Web of Things Standard Recommendation

“WoT Label”

Version 0.1 - 27. May 2015

Authors:
Vlad Trifa, EVRYTHNG
Dominique Guinard, EVRYTHNG

Contributors:
David Carrera, Barcelona Supercomputing Center
François Daoust, W3C

Based on work done within the Compose project and within the Webofthings.org initiative.

NOTE: This is a work in progress document, nothing here is finalized and we are editing this document as you read it. Feedback is very welcome by means of comments that you can add inline!

Introduction

This document aims to standardize the Web of Things by defining a set of specific implementation and resource design guidelines to be followed by anyone wanting to create a product, device, service, or application for the Web of Things.

The following guidelines and recommendations will ensure interoperability across all entities on the Web of Things. Unlike other custom (non-Web) protocols used for M2M that cause a “parallel universe” to the existing Web; the Web of Things is designed to be seamlessly integrated to the existing Web so it can fully leverage its infrastructure and standards to minimize integrations across applications and systems.

Ultimately the goal is for Things to be accessible via HTTP, extended with WebSockets (WS) to support Pub/Sub. This ensures that Web applications, including those purely browser-based, can access Web Things. However, but the patterns, resources and payloads definitions are useable for any system based on REST (e.g., CoAP) even if not using HTTP. The section “Integration Patterns” discusses the integration of non-HTTP/WS devices to the Web of Things.

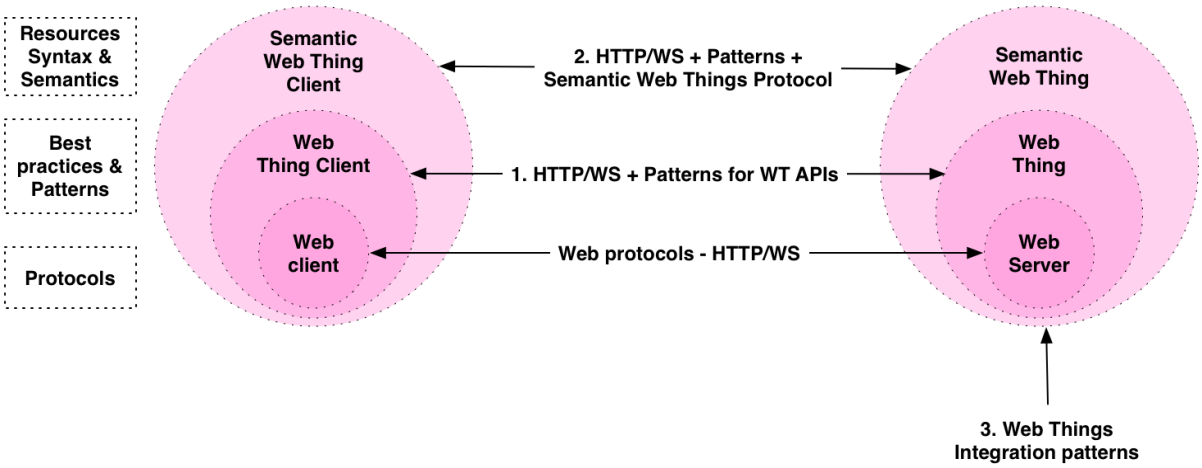


Figure 1: The three pillars of the Web of Things standard recommendation: 1. Patterns for Web of Things APIs, 2. Semantic Web Things Protocol, 3. Web Things Integration Patterns

As such, this recommendation is composed of 3 parts as shown in Figure 1.

1. **Patterns for Web Things (WT) APIs**. This part doesn't introduce any specific data naming conventions of domain-specific models. It is solely composed of multiple constraints and recommendations on how the HTTP protocol should be implemented so that any entity on the Web of Things can easily interact with each other. All WTs MUST implement all the MUST constraints of these requirements and if technically possible the SHOULD constraints as well.
2. **Semantic Web Things Protocol**. Once a WT follows the conventions defined above, it is then capable to read and exchange data with any other entities of the Web of Things. However, it doesn't mean it can "understand" what the object is, what data or services it offers, etc. Therefore, the second component of this specification proposes an HTTP-based protocol with a set of resources, data models, payload syntax and semantic extensions that WT and applications should follow. Implementing the Semantic Web Things protocol as well will ensure not only that your device can be easily used by other HTTP clients, but especially be findable and automatically usable by other WoT applications.

