The Web of Things

Course website: http://site.unibo.it/iot

Prof. Luciano Bononi
luciano.bononi@unibo.it

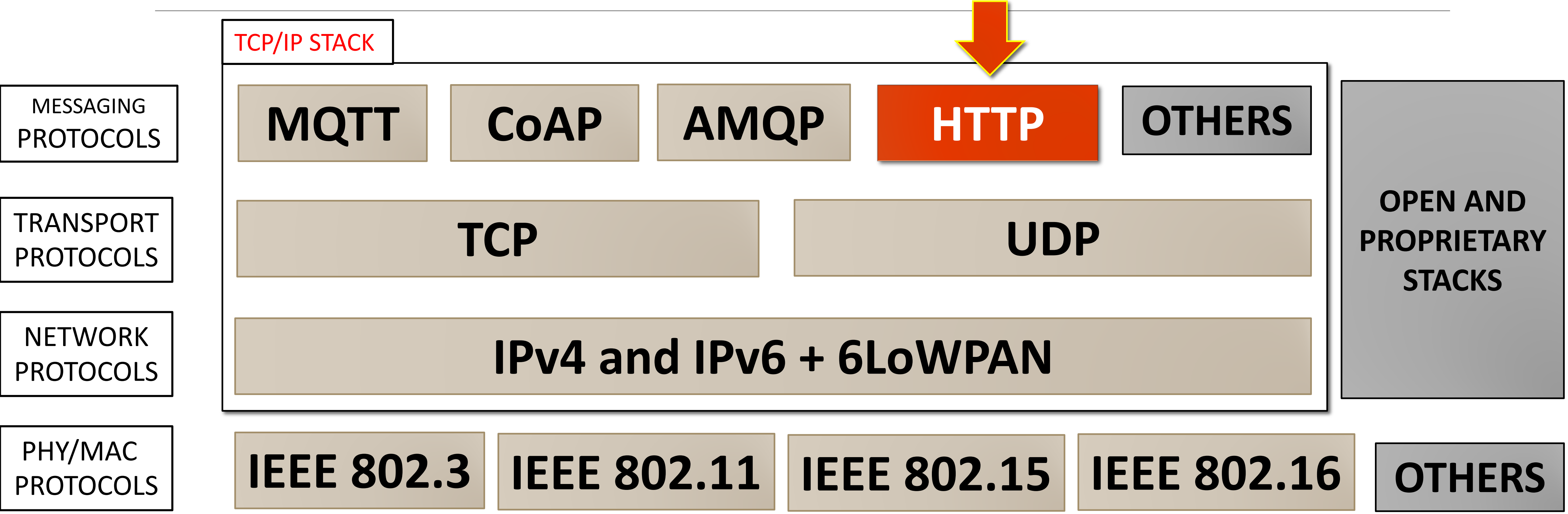
Prof. Marco Di Felice
marco.difelice3@unibo.it

MASTER DEGREE IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, UNIVERSITY OF BOLOGNA, ITALY

IoT Protocol Stack

IoT Protocol Stack



TCP/IP STACK

MESSAGING PROTOCOLS: MQTT, CoAP, AMQP, HTTP, OTHERS

TRANSPORT PROTOCOLS: TCP, UDP

NETWORK PROTOCOLS: IPv4 and IPv6 + 6LoWPAN

OPEN AND PROPRIETARY STACKS

PHY/MAC PROTOCOLS: IEEE 802.3, IEEE 802.11, IEEE 802.15, IEEE 802.16, OTHERS

The Web of Things (WoT)

❑ **Overview**

❑ Background

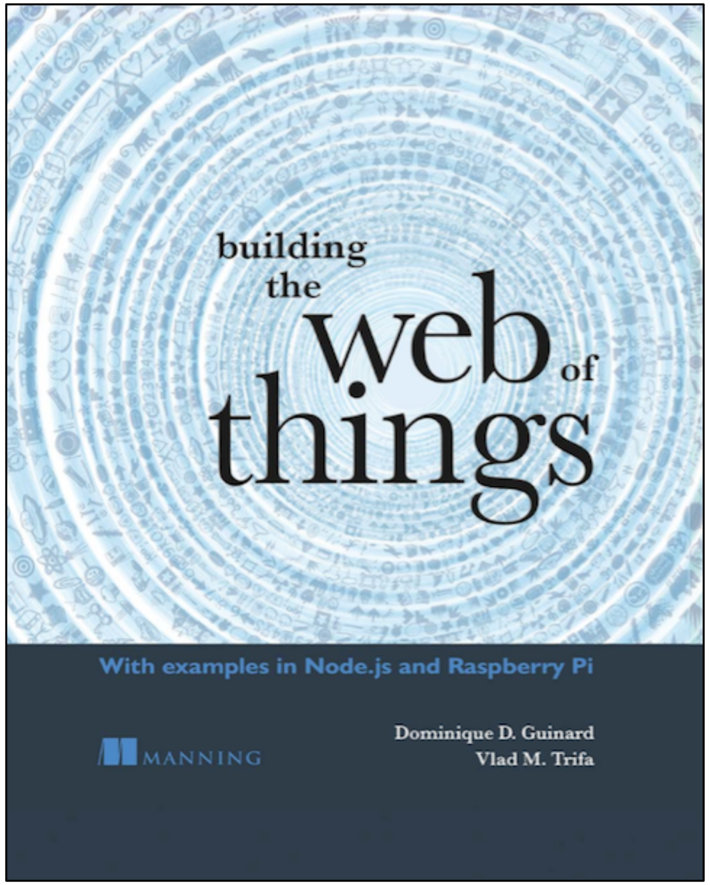
❑ Web Thing: Characteristics

❑ Web Thing: Architectures and Technologies

❑ Findability problem: The Web Thing Model and the semantic Web

❑ Implementing the WoT with Node.js

The Web of Things (WoT)


building the web of things
With examples in Node.js and Raspberry Pi
MANNING
Dominique D. Guinard
Vlad M. Trifa

Building the Web of Things
D. D. Guinard and V. M. Trifa
MANNING Editions, 2016

https://webofthings.org/book/

The Web of Things (WoT)

❑ IDEA: Use the **World Wide Web** (WWW) ecosystem and infrastructure to build applications for the IoT.

- ✧ Interact with Things via web browsers.
- ✧ Explore the Web of Things as surfing the web.
- ✧ Retrieve, process and display sensor data by using web technologies, like JavaScript, JSON, WebSockets.

- ✧ Novel paradigm, but also complementary to the IoT.
- ✧ The term appeared first in 2007, at present several research groups working on closely related concepts (e.g. The **Physical Web**).

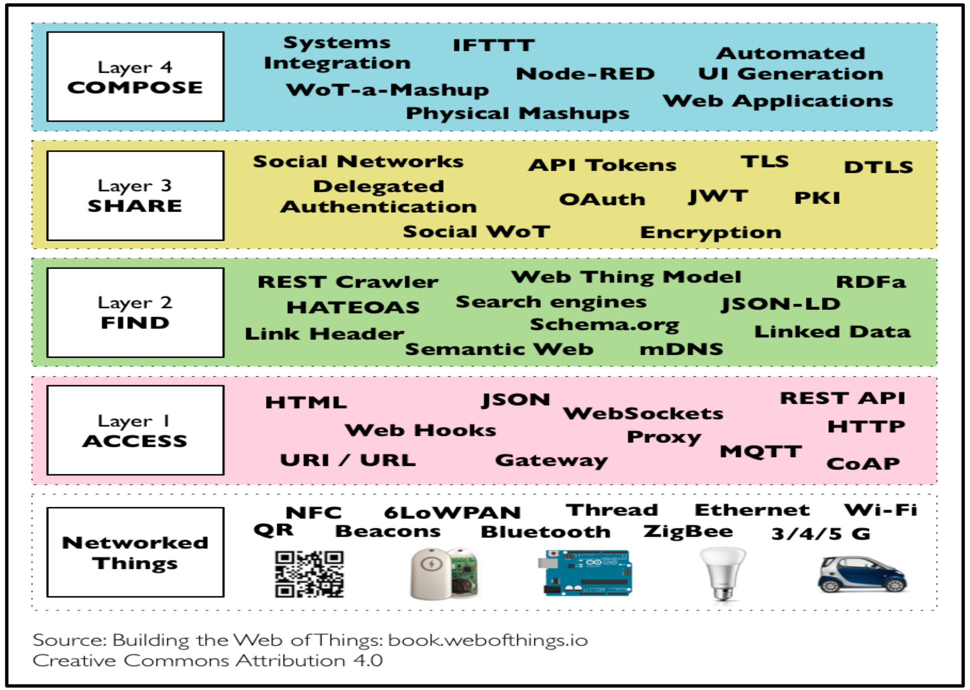
The Web of Things (WoT)

- ❑ **ADVANTAGES** of the WoT
  - ✧ Hide the complexity and variety of lower-layer protocols behind the simple model of the WWW.
  - ✧ Facilitate the integration with all sorts of IoT devices.
  - ✧ Ease the application deployment and maintenance.
  - ✧ Rely on widely used security and privacy mechanisms.

- ❑ **SHORTCOMINGS** of the WoT
  - ✧ WoT devices must support the TCP/IP stack.
  - ✧ Performance on resource-constrained devices.

The Web of Things (WoT)

- ❑ The WoT is implemented on top of the TCP/IP stack (i.e. at the **Application Layer**).



Source: Building the Web of Things: book.webofthings.io
Creative Commons Attribution 4.0

Create mash-up data applications involving multiple Web Things and external Web services.

Share the WoT data in a secure way.

Make Things discoverable and usable by Web apps.

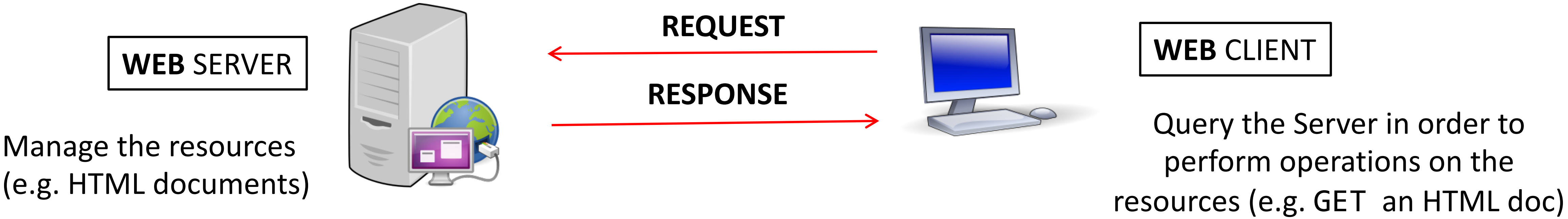
Technologies enabling the connection among Things.

The Web of Things (WoT)

- ❑ Overview
- ❑ **Background**
- ❑ Web Thing: Characteristics
- ❑ Web Thing: Architectures and Technologies
- ❑ Findability problem: The Web Thing Model and the semantic Web
- ❑ Implementing the WoT with Node.js

Reference: The WWW

- ❑ **Internet application, Client-server** architecture


WEB SERVER
Manage the resources (e.g. HTML documents)
REQUEST
RESPONSE
WEB CLIENT
Query the Server in order to perform operations on the resources (e.g. GET an HTML doc)

- ❑ Based on the **HTTP** (**Hypertext Transfer Protocol**) Protocol
  - ✧ Stateless, textual, request-response protocol
  - ✧ Versions: HTTP/1.1, HTTP/1.2, HTTP/2
  - ✧ Limited set of operations: GET, POST, HEAD, PUT, OPTIONS, …

Reference: The REST Principles

- **Representational State Transfer (REST)** → set of architectural principles for distributed systems.

1. Client Server → Interactions based on a request-response communication pattern.
2. Uniform Interfaces →Unambigous standard interface for accessing the resources (e.g. the URI).
3. Stateless → client context and state are not stored on the server.
4. Cacheable → data are cached by clients and intermediaries.
5. Layered System → intermediate components can hide what is behind them (e.g. content delivery networks).

JSON W3C Tutorial: https://www.w3schools.com/js/js_json_intro.asp

Reference: The JSON Language

- ❑ **JavaScript Object Notation → Data description format**
  - ✧ A JSON file is also called a "**Document**".
  - ✧ JSON document can be easily **parsed** by machines.
  - ✧ Single data model, many use-cases.
  - ✧ Favour **system integration and interoperability** among third-party software components.
    - ✓ A JSON document is surrounded by **brackets** { }
    - ✓ Each data entry is a **<key,value>** couple.

```
{ givenname: “mario” }
{ givenname: “mario”, lastname: “rossi”}
```

Reference: The JSON Language

- Value → **Number**, integer or real

```
{ name: “mario”, age: 15, score: 13.45 }
```

- Value→ **String**, surrounded by quotes

```
{ givenname: “mario”, lastname: “rossi” }
```

- Value→ **Boolean**, i.e. true or false

```
{ name: “mario”, employed: true }
```

- Value→ **Array**, surrounded by square brackets

```
{ name: “mario”, codes: [“134”,”042”] }
```

- Value→ **JSON Object** , surrounded by brackets

```
{ name: “mario”, address: {city: “bologna”, nation: “italy” }}
```

Reference: The JSON Language

JSON Document

```
 { givenname: “Mario”,
   lastname: “Rossi”,
   age: 45,
   employed: true,
   salary: 1200.00,
   phones: [“0243434”, “064334343”],
   office: [
   {name: “A”, street: Zamboni, number: 7},
   {name: “B”, street: Irnerio, number: 49}]
}
```

The Web of Things (WoT)

- ❑ Overview
- ❑ Background
- ❑ **Web Thing: Characteristics**
- ❑ Web Thing: Architectures and Technologies
- ❑ Findability problem: The Web Thing Model and the semantic Web
- ❑ Implementing the WoT with Node.js

Web Thing: Characteristics

❑ **Uniform interface →** Things must follow the same rules of the web RESTful components, i.e.:.

- ✧ Addressable resources. Every resource must have a unique identifier and should be addressable using a unique mechanism.
- ✧ Representation of resources. Servers can manage multiple representation of the resources; clients can query for a specific representation of the available resources.
- ✧ Self-descriptive messages. Clients must use and implement only those methods provided by the HTTP protocol.
- ✧ Hypermedia as the engine of the application state (HATEOS)

Web Thing: Characteristics

- Every device on the Web of Things must have a **root URL** corresponding to its network address.

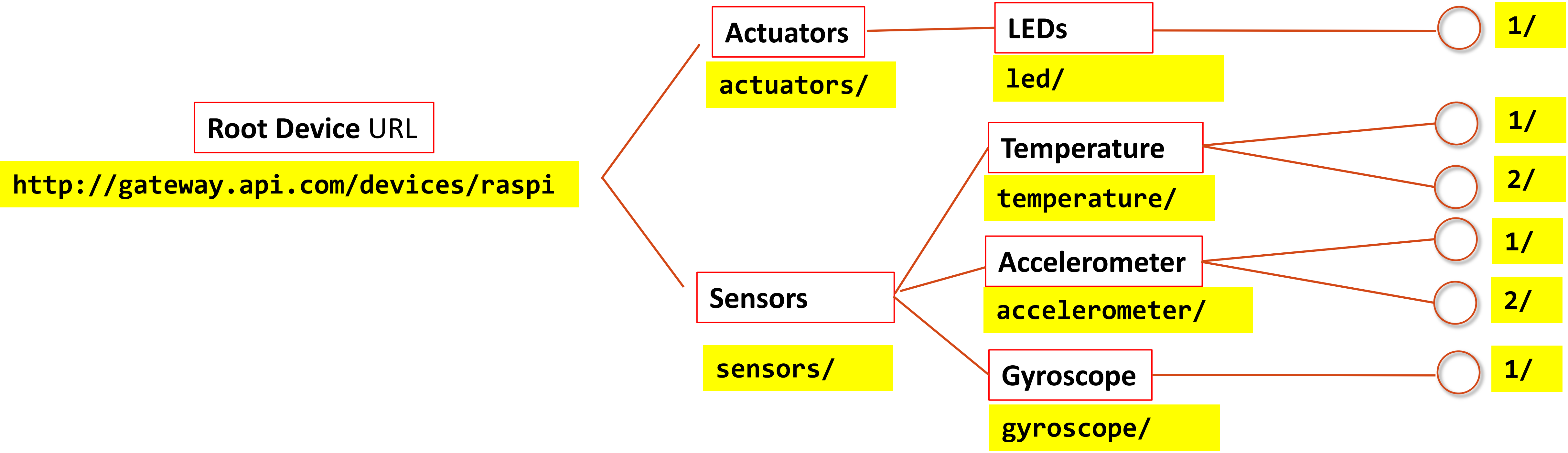
```
<scheme>”:”<authority><path> [“?” query ] [  “#” fragment]
```

```
http://gateway.api.com/devices/TV
https://kitchen:3000/fridge/
http://192.168.1.23/buildings/devices/raspberryPI
```

  - Web Things must be an HTTP server.
  - Web Things should use secure HTTP connections (HTTPS).
  - Web Things must expose their properties using a **hierarchical structure.**

Web Thing: Characteristics

- ❑ Resources on the WoT can be organized in a **hierarchy** defined by a **URL path** (talk more later).



Web Thing: Characteristics

- ❑ **Uniform interface →** Things follow the same rules of the web RESTful components, i.e.:.
  - ✧ Addressable resources. Every resource must have a unique identifier and should be addressable using a unique mechanism.
  - ✧ Representation of resources. Servers can manage multiple representation of the resources; clients can query for a specific representation of the available resources.
  - ✧ Self-descriptive messages. Clients must use and implement only those methods provided by the HTTP protocol.
  - ✧ Hypermedia as the engine of the application state (HATEOS)

Web Thing: Characteristics

- ❑ A Web Thing can support **multiple representations** (=multiple data formats) of its resources.
- ❑ Client can request a preferred representation through the **HTTP content negotiation mechanism.**

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

HTTP REQUEST HEADER

```
200 OK
Content-Type: application/json
…
```

HTTP RESPONSE HEADER

Web Thing: Characteristics

- ❑ **Uniform interface →** Things follow the same rules of the web RESTful components, i.e.:.
  - ✧ Addressable resources. Every resource must have a unique identifier and should be addressable using a unique mechanism.
  - ✧ Representation of resources. Servers can manage multiple representation of the resources; clients can query for a specific representation of the available resources.
  - ✧ Self-descriptive messages. Clients must use and implement only those methods provided by the HTTP protocol.
  - ✧ Hypermedia as the engine of the application state (HATEOS)

Web Thing: Characteristics

- A Web Thing can provide **basic HTTP-based operations** on its resources: `GET`, `POST`, `PUT`, `DELETE`.

GET operation

- ✧ Read the value of a resource.
- ✧ **Safe** and **idempotent** operation.

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

HTTP REQUEST

```
200 OK HTTP/1.1
Content-Type: application/json
{“temperature”:”25”}
```

HTTP RESPONSE

Web Thing: Characteristics

❑ A Web Thing can provide **basic HTTP-based operations** on its resources: GET, POST, PUT, DELETE.

POST operation

- Create a new instance of something that doesn't have its own URL.
- **Unsafe** and **non-idempotent** operation.

```
POST /pi/display/messages HTTP/1.1
Host: devices.webofthings.io
Content-Type: application/json
{“Content”:”Hello world”, “duration”:10}
```

HTTP REQUEST

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

HTTP RESPONSE

Web Thing: Characteristics

- ❑ A Web Thing can provide **basic HTTP-based operations** on its resources: `GET`, `POST`, **`PUT`**, `DELETE`.

PUT operation

- ✧ Update something that already exists and has already its own URL.
- ✧ **Unsafe** and **idempotent** operation.

```
PUT /pi/leds/4 HTTP/1.1
Host: devices.webofthings.io
Content-Type: application/json
{“red”:0, “green”:123”, “blue”: 123}
```

HTTP REQUEST

```
200 OK HTTP/1.1
```

HTTP RESPONSE

Web Thing: Characteristics

❑ A Web Thing can provide **basic HTTP-based operations** on its resources: `GET`, `POST`, `PUT`, **`DELETE`**.

`DELETE` operation

- ✧ Permanently remove a resource from a Thing.
- ✧ **Unsafe** and **idempotent** operation.

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

HTTP REQUEST

```
200 OK HTTP/1.1
```

HTTP RESPONSE

Web Thing: Characteristics

❑ A Web Thing can provide **basic HTTP-based operations** on its resources: GET, POST, PUT, DELETE.

HTTP defines a **list of standard status codes** that must be returned by the server upon reception of a request from the Web client:

- ✧ 200 OK (Successful completition of a request)
- ✧ 201 CREATED (Returned after the creation of a resource)
- ✧ 202 ACCEPTED (Returned by synch operations after request)
- ✧ 401 UNAUTHORIZED (Authorization failed or not issued)
- ✧ 404 NOT FOUND (Resource or document has not been found)
- ✧ 500 INTERNAL SERVER ERROR (Error in processing the request)
- ✧ 501 SERVICE UNAVAILABLE (Server can’t handle the request)

Web Thing: Characteristics

- ❑ **Uniform interface** → Things follow the same rules of the web RESTful components, i.e.:.
  - ✧ Addressable resources. Every resource must have a unique identifier and should be addressable using a unique mechanism.
  - ✧ Representation of resources. Servers can manage multiple representation of the resources; clients can query for a specific representation of the available resources.
  - ✧ Self-descriptive messages. Clients must use and implement only the methods provided by the HTTP protocol.
  - ✧ Hypermedia as the engine of the application state (HATEOS)

Web Thing: Characteristics

- A Web Thing can inform the clients about the list of operations permitted on a specific resource, by using the **OPTIONS** HTTP command.

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

HTTP REQUEST

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

HTTP RESPONSE

[1] Web Thing model, COMPOSE EU project: http://model.webofthings.io

Web Thing: Characteristics

- The WoT model defined in [1] states that each *Web Thing* **MUST** meet these requirements:

1) A Web Thing MUST at least be an **HTTP/1.1 server**.
2) A Web Thing MUST have a root resource accessible via an HTTP URL.
3) A Web Thing MUST support GET, PUT, POST, and DELETE HTTP commands.
4) A Web Thing MUST implement HTTP status codes: 200, 400 and 500.
5) A Web Thing MUST support **JSON** as default representation.
6) A Web Thing MUST support GET on its root URL.

[1] Web Thing model, COMPOSE EU project: http://model.webofthings.io

Web Thing: Characteristics

- ❑ The WoT model defined in [1] states that each *Web Thing* **SHOULD** meet these requirements:

1) A Web Thing SHOULD use secure **HTTP** connections (HTTPS).
2) A Web Thing SHOULD implement the `WebSocket` Protocol.
3) A Web Thing SHOULD support the Web Things model (see later).
4) A Web Thing SHOULD return a 204 code for all write operations.
5) A Web Thing SHOULD provide a default **human-readable documentation**.

[1] Web Thing model, COMPOSE EU project: http://model.webofthings.io

Web Thing: Characteristics

❑ The WoT model defined in [1] states that each *Web Thing* **MAY** meet these requirements:

1) A Web Thing MAY support the HTTP `OPTIONS` verb on its resource.
2) A Web Thing MAY offer a HTML-based user interface.
3) A Web Thing MAY provide additional data about the intended meaning of individual components of its model (e.g. through semantic Web)

The Web of Things (WoT)

- ❑ Overview
- ❑ Background
- ❑ Web Thing: Characteristics
- ❑ **Web Thing: Architectures and Technologies**
- ❑ Findability problem: The Web Thing Model and the semantic Web
- ❑ Implementing the WoT with Node.js

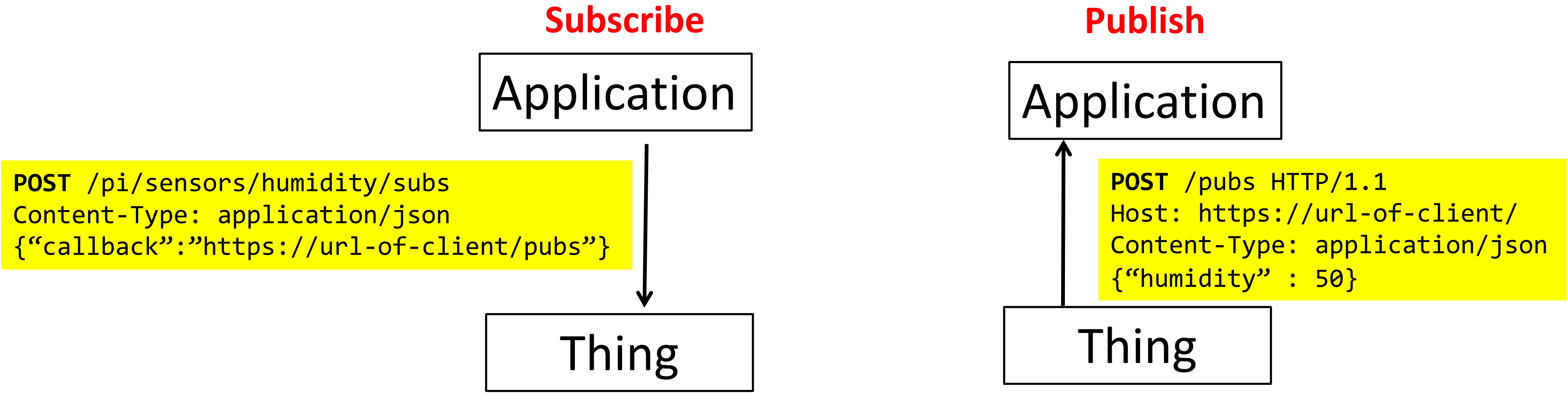
Web Thing: Enabling Technologies

- PROBLEM. HTTP implements a **request-response** communication pattern. What about push-based IoT applications?

  - ✧ Use **polling** mechanism
  - ✧ Use **Webhooks/HTTP callbacks**
  - ✧ Use **long-polling** mechanism
  - ✧ Use **WebSockets**

Web Thing: Enabling Technologies

- ❑ **WebHooks** → The Web Thing and the Web client will act as HTTP clients and also as HTTP servers.



Subscribe

```
POST /pi/sensors/humidity/subs
Content-Type: application/json
{“callback”:”https://url-of-client/pubs”}
```

Publish

```
POST /pubs HTTP/1.1
Host: https://url-of-client/
Content-Type: application/json
{“humidity” : 50}
```

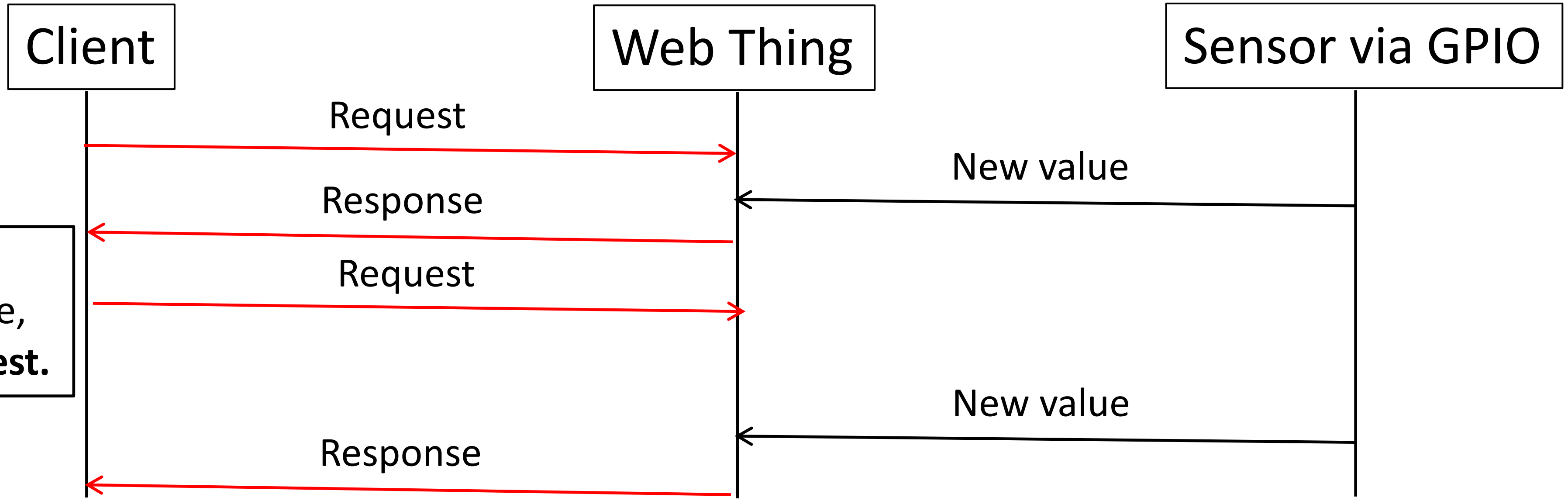
Web Thing: Enabling Technologies

- PROBLEM. HTTP implements a **request-response** communication pattern. What about push-based IoT applications?

  - Use **polling** mechanism
  - Use **Webhooks/HTTP callbacks**
  - Use **long-polling** mechanism
  - Use **WebSockets**

Web Thing: Enabling Technologies

- ❑ **Long Polling→** A client sends the HTTP request to the server; the server holds the request till a new value of the resource is available, then it sends a response.



As soon as the Client receives the response, **it issues a new request.**

Web Thing: Enabling Technologies

- ❑ PROBLEM. HTTP implements a **request-response** communication pattern. What about push-based IoT applications?

  - ✧ Use **polling** mechanism
  - ✧ Use **Webhooks/HTTP callbacks**
  - ✧ Use **long-polling** mechanism
  - ✧ Use **WebSockets**

WebSocket Protocol Specification: https://tools.ietf.org/html/rfc6455

Web Thing: Enabling Technologies

❑ **WebSockets** enable **full-duplex** (bidirectional) communication over a single TCP connection.

- ✧ Part of the HTML 5 specification
- ✧ Novel protocol, alternative to the HTTP
- ✧ Much shorter header (2 bytes) than HTTP

WEBSOCKETS **PROTOCOL HANDSHAKE**

1. Client sends an HTTP request to the server, asking for an **upgrade** to WebSockets.
2. The server replies with Code `101 Switching Protocols` if it supports WebSockets
3. A bidirectional TCP socket is open and used for the data transfer.
4. The TCP socket is long-living, i.e. termined only when Client or Server transmit a **Close** frame.

WebSocket Protocol Specification: https://tools.ietf.org/html/rfc6455

Web Thing: Enabling Technologies

- ❑ **WebSockets** enable **full-duplex** (bidirectional) communication over a single TCP connection.

WEBSOCKETS **PROTOCOL HANDSHAKE**

```
GET /pi/sensors/humidity/ HTTP/1.1
Host: devices.webofthings.io
Upgrade: websocket
Connection: Upgrade
```

REQUEST for a WEBSOCKETS UPGRADE

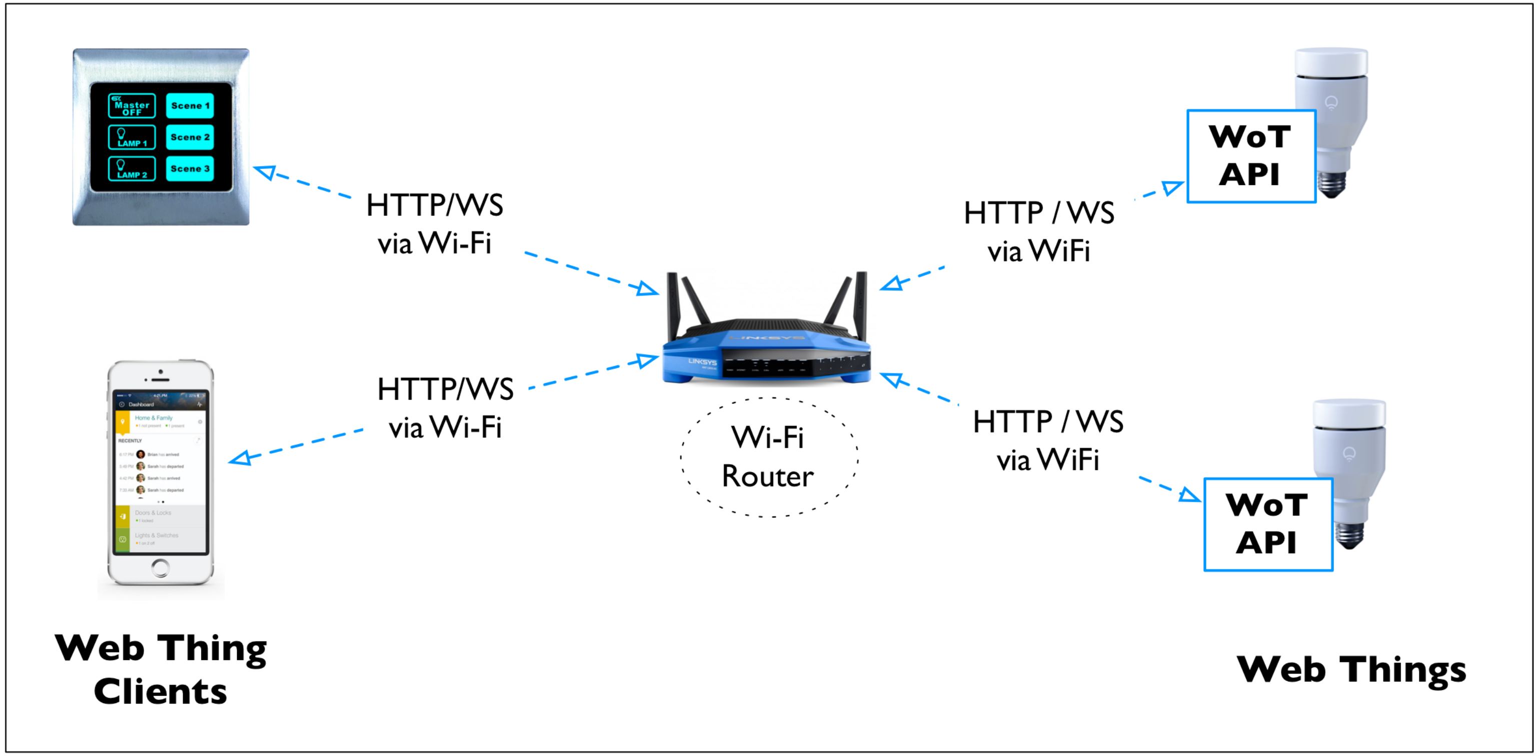
```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
[ ]
Upgrade: websocket
Access-Control-Allow-Origin: http://localhost:63342
```

ACK of a WEBSOCKETS UPGRADE

WebSocket Protocol Specification: https://tools.ietf.org/html/rfc6455

Web Thing: Architectures

✧WoT scenario: **Direct Connectivity**

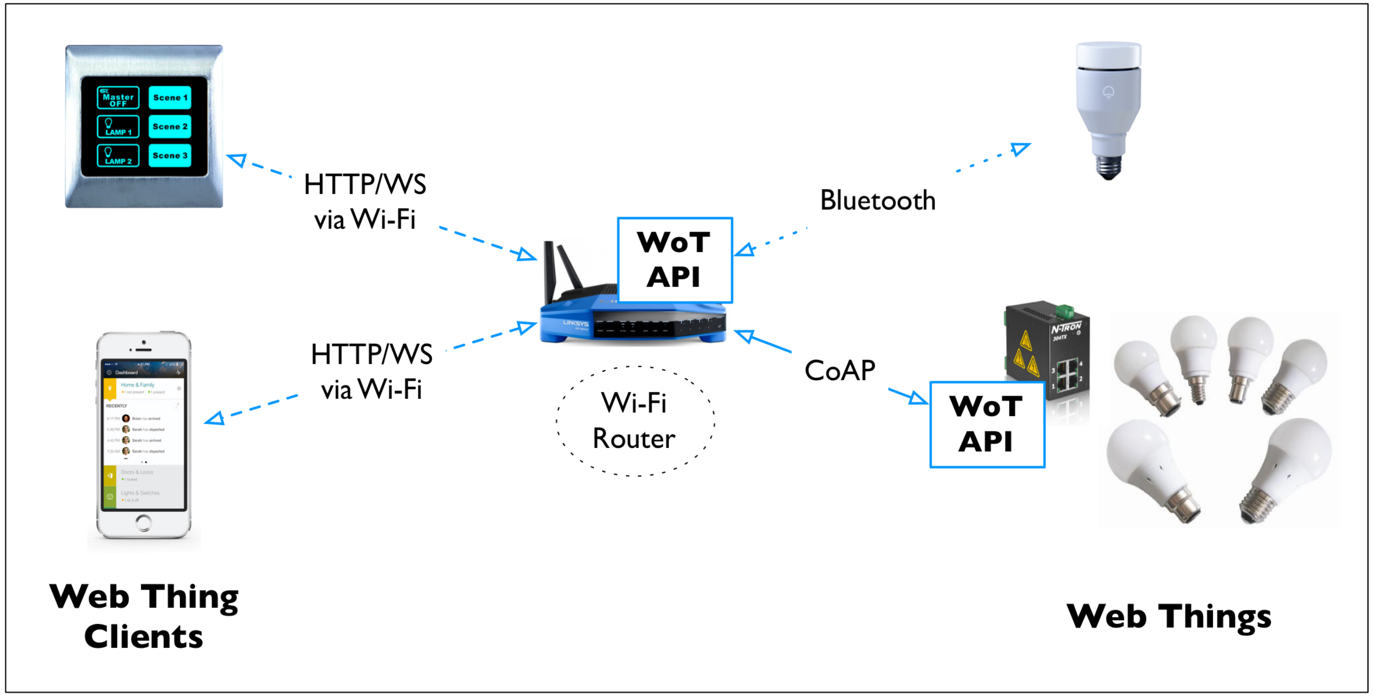


- ✧ Web Clients and Web Things can belong to the the same network or to different networks.
- ✧ Each Web Thing implements an HTTP server and the WoT API.
- ✧ The Router is not a Web Thing Object.

WebSocket Protocol Specification: https://tools.ietf.org/html/rfc6455

Web Thing: Architectures

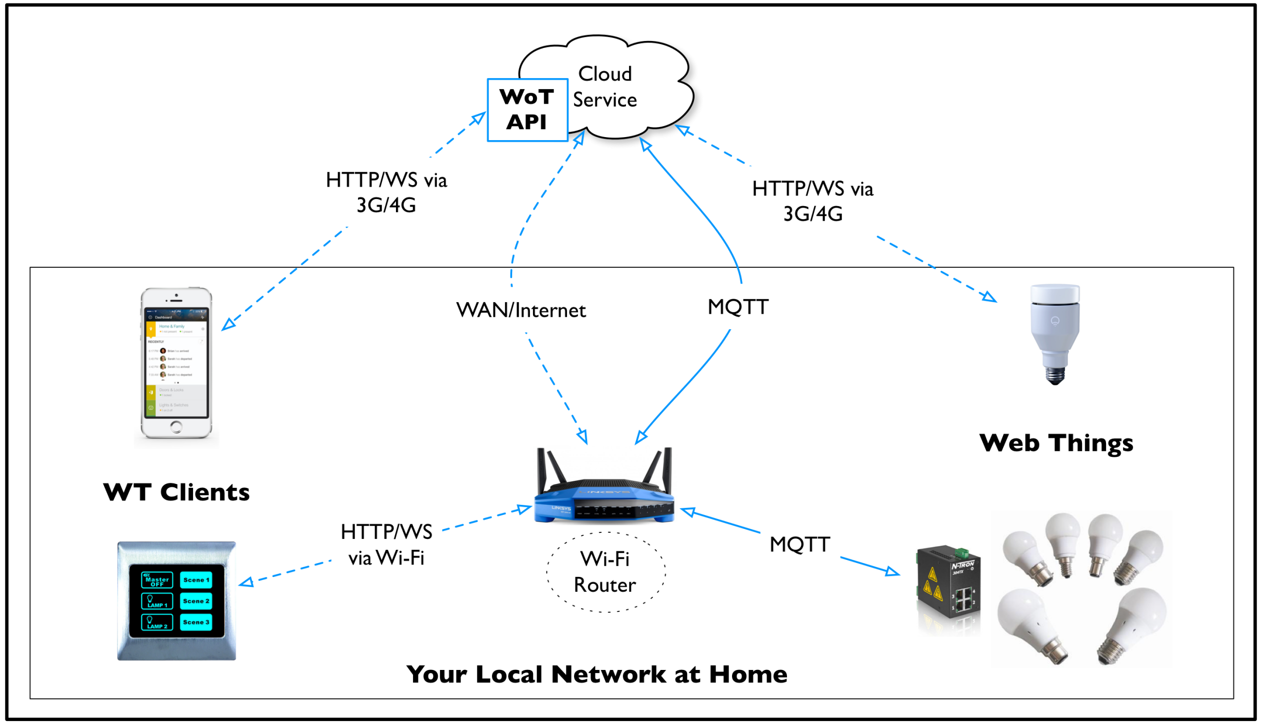
✧WoT Scenario: **Gateway-based Connectivity**



- ✧ Not all the Things are able to implement the WoT API and to support the WebSockets.
- ✧ The Gateway is a Web Thing Object, and **works as proxy** for the other Things.

Web Thing: Architectures

✧WoT Scenario: **Cloud-based Connectivity**



- ✧ As in the previous scenario, not all the Things are able to implement the WoT API and to support the WebSockets.
- ✧ Differently from the previous scenario, **the gateway/proxy is a cloud service** and not another device located within the same network.

The Web of Things (WoT)

- ❑ Overview
- ❑ Background
- ❑ Web Thing: Characteristics
- ❑ Web Thing: Architectures and Technologies
- ❑ **Findability problem: The Web Thing Model and the semantic Web**
- ❑ Implementing the WoT with Node.js

WoT Findability Problem

- ❑ **Findability** → capability of easily discover and understand any entity of the Web of Things. Three separate sub-problems:
  - ❑ How to discover Web Things.
  - ❑ How to know what commands to send and how.
  - ❑ How to understand the meaning of data being exchanged with the Web Thing.

WoT Findability Problem

- There are several **discovery protocols** for LANs: mDNS, DLNA, UPnP, Apple Bonjour, …

mDNS message

```
service up: {
    interfaceIndex: 4
    type: {
        name: ‘http’
        protocol: ‘tcp’
        fullyQualified: true }
    fullname; ‘Man\\032MFC-8520DN._http._tcp.local.’
    host: ‘EVT-BW-MAN.local’
    port: 80
    addresses: [ ‘192.168.0.6’ ]
    ….
}
```

✧ Clients listen for new **mDNS messages** and populate the local DNS table.

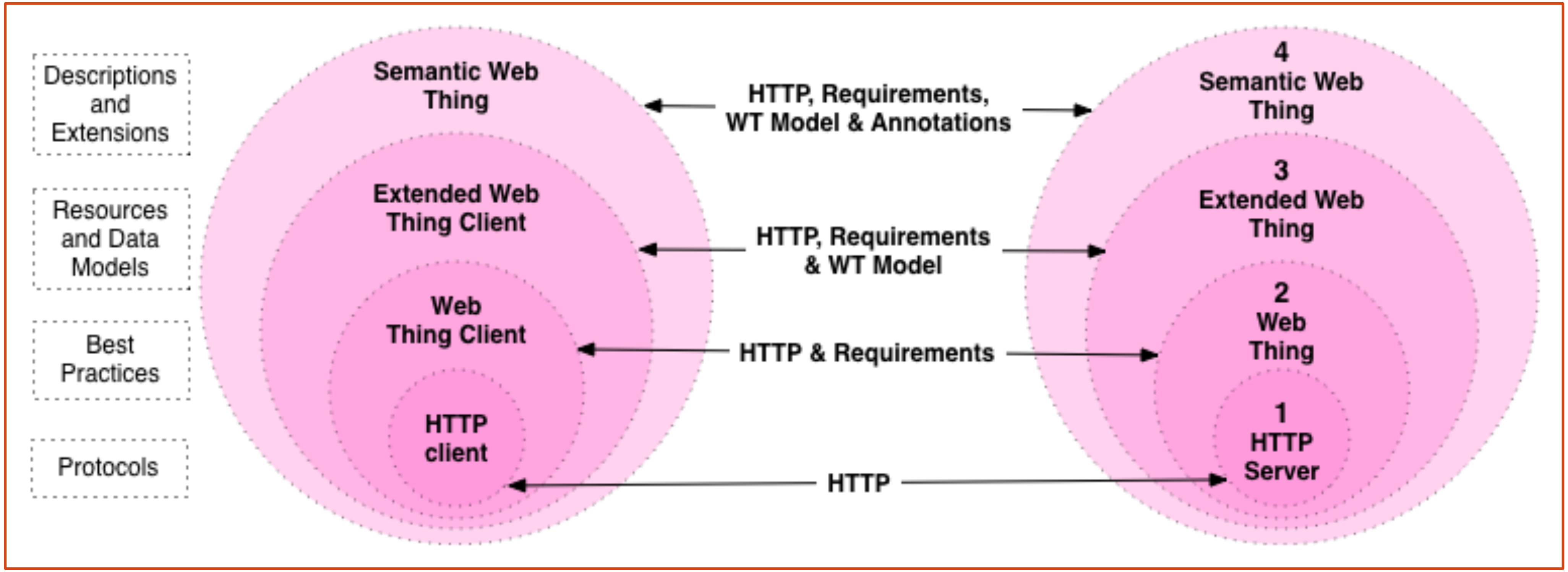
A service of type **HTTP/TCP** has been discovered.

The service is reachable at: **http://evt-bw-man.local**

WoT Findability Problem

- □ **Findability →** capability of easily discover and understand any entity of the Web of Things. Three separate sub-problems:
  - □ How to discover Web Things.
  - □ How to know what commands to send and how.
  - □ How to understand the meaning of data being exchanged with the Web Thing.

WoT Findability Problem



http://model.webofthings.io

WoT Findability Problem

- ❑ **Web Thing Model** → conceptual, uniform description of a Thing and of its capabilities.
  - ✧ **Flexibility**: it should be able to represent all sorts of devices and products, as well as all sorts of interactions.
  - ✧ **Viability**: it should ensure that client applications can interact with new Things automatically (without any human in the loop)
  - ✧ Several approaches proposed, few consolidated solutions.
  - ✧ We follow the model proposed in:

http://model.webofthings.io

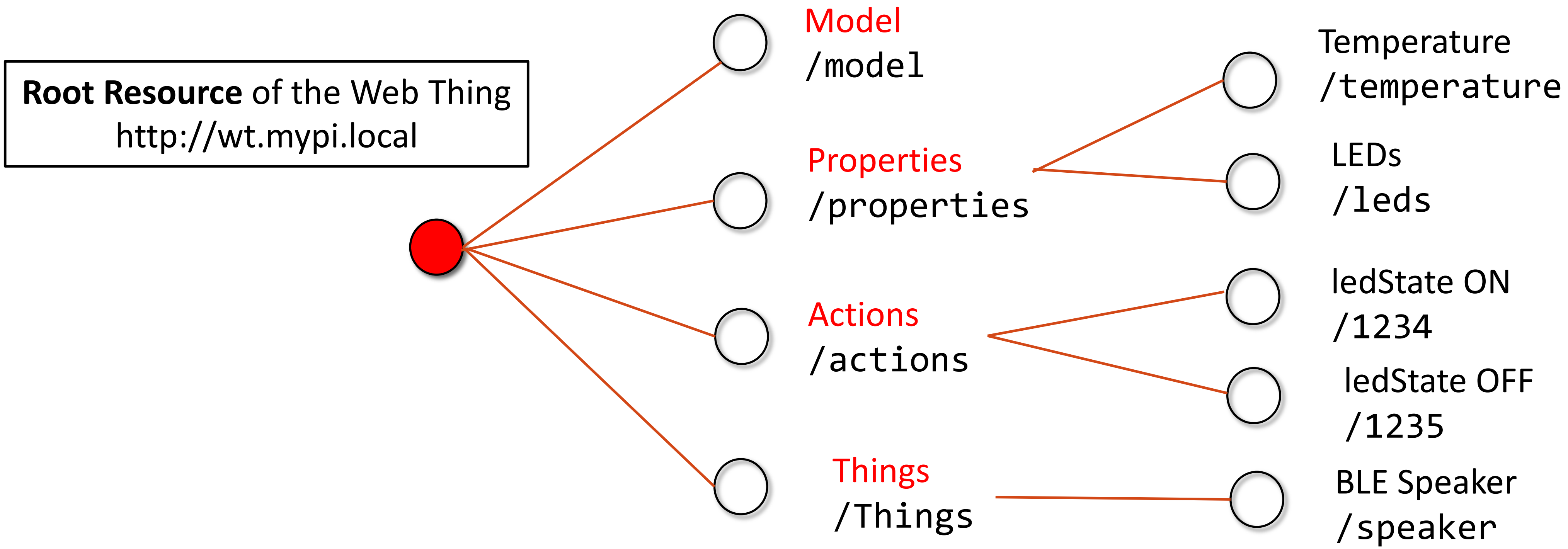
WoT Findability Problem

❑ All the Web of Things have a root URL, and implement a **logical tree structure** for resources:

- ✧ `/model` → metadata such as Thing name, descriptions or configurations (a GET of the model will return its complete description).
- ✧ `/properties` →internal state of a Thing, expressed by a list of <key, value> tuples, where the value can be any JSON value.
- ✧ `/actions` → functions offered by the Thing to clients.
- ✧ `/Things` → list of Things proxied by the current device.
- ✧ `/subscriptions` → list of active subscriptions (in case publish-subscribe paradigm is implemented).

WoT Findability Problem

- All the Web of Things have a root URL, and implement a **logical resource tree structure**:



WoT Findability Problem

- □ In response to a **HTTP GET request** **(on the URL)**, a Web of Thing must return a JSON representation like this:

Field name	Type	Description
`id`	String	Relative URL of the resource
`createdAt`	String	Timestamp when the resource was created
`updatedAt`	String	Timestamp when the resource was last updated
`name`	String	Short human-readable name of a resource
`description`	String	Human-readable description of a resource
`tags`	String	Array of tags
`customFields`	Objects	JSON object with key-value pairs
`links`	Objects	JSON Object with the list of sub-resources

WoT Findability Problem

```
{
  "id": "myCar",
  "name": "My great car",
  "description": "This is such a great car.",
  "createdAt": "2012-08-24T17:29:11.683Z",
  "updatedAt": "2012-08-24T17:29:11.683Z",
  "tags": [
    "cart",
    "device",
    "test"
  ],
  "customFields": {
    "size": "20",
    "color": "blue"
  },
```

```
"links": {
    "model": {
      "link": "model/",
      "title": "Model this Web Thing."
    },
    "properties": {
      "link": "properties/",
      "title": "Properties of this Web Thing."
    },
    "actions": {
      "link": "actions/",
      "title": "Actions of this Web Thing."
    },
    ...
  }
}
```

WoT Findability Problem

- ❑ Each resource may link to different **sub-resources.**
- ❑ Each link is defined by a **relation type** (the "link type"), the actual **URL** of the sub-resource (the "link"), and a human-readable **identifier** for the relation (the "title").
- ❑ Links should be exposed in two ways:
  - ✧ Using the `links:` field of the JSON payload.
  - ✧ Using the HTTP `Link` header field.

WoT Findability Problem

- ❑ The WT Model includes the following **link types**:

Relation type	Description
`model`	A link to the resource description.
`properties`	The properties of this resource.
`actions`	The actions that this resource can perform.
`things`	The Web things proxied by this resource.
`subscriptions`	The endpoint to manage subscriptions to this resource.
`type`	The instance of the resource identified by a target external URL.
`product`	A link to authoritative product information for this Web Thing.
`help`	A link to the online manual page for this Web Thing.
`ui`	A link to the HTML-based user interface for this Web Thing.

WoT Findability Problem

- **EXAMPLES**: Links included in the JSON format:

```
{    ...
          "links":{
            "<relType>":{
              "link": "<String>",
              "title": "<String>"
            }
            "<_customRelType>":{
              "link": "<String>",
              "title": "<String>"
            },
            ...
          }
}
```

```
“links” : {
           "model": {
             "link": "model/",
             "title": "Model of this Web Thing."
           },
           "properties": {
               "link": "properties/",
               "title": "Properties of thisThing."
              },
            "actions": {
                "link": "actions/",
                "title": "Actions of this Web Thing."
              },
   ….
```

WoT Findability Problem

- **EXAMPLES**: Links included in the HTTP Header:

```
 --> REQUEST
              GET /http://wt.mypi.local

<-- RESPONSE
              200 OK
              Link: <model/>; rel="model"
              Link: <properties/>; rel="properties"
              Link: <actions/>; rel="actions"
              Link: <product/>; rel="product"
              Link: <type/>; rel="type"
              Link: <help/>; rel="help"
              Link: <ui/>; rel="ui"
              Link: <_myCustomLinkRelType/>; rel="_myCustomLinkRelType”
                  … Here it follows the JSON representation of the Web Thing….
```

WoT Findability Problem

- **EXAMPLES**: Links included in the HTTP Header:

```
 --> REQUEST
              GET /http://wt.mypi.local

<-- RESPONSE
              200 OK
              Link: <model/>; rel="model"
              Link: <properties/>; rel="properties"
              Link: <actions/>; rel="actions"
              Link: <product/>; rel="product"
              Link: <type/>; rel="type"
              Link: <help/>; rel="help"
              Link: <ui/>; rel="ui"
              Link: <_myCustomLinkRelType/>; rel="myCustomLinkRelType"
                   … Here it follows the JSON representation of the Web Thing….
```

WoT Findability Problem

- ❑ **OPERATION EXAMPLE**: Retrieve properties’ values

```
--> REQUEST
              GET  http://wt.mypi.local/properties
<-- RESPONSE
              200 OK
              Link: <model/>; rel=”model”
              [
                {
                  "id":"temperature",
                  "name":"Kitchen Temperature Sensor",
                  "values":{
                    "temp":22,
                    "timestamp":"2015-06-14T14:30:00.000Z"
                  },
                  … ]
```

WoT Findability Problem

- ❑ **OPERATION EXAMPLE**: Update a property

```
--> REQUEST
              PUT {wt}/properties/temperature/

              [
                {
                  "temp":24
                }
              ]


<-- RESPONSE
              204 NO CONTENT
              Location: {wt}/properties/temperature/
```

WoT Findability Problem

❑ **OPERATION EXAMPLE**: Retrieve the list of actions

```
--> REQUEST
             GET http://wt.mypi.local/properties
<-- RESPONSE
             200 OK
             Link: <http://webofthings.org/actions/upgradefirmware>; rel="type”
             [
               {
                 "id":"upgradeFirmware",
                 "name":"Upgrade Device Firmware"
               },
               {
                 "id":"reboot",
                 "name":"Reboot"
               } ]
```

WoT Findability Problem

❑ **OPERATION EXAMPLE**: Execute an action

```
--> REQUEST
              POST http://wt.mypi.local/actions/reboot

              {
                "delay":50,
                "mode":"debug"
              }

<-- RESPONSE
              204 NO RESPONSE
              Location: {wt}/actions/reboot/233
```

WoT Findability Problem

- ❑ **OPERATION EXAMPLE**: Retrieve the action status

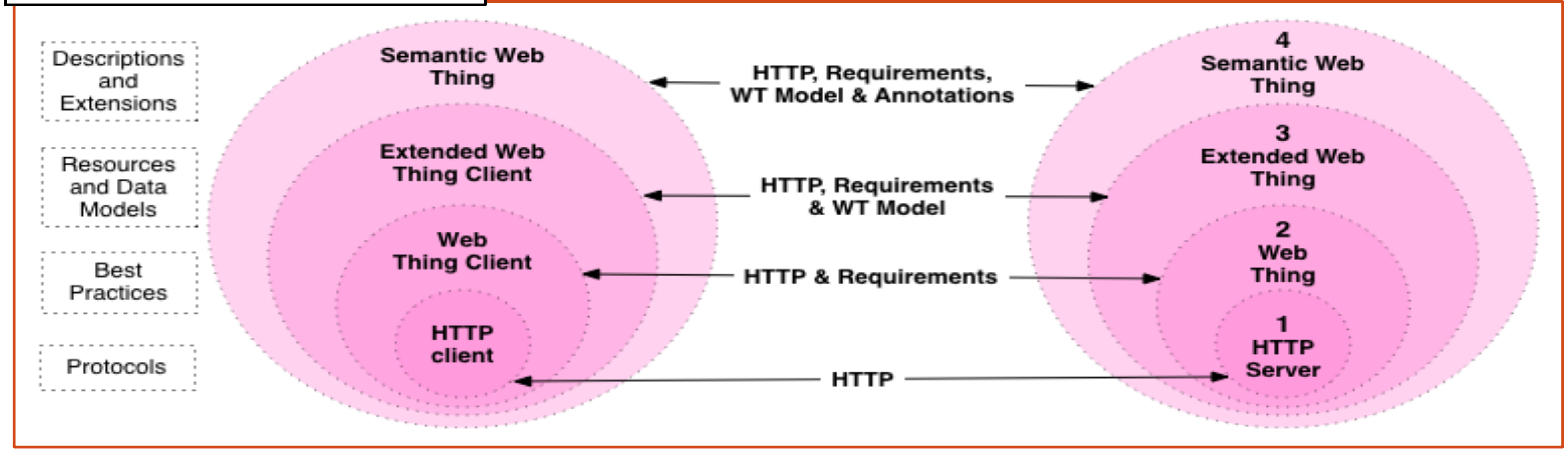
```
--> REQUEST
                     GET http://wt.mypi.local/actions/reboot/233

<-- RESPONSE
                     200 OK
                     {
                       "id":"233",
                       "value":{
                         "delay":50,
                         "mode":"debug"
                       },
                       "status":"executing",
                       "timestamp":"2015-06-14T14:30:00.000Z"
                     }
```

WoT Findability Problem

- ❑ **Findability →** capability of easily discover and understand any entity of the Web of Things. Three separate sub-problems:
  - ❑ How to discover Web Things.
  - ❑ How to know what commands to send and how.
  - ❑ How to understand the meaning of data being exchanged with the Web Thing.

WoT Findability Problem

http://model.webofthings.io



- ✧The Web Thing Model described so far provides the abstraction of an **Extended Web Thing (i.e. Level 3).**
- ✧Clients can discover the way to interact with WebThings … however they cannot infer the **meaning of data**, and relationships among different data entries.

WoT Findability Problem

- ✧ **Semantic Web** refers to a set of techniques to ease the finding, sharing and process of web contents thanks to a common and extendible data description and interchange format.
  - ✧ Meaning is associated with data entities by annotating the meta-data based on a shared **Vocabulary**.
  - ✧ Vocabulary elements can also have **relationships** with each other.
  - ✧ A **reasoner** can be used to **infer** additional properties or relationships.

JSON-LD 1.0 W3C Specifications: http://www.w3.org/TR/json-ld/

WoT Findability Problem

✧ JavaScript Object Notation for Linked Data (JSON-LD)

- ✧ **Lightweight syntax** to serialize Linked Data in JSON.
- ✧ 100% compatible with the **JSON language**.
- ✧ In addition, it introduces semantic features such as:
  - ❑ A universal **identifier mechanism** for JSON objects via the use of IRIs.
  - ❑ A mechanism in which a value in a **JSON object may refer to a JSON object** on a different site on the Web.
  - ❑ The ability to **annotate** strings with their language.
  - ❑ A way to associate **datatypes with values** such as dates and times.

WoT Findability Problem

✧ JavaScript Object Notation for Linked Data (JSON-LD)

Keyword	Description	Example
@context	Used to define the vocabolary used throughout a JSON-LD document	
@id	Used to uniquely identify things with IRI	
@type	Used to set the data type of a node	
@language	Used to specify the language for a particular string value	

The complete description of JSON-LD syntax tokens and keywords can be found at:
`http://www.w3.org/TR/json-ld/`

The Web of Things (WoT)

❑ Overview

❑ Background

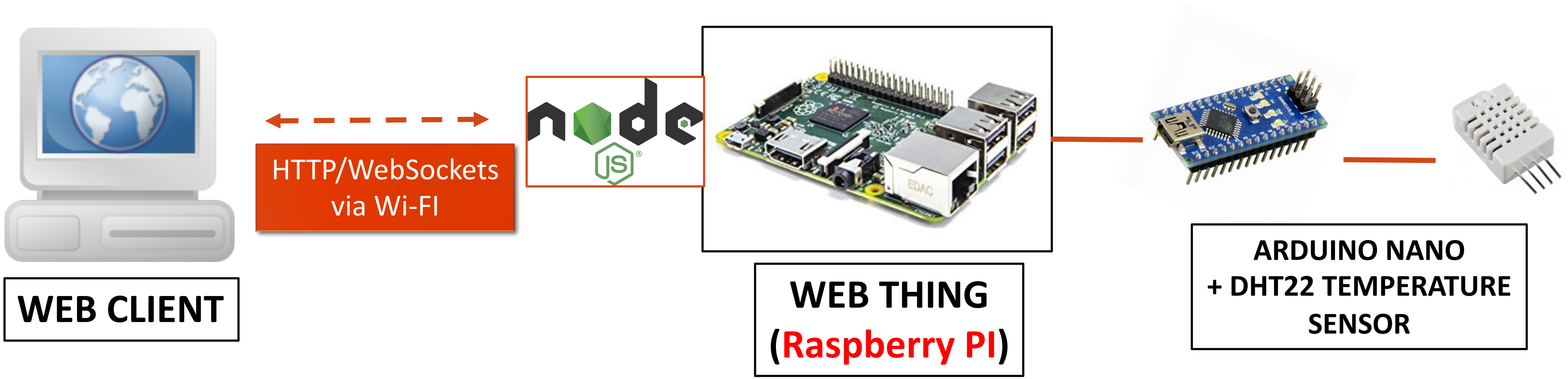
❑ Web Thing: Characteristics

❑ Web Thing: Architectures and Technologies

❑ Findability problem: The Web Thing Model and the semantic Web

❑ **Implementing the WoT with Node.js**

The Web of Things (WoT)



HTTP/WebSockets via Wi-FI

WEB CLIENT

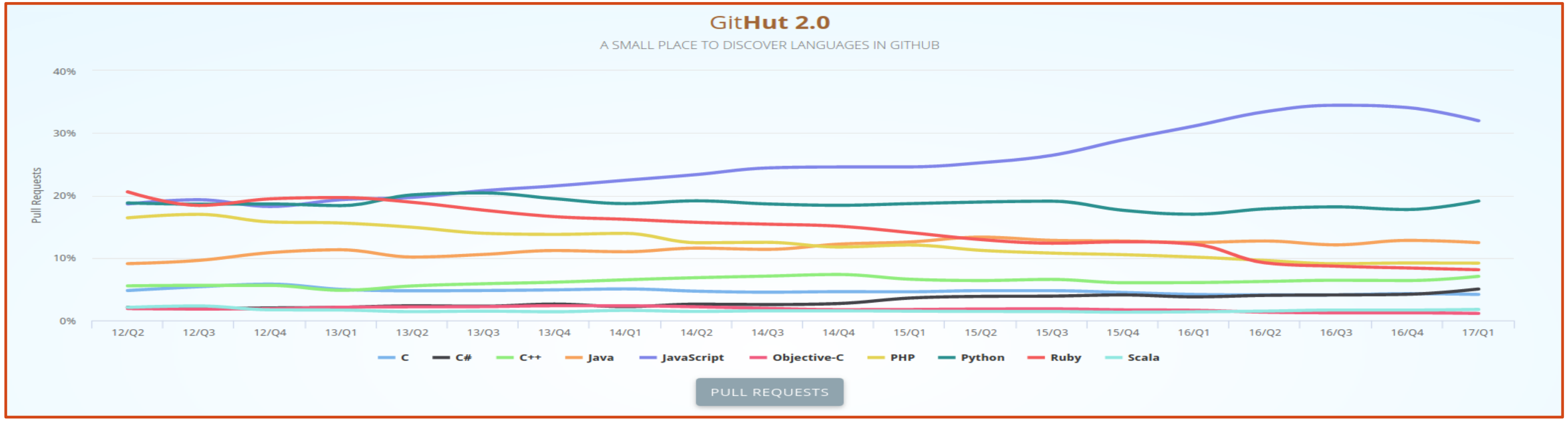
WEB THING (Raspberry PI)

ARDUINO NANO + DHT22 TEMPERATURE SENSOR

Reference: Node.js Framework

- ❑ **Javascript** is the most popular programming language, according to the number of public repositories in GitHub.

https://github.com/madnight/githut



Reference: Node.js Framework

- ❑ **Node.js** Framework
  - ✧ Open source server framework for deploying high-performance server-side applications.
  - ✧ Node.js applications are deployed in **Javascript**.
  - ✧ Single-threaded, non-blocking web servers.
  - ✧ Asynchronous programming.
  - ✧ Highly modular, based on the npm packet manager.

Reference: Node.js Framework

❑ **Node.js** Framework

first.js

```
var http = require('http');

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end('Hello World!');
}).listen(8080);
console.log(‘Web Server started on port 8080);
```

Create a server and pass it a function to be called whenever a new client sends an **HTTP request.**

The server **listens** on port 8080

```
/home/raspi/$node first.js
Web Server started on port 8080
```

Starting the **server** via command-line

Reference: Node.js Framework

❑ **Node.js** Framework

first.js

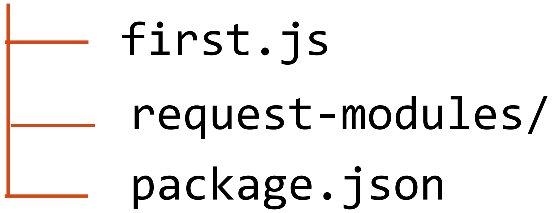
```
var http = require(’request’);

http.createServer(function (req, res) {
   …
```

Include an external **Module** named **request.**

```
/home/raspi/$npm install request--save
```

Install additional Packages through the **npm** tool

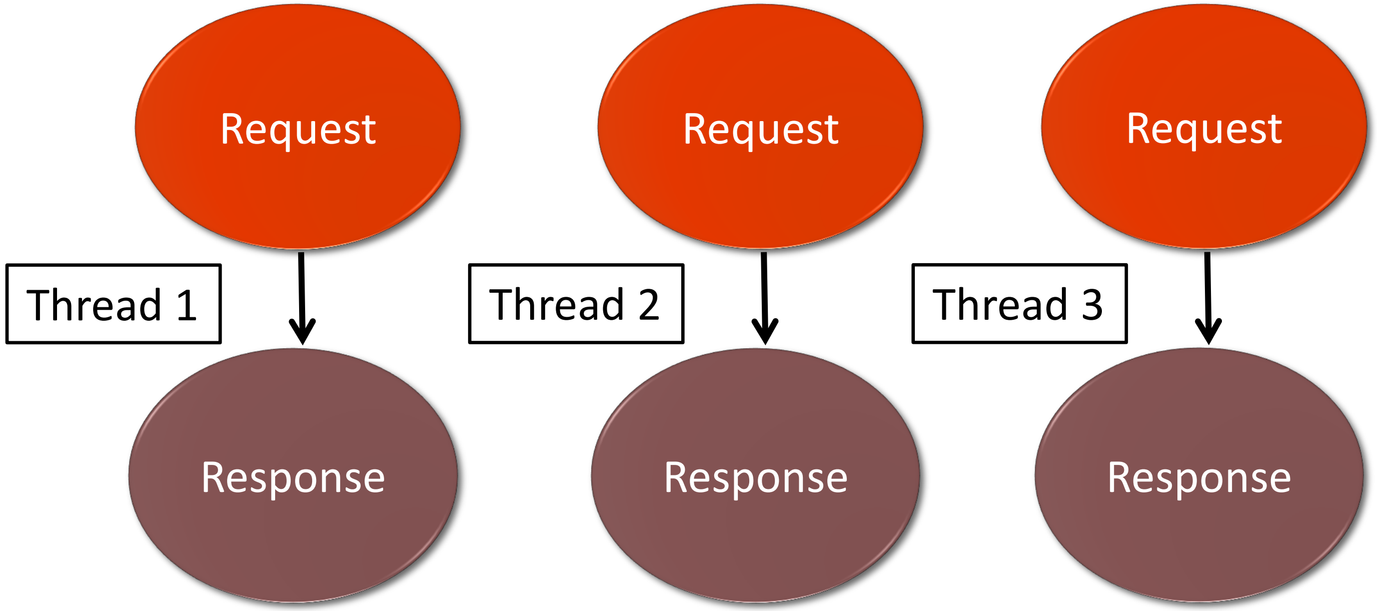
APPLICATION STRUCTURE



```
├── first.js
├── request-modules/
└── package.json
```

Reference: Node.js Framework

- ❑ **Synchronous Server-side** Programming (e.g. PHP)

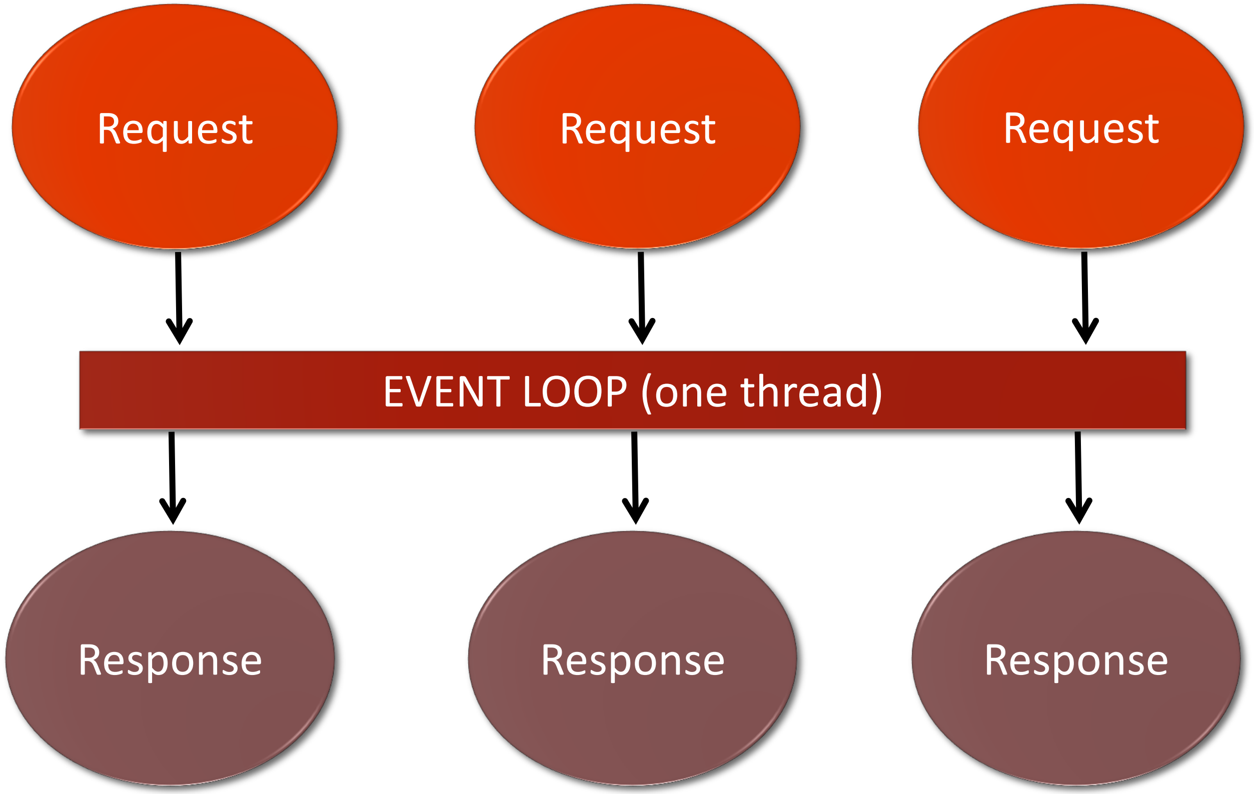


```
var result=database.query(“SELECT
                Things FROM
                DeviceTable”);
console.log(result);
```

- ✧ Script works in a **sequential** manner.
- ✧ Script execution is blocked till current I/O operations are completed.
- ✧ (IN THE CODE ABOVE) The message is written on the console AFTER the database query has been completed.

Reference: Node.js Framework

❑ **Asynchronous Server-side** Programming



```
database.query("SELECT
               Things FROM
              DeviceTable",
          function result {
         //do something with results
} };
console.log(result);
```

- ✧ Asynchronous I/O operations.
- ✧ Anonymous callbacks are executed once a request has been completed.
- ✧ (IN THE CODE ABOVE) The message might be written on the console before the query has been completed.

Reference: Node.js Framework

❑ **Learn more about Node.js programming**

Official Page: `https://nodejs.org/en/`

W3C Tutorial: `https://www.w3schools.com/nodejs/`