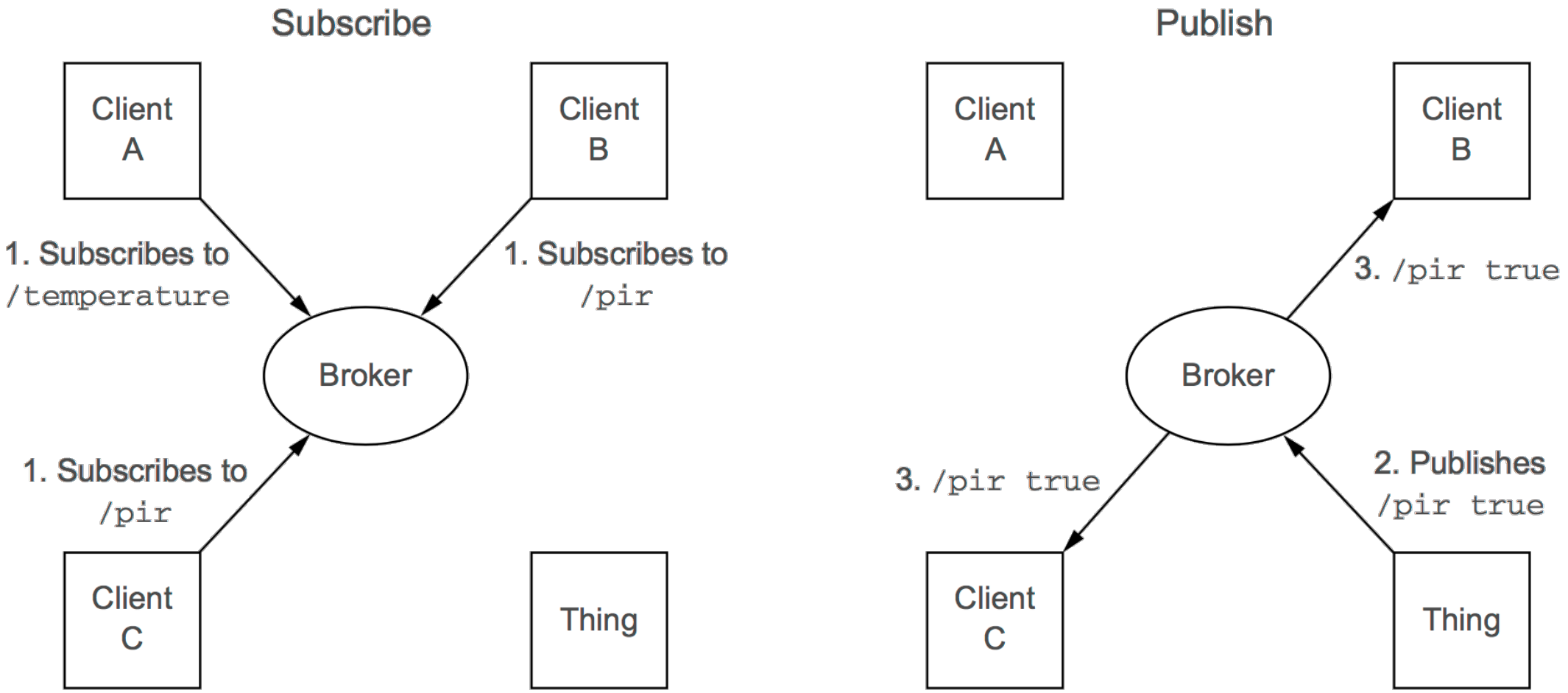
REST is Request/Response

- Request/Response is well suited for many purposes
- But… how can the server announce a change or event if the client does not request an update?
- How can events be efficiently propagated from server to client?

Busy waiting

- **The client requests the resource periodically**
  - highly inefficient, even if the client only uses HEAD to detect changes
- **Good enough for slowly changing parameters, but not for, e.g., button presses or IR detectors**

Publish/subscribe



- **Ideally, clients interested in a change should be able to register that interest, and then be alerted upon it**

WebSockets

- **Opens a permanent connection between Web browser and server**
  - supported by all modern Web browsers
- **Once established, data frames can be sent back and forth through JavaScript**
  - the format is not http—the overhead per frame is 2 bytes, much less than http
- **More efficient in data traffic, though having an open connection might be a battery strain for a small device**
  - (there are many libraries, socket.io seems to be popular for Node.js)

Conclusions

- **The careful observance of established REST practices enables Things to provide a well defined API that can be explored by human and program alike**
- **This eases development and makes for robust systems**
- **The approach described is a style, rather than a law, and some elements are in contention**
  - especially HATEAOS, though the general concept of having an explorable API is a good one

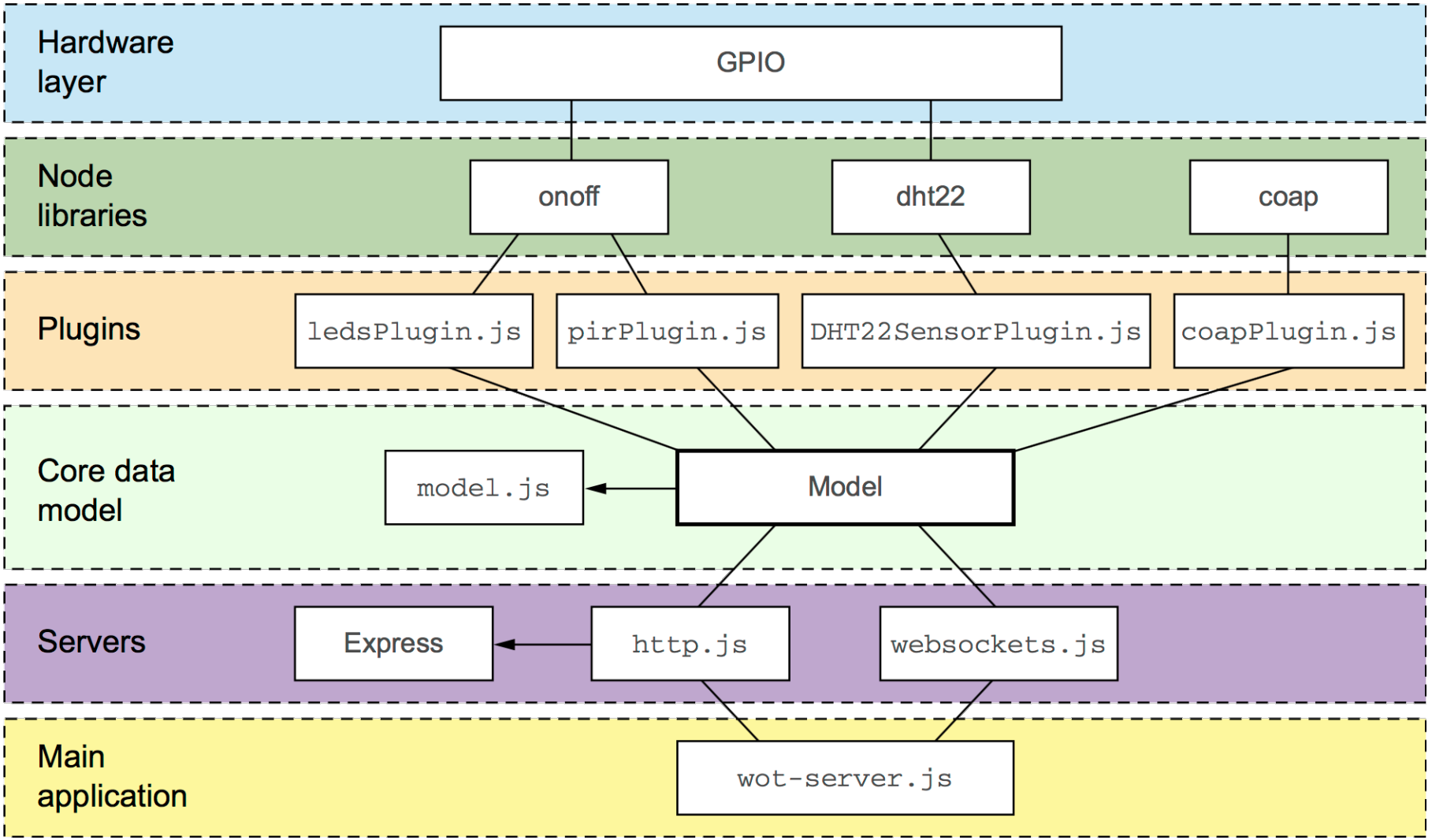
Overview

- RESTful design for WoT
- Real time data for the Web of Things
- **Web based Things**
- **Gateways for Things**
- **Things in the Cloud**

Web based Things

- **The most straightforward**
  - the device sensing/actuating reports its own state and can accept commands
- **Requirements**
  - sufficient processing power to handle a HTTP stack—though that can be pretty light
  - adequately open to permit installation of a HTTP stack
  - always online (if we need to access it)—expensive, if battery powered

Architecture of a WoT Thing



- **Arrange your application in layers for clear separation of concerns, and easy future extensibility**

Well structured code

- Dividing the code across meaningful modules, files, and folders makes it easier to navigate, understand, and maintain

```
wot-pi
├── middleware
|   └── converter.js
├── package.json
├── plugins
|   ├── external
|   |   └── coapPlugin.js
|   └── internal
|       ├── DHT22SensorPlugin.js
|       ├── ledsPlugin.js
|       └── pirPlugin.js
├── resources
|   ├── model.js
|   └── resources.json
├── routes
|   ├── actuators.js
|   ├── sensors.js
|   └── things.js
├── servers
|   ├── coap.js
|   ├── http.js
|   └── websockets.js
├── utils
|   └── utils.js
└── wot-server.js
```

Configuration in `resources.json`, not code

- Accessible through `model.js`
- Quick overview of available resources
- Updated by the sensor plugins
- Used throughout the code
- Eases extension and modification

```
{
  "pi": {
    "name": "WoT Pi",
    "description": "A simple WoT-connected Raspberry PI for the WoT book.",
    "port": 8484,
    "sensors": {
      "temperature": {
        "name": "Temperature Sensor",
        "description": "An ambient temperature sensor.",
        "unit": "celsius",
        "value": 0,
        "gpio": 12
      },
      "humidity": {
        "name": "Humidity Sensor",
        "description": "An ambient humidity sensor.",
        "unit": "%",
        "value": 0,
        "gpio": 12
      },
      "pir": {
        "name": "Passive Infrared",
        "description": "A passive infrared sensor. When 'true' someone is present.",
        "value": true,
        "gpio": 17
      }
    },
    "actuators": {
      "leds": {
        "1": {
          "name": "LED 1",
          "value": false,
          "gpio": 4
        },
        "2": {
          "name": "LED 2",
          "value": false,
          "gpio": 9
        }
      }
    }
  }
}
```

Talking in different protocols & formats

- **The rich JavaScript/Express ecosystem makes it fairly simple to support different protocols**
  - HTTP, websockets, CoAP, MQTT
- **as well as different representations**
  - HTML, JSON, MessagePack, …

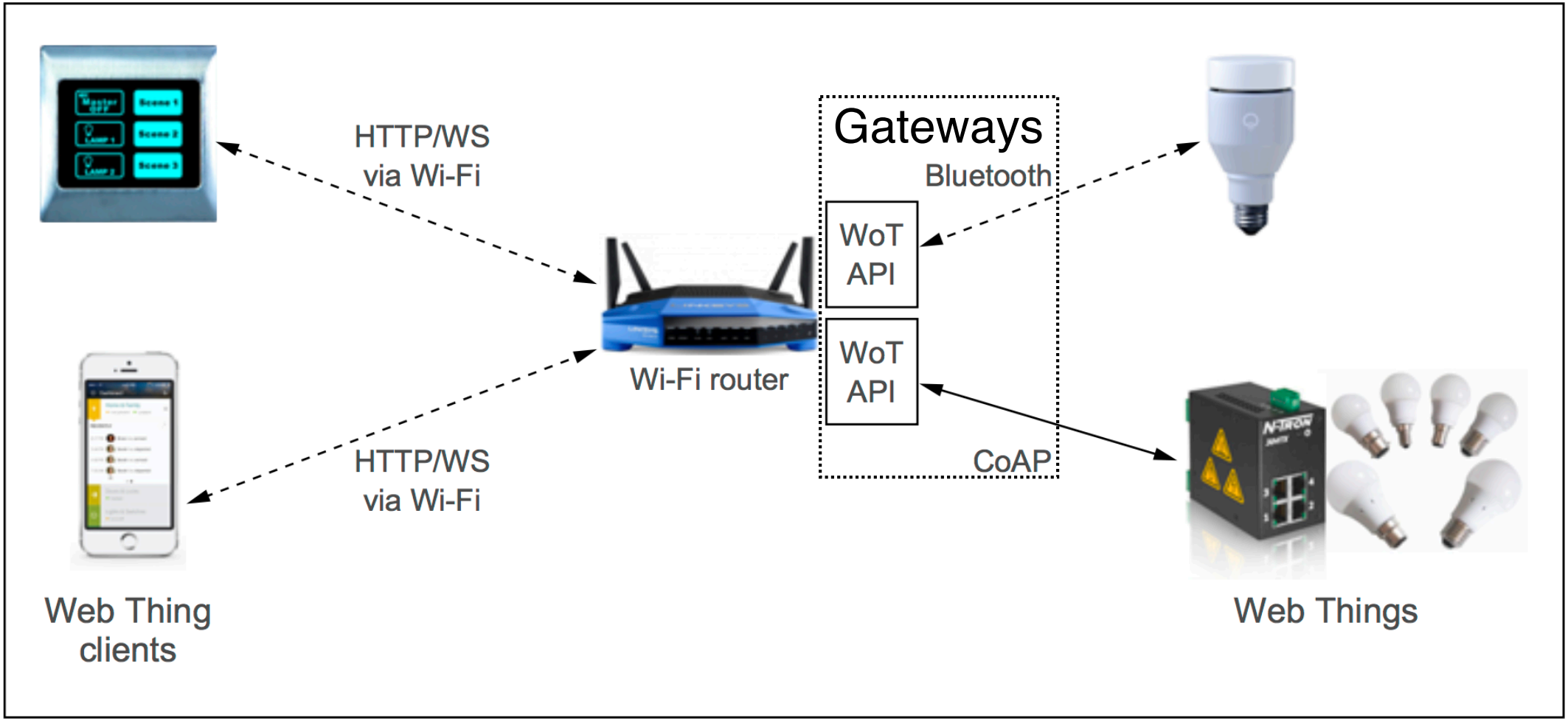
Overview

- RESTful design for WoT
- Real time data for the Web of Things
- Web based Things
- **Gateways for Things**
- **Things in the Cloud**

Gateways

- **Not all things can connect to the Web or the Internet**
  - they may be limited in power and capabilities, or they may be using a whole different protocol stack
- **Someone must cover for them, and provide access on their behalf**

A gateway example



- The gateways translate between Web protocols and the specialised protocols, here Bluetooth and CoAP

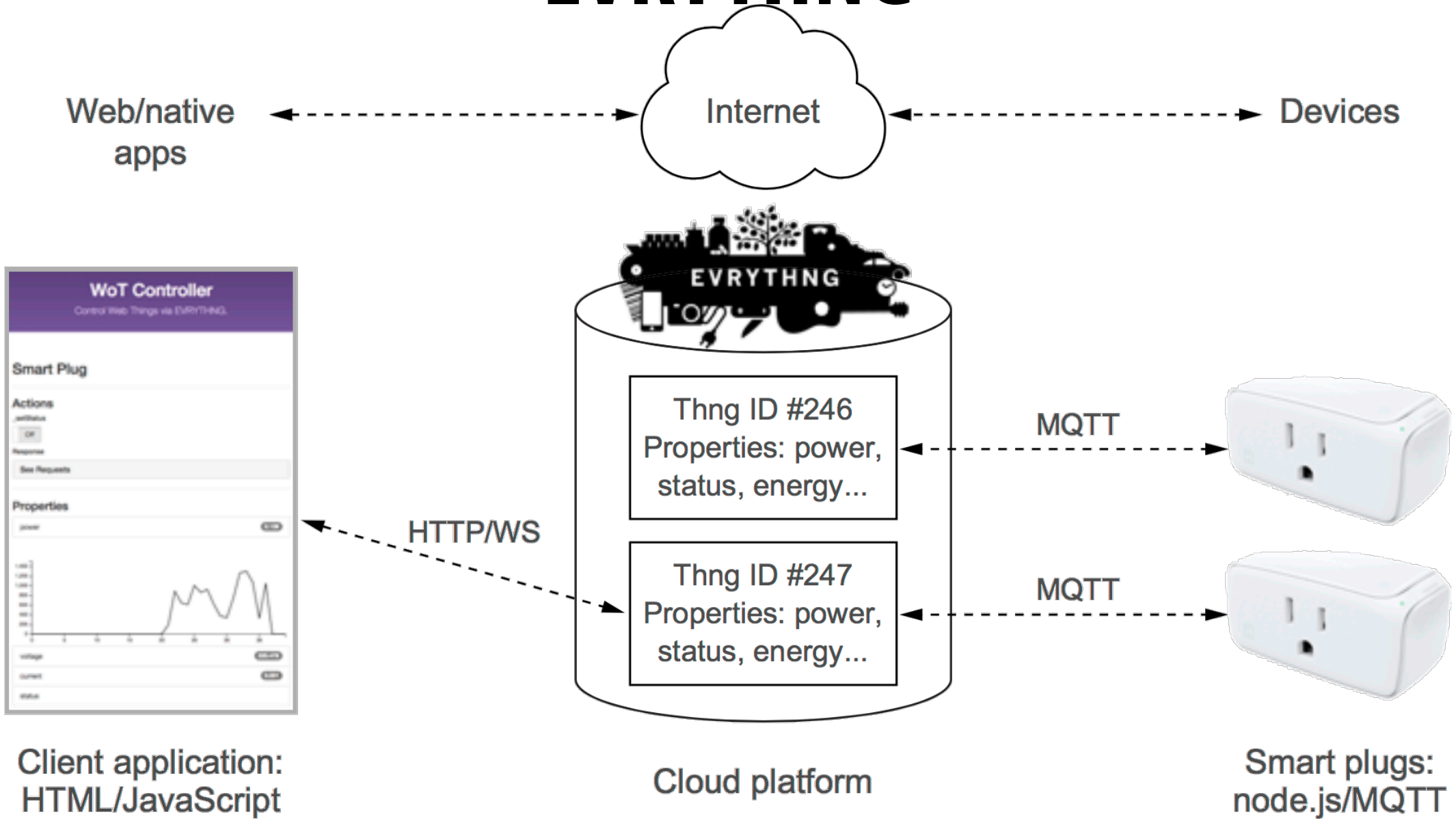
Overview

- RESTful design for WoT
- Real time data for the Web of Things
- Web based Things
- Gateways for Things
- **Things in the Cloud**

Cloud-based solutions

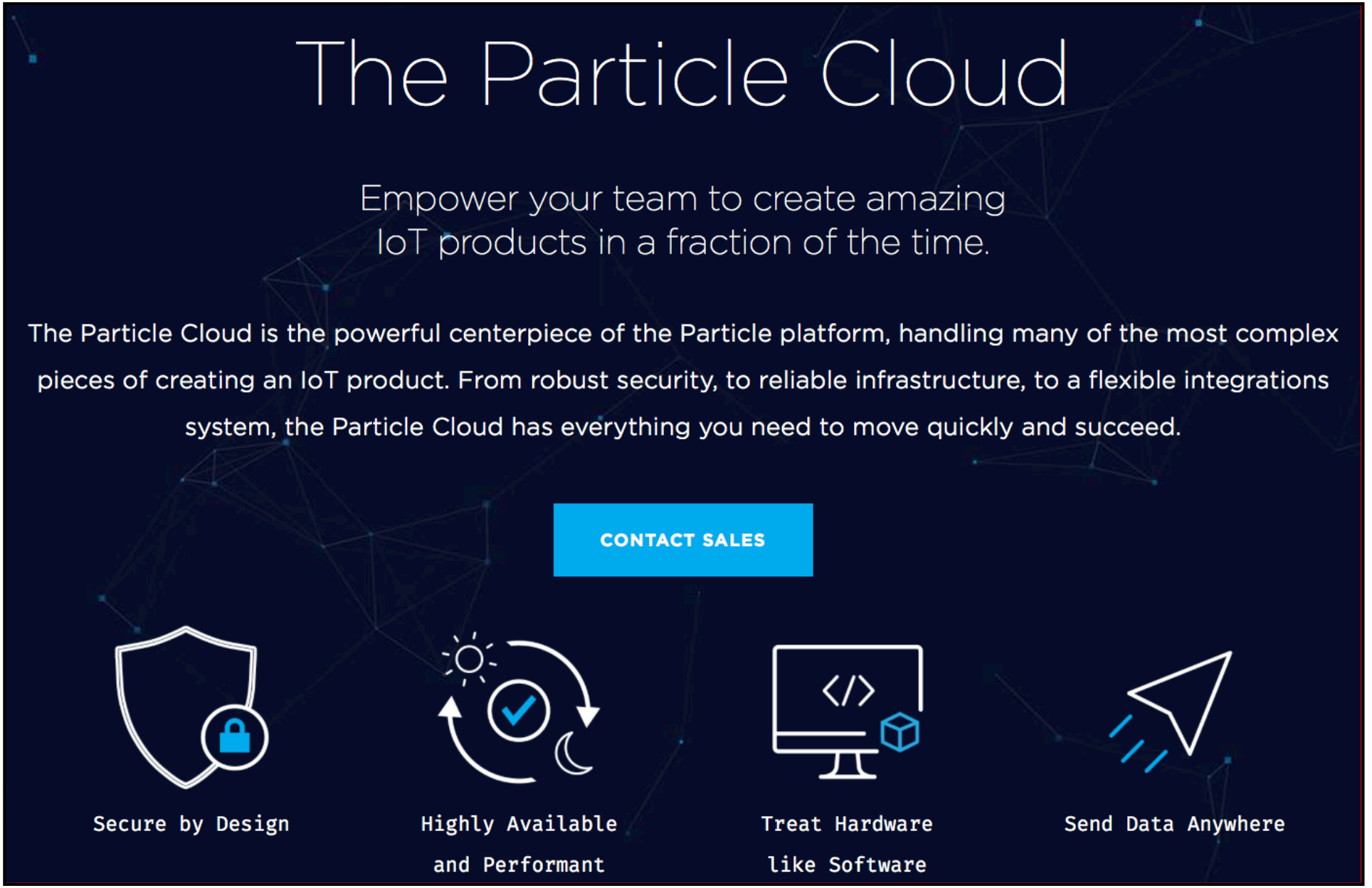
- **Cloud services exist for many, *many* purposes**
- **Some of these services are concerned with the access to and data collection from IoT Things**
  - as well as services such as authentication, data visualisation and analysis
- **If the cloud servers are dimensioned correctly, this can scale very well**
  - for a price
  - remember: a cloud server is just someone else's computer

EVRYTHNG



- **One example of an IoT oriented cloud service**

particle.io



- And another (also works with Raspberry Pi)

Cloud services

- **Excellent for centralised activities**
  - authentication
  - storing and processing data at large scale
  - letting someone else bother with hosting and backup
  - providing a single entry point (no firewall to contend with, stable URL)
- **Some concerns**
  - cost (someone must be paying the bills)
  - some privacy concerns (who else can access the data?)
  - vendor lock-in (once you have committed to one provider, it may be difficult to move)

Conclusions

- **A Thing can be connected to the Web**
  - directly
  - through a gateway
  - through a cloud service
- **Keeping a clear separation between layers and APIs makes life far simpler**

The choices of strategy & architecture

- **The Web of Things can be realised in a number of ways:**
  - running on the Thing itself
  - façading the Thing with a Web based gateway
  - relying on Cloud services to provide access
- **These choices come with consequences and possibilities**