What are Web Things?

understand what it is and what properties it has, what commands are supported by the device, and finally send commands to it that the device can understand and execute.

Entities and Actors

- ***Client*** is a general term for any physical or digital entity that can interact with a Web thing. Can be a mobile app, a client, a phone, a user, or another Web Thing.
- ***Web Things*** (WT). A Web Thing is a physical object or place in the real-world that offer RESTful APIs.
  Examples of Web Things are: an Arduino board, a garage door, a bottle of soda, a building, a TV, etc. It could also be an intermediate in the network such as a Gateway or a Cloud service.
- ***Actions*** are requests that an external entity can send to a WT, which may or may not change the internal state of the WT. This is used mainly to send commands or notifications to a device. Examples of Actions are “open” or “close” for a garage door, “enable” or “disable” for a smoke alarm. “Checkin” or “scan” for a bottle of soda.
  The direction of an Action is Client -> Web
- ***Events*** are notifications generated by Web Things. The direction of an Event is Web Thing -> Web.
- ***Properties*** are values reflecting the current (and possibly historical) internal state of a Web Thing.

Actions and Events are structured messages, that are exchanged between entities in the Web of Things.

Actions are specific commands sent TO a device by an external entity
Events are specific notifications sent FROM a device to an external entity

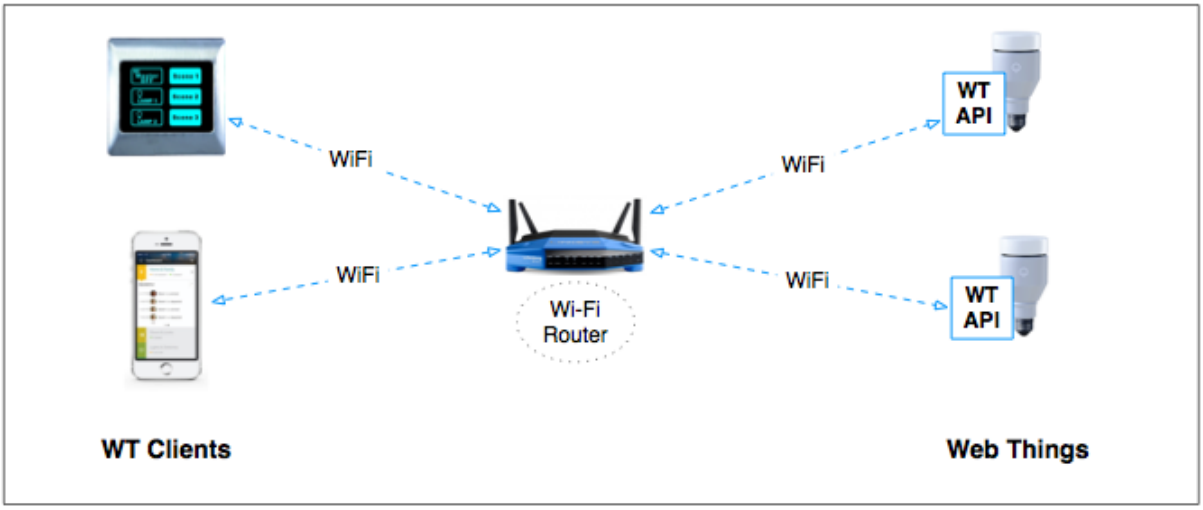
Properties are raw semi-structured, typed streams of data generated by Web Things.

Integration Patterns

Some physical entities might not expose a Web API themselves for various reasons (e.g. a ZigBee sensor node, or an Alarm clock accessible over HomeKit only), in which case they aren’t WTs as such.

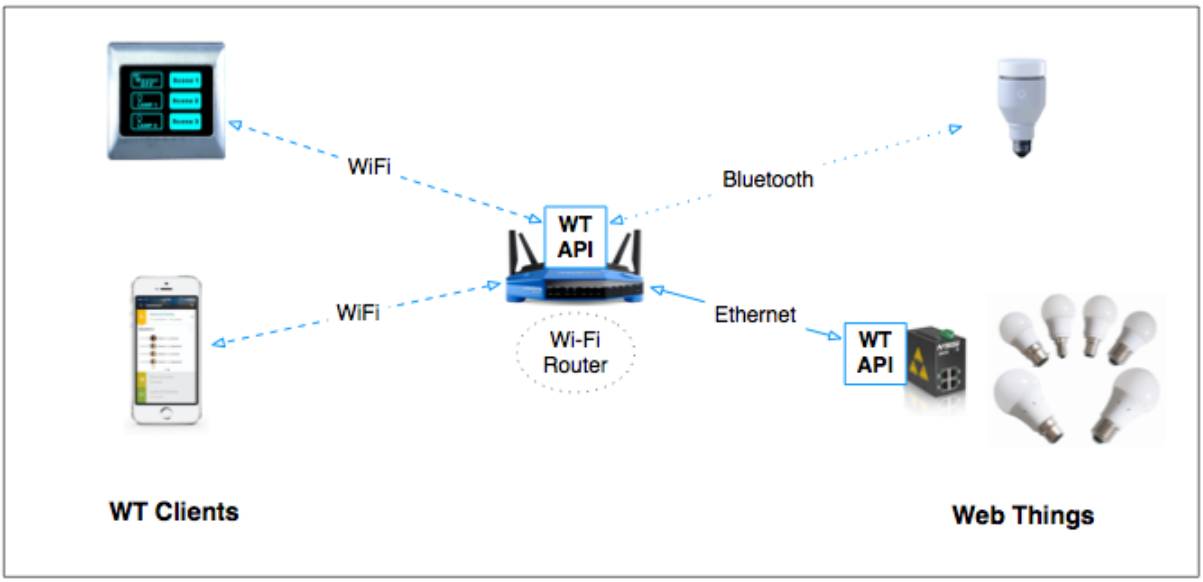
However, thanks to the distributed architecture of the Web where a proxy can be used to bridge those non-HTTP objects to the Web, any physical object can become a WT as soon as an HTTP server somewhere allows to access that device (directly or indirectly) over HTTP.
In other cases, some entities can only act as HTTP clients (i.e. they cannot serve HTTP or do not have a publicly accessible URL), but they can send HTTP requests to a server in the cloud. When an object isn’t electronically connected (e.g. a shipment with an RFID tag or QR code), a third party device must interact with the WT on its behalf.

Use Case 1 - Direct Connectivity over LAN



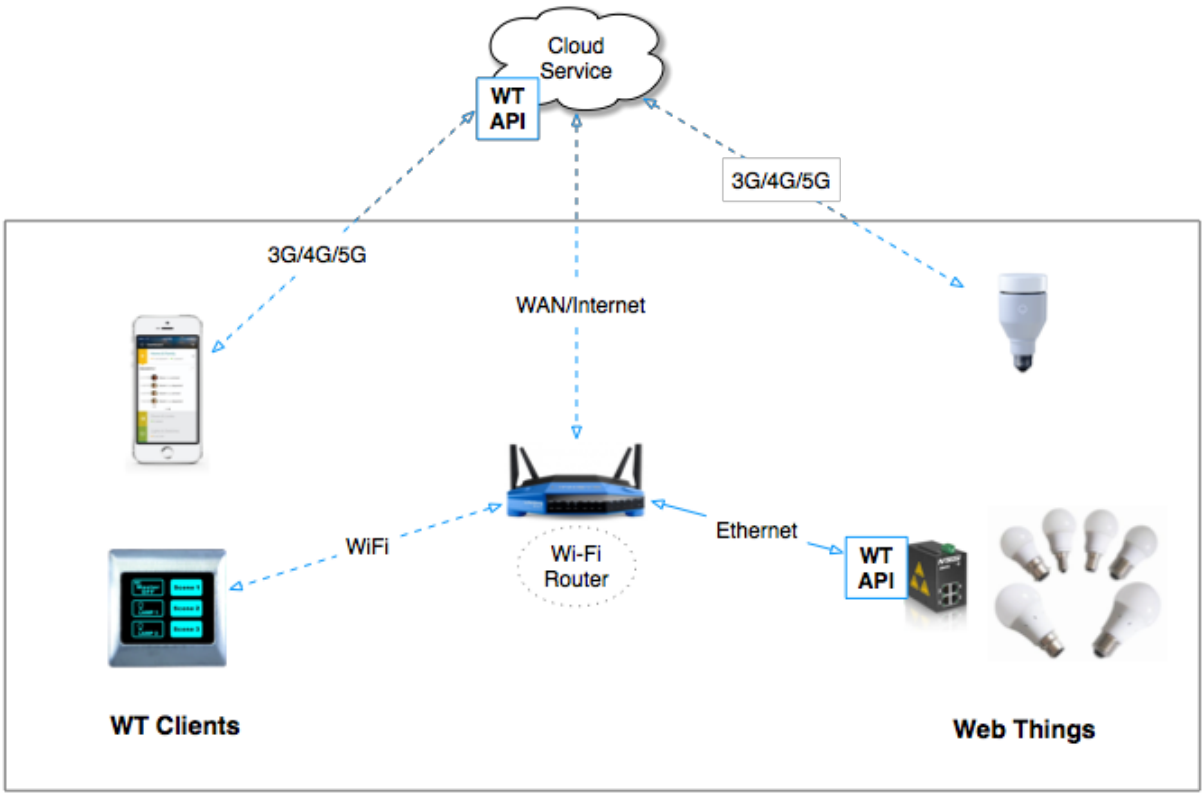
In the most straightforward case, a WT is simply an HTTP server that HTTP client send requests to. In some cases, the client and the WT can be on the same network (e.g. a native application on your mobile phone sends an HTTP request to a smart lamp – when both are connected to the same Wi-Fi network) or on different networks (e.g. you use the same application on your phone connected to a 4G network to send a request to a server in the cloud, and the lamp will poll or receive a notification from that server at a later stage. In both cases, you are sending the same request to a WT, the only difference is the URL to which you are sending the request (and obviously how the lamp gets the message).

Use Case 2 - Gateway-based connectivity over LAN



HTTP clients that don't directly expose an API is not a Web Thing – but it can become one when a third party Web server exposes an API on its behalf, therefore acting as a proxy (or gateway).

Use Case 3 - Cloud-based connectivity over WAN/Internet



Part 1 – Requirements for Web Things APIs

In this section we define the core requirements for a Web Thing (WT). These requirements are not a protocol per se, but only a set of important constraints over how the Web API of Web Things should be implemented so that Web Things can easily talk to each other or be integrated in innovative, Web-based applications.

If you're building a Web Thing client application, note that your application should assume that the "MUST" requirements are implemented on all WT, but the "SHOULD" and "MAY" might not.

Level 0 – MUST

This section defines the Level 0 requirements – all of which MUST be in place in any WT implementation, as all WT client applications will expect these constraints. If a WT implements all of them, it's compatibility with any other Web of Things entity is maximized.

R0 - WT MUST be an HTTP 1.1 server

All WT MUST support communication over HTTP 1.1. When possible, WTs SHOULD also support HTTP/2. WT clients SHOULD NOT expect that a WT always supports HTTP/2, but using HTTP 1.1 MUST always be possible with any WT.

HTTP 1.1 is the most widely spread protocol over the Web and virtually any HTTP client or server implementation supports it, therefore it's likely that the majority of clients might not support HTTP/2 as of yet.

Notes

An MQTT broker is not directly part of the Web of Things (because it doesn't use HTTP), just like database server isn't directly part of the Web. But exposing an MQTT broker over HTTP using a gateway connects the broker to the Web of Things. Just like the Web is a collection of documents accessible via HTTP, to be part of the Web of Things, any entity must be accessible via HTTP. An HTTP client (your browser) is not part of the Web, it only uses the Web to access Web resources. As mentioned, a database server is not part of the Web either - it needs an HTTP server to connect them to the Web. Therefore, the same should hold true for Web Things.

R0 - WT MUST have a root resource accessible via an HTTP URL

Any entity on the WoT must be both uniquely identifiable and addressable over a network so that other clients can interact with it. This means any WoT entity must have a so-called "**root resource**" identified by a HTTP URL (uses the HTTP protocol, therefore starts with `http://` or `https://`) that acts as the *entry point* for the Web Thing and enables the interaction with it. The URLs of all sub-resources of that Web Thing (streams, properties, etc.) are constructed by appending elements to the root URL of that Thing.

If the WT is a device connected on a LAN (e.g. a printer, a lamp, etc.), the root URL of a WT SHOULD be its IP address and standard HTTP port. In other words, it by only knowing the IP address of a networked device, if that device is a WT, it should be possible to do a GET on the IP address of the WT over the default port (80 for HTTP, or 443 for HTTPS).

This feature is envisioned for a facilitated mechanism for discovery of devices at the network level, without further protocol needs.

Notes

A WT does not need to be publicly accessible over the Internet, as some WoT scenarios might only work within a local area network (LAN) as defined above. This means a root URL does not need to be public and can be simply the local IP address of a device.

Examples of valid root URLs are:

```
http://gateway.api.com/devices/TV
http://kitchen-raspberry.device-lab.co.uk
https://192.168.10.10:9002
https://kitchen:3000/fridge/root
```

R0 - WT MUST support GET, POST, PUT, and DELETE HTTP verbs

The Web of Things aims to maximise interoperability by exposing the services of Things using REST. For this idea to come to reality, supporting certain HTTP verbs of the specification is a must. Considering the REST paradigm is based on resources and CRUD operations on them, the WT must support GET for reading operations, POST for creation, PUT for updates and state changes, and DELETE for removal.
The update of information of resources is usually done with the PUT (or PATCH) verbs, but as we are considering the possibility of restricted devices PUT/PATCH are rarely supported by certain libraries, therefore POST shall be used instead for updates.

Notes

We recommend that the “minimum” verbs WT need to support are GET (for reads) & POST (for writes) because they are the only two verbs supported by virtually any HTTP client implementation (including HTML forms). GET MUST only be used for safe and idempotent operations that do not have any effect upon the WT (reading the status, etc.) and that could be cached. POST MUST be used for any operation that has an effect on the device, which cannot be cached, such as edits and updates (which should be PUT) or creation of new resources (POST). If a WT is a read-only, for example a publicly shared pollution sensor in Chicago, it MAY NOT implement POST because GET is sufficient.

R0 - WT MUST implement HTTP status codes 200, 400, 500

Proper usage of HTTP 1.1 error codes is advisable. However, considering that some devices are restricted, only the following subset is mandatory:

- `2XX Success`. Returned for any successful action. The WT SHOULD at least support the `200 OK` response.
- `4XX Client Error`. This should be returned for any error due to the client that sent the request (from a wrong URL, to invalid authentication, or incorrect parameter). The WT SHOULD at least support the `400 Bad Request` error code.
- `5XX Server Error`. This should be used for any error on the WT side when a request was valid but cannot be processed at this time for any reason, e.g. the WT has a server error, etc. The WT SHOULD at least support the `500 Internal Server Error` error code.

R0 - WT MUST support JSON as default representation

A WT MUST always accept a valid JSON document as payload for request and always return a JSON representations when requested. If no "`Accept:`" HTTP header is specified in the request (or if the format is unknown or not supported), the WT SHOULD return an appropriate error code, but it also MAY return JSON as default. Additional formats MAY be supported by the WT, but JSON is the only default format that a WT MUST support.

R0 - WT MUST support UTF8 encoding for requests and responses

To avoid problems with special characters and localized content, any WT MUST always support UTF8 encoding for all requests and responses. A WT MAY support additional encodings , but a client should always be able to send an UTF8 to the server without specifying anything (and the server should interpret anything as UTF8 if a different encoding has not been specified).

R0 - WT MUST support GET on its root URL

Because all WT have a unique root URL and sometimes it's all we know about them, WT should always respond to HTTP GETs on their root URL and return their basic representation, so that clients of developers can use and understand it.

Level 1 - SHOULD

Unless there are technical or practical limitations for not adhering to Level 1 constraints, Web Things SHOULD always support the following constraints,

R1- WT SHOULD use secure HTTP connections (HTTPS)

Ideally, WT SHOULD always support secure connections with HTTPS over TLS. The ideal case for security is to support only HTTPS over TLS. However, for compatibility, WT Clients should support both as well, and not expect a device to support only HTTPS. WTs MAY use other security mechanisms if they want to (in addition or in place of HTTPS). In some cases, it might be impossible or impractical to implement HTTPS (for example in pure intranet scenario), in which case using another security mechanism is still highly recommended. WT that are exposed on the Internet MUST always implement HTTPS over TLS.

R1 - WT SHOULD support additional HTTP status codes as needed

Especially for errors, WT SHOULD support the most common HTTP status codes as applicable (415 Invalid media type, 403 Forbidden, 201 Created, 503 Unavailable, etc.).

R1 – WT SHOULD have a “doc” link in their root representation

The root resource of any WT should always have a link to a human-readable description about itself (and possibly machine-readable as well), which is done using the “doc” link type as described below. This link should be included in any representation of the root resource, and SHOULD be returned as a “Link” HTTP header when possible as well. For example, a GET on the IP address of a printer in my LAN should have a link its doc that is ideally hosted in the cloud (e.g. hp.com/printers/x22/doc), but might also be locally hosted if needed (or use a redirect).

R1 - WT SHOULD support the Web Streams Protocol

If possible, every WT should implement the Web Streams protocol defined below. It is not essential to use it for a device to be part of the Web of Things, but implementing it will allow much more flexibility and interoperability.

Level 2 - MAY

Any Web Thing MAY support these rules, but not expected to. It depends on the capabilities of the devices and implementation aspects.

R2 - WT MAY support OPTIONS for each of its resources

In particular, if possible the use of PUT and PATCH in addition to POST should be used for changing the state of a device (or sending a request to an actuator).

R2 - WT MAY have a default machine-readable documentation

WT may decide to provide their information in any machine-readable format available. As the only requirement is JSON it is left to the Web Thing developers to provide any alternative machine-readable format, like JSON-LD or JSON-schema.

R2 - WT MAY provide additional representation mechanisms (RDF, XML, JSON-LD)

WT can provide additional representations of the data they provide and even their own descriptions. Being JSON the only mandatory representation, the rest of the formats are up to the implementers if needed for additional purposes. The protocol does not define the mappings to other representations so far, we may consider adding them in future revisions if needed.

R2 - WT MAY offer an HTML interface/representation (UI)

Some WT may come with a pre-programed HTML representation or UI that could be used under some conditions to integrate WT in dashboard environments.

WT sensed information might require the use of special streaming protocols. For this situations WT can provide special subscription channels that will be tailored to the concrete streaming protocol.
Ideally, WT provide subscription mechanism so that interested parties can get notifications of the updates on their state (), being able to subscribe to streams and not have to continuously poll WTs for new data.
Subscriptions should be also configurable ideally to optimize the communication between WT and their subscribers. As the WT are HTTP based, alternatives for this subscription mechanism are, but not restricted to, HTTP Callbacks or Web Sockets.

Web Streams Model

As we described earlier, where the WT is implemented (whether on a device directly, on another device in the same network, or in the cloud), should be transparent to the client. Regardless of this distinction, we aim at providing a unified data model that can be used throughout all the components.

Web Things – Basic Model

Note that the WT requirements proposed earlier do not state anything about the actual interaction between WT and WT clients, i.e. the API description, conventions, or data models used by the WT. This section bridges this gap by proposing an actual data model to be used on top of Web Things as defined in the previous chapter.
The WT Model is defined by three base fields (id, name, description) and two JSON objects “properties” and “links” (that contain streams, actions, and subscriptions amongst others). The general JSON structure of any WT MUST be as follows:

1. Base fields

The Root Resource of Web Things MUST follow the following model:

Basic WT - Data Model

```
{
	"id":<STRING>,
	"name":<STRING>,
	"description":<STRING>,
	"configurations":<OBJECT>,
	"links":<OBJECT>,
	"tags":<ARRAY>,
	"customFields":<OBJECT>
}
```

WT MUST have the following base fields:

- ***id***: this is the unique identifier of the Web Thing. Ideally, this ID should be globally unique, so that there is no collision between objects (we recommend using a 20 character alphanumeric string – MongoIDs with an 8 character pseudo-random secret to avoid brute force finding of IDs).
- ***name***: human readable name of the Web Thing (the display name of that WE).

WT SHOULD have those fields:

- ***description***: human readable description of the Web Thing.

WT MAY have those fields:

- ***tags***: an array of keywords (as strings) about this WT.

Additional fields SHOULD NOT be used, as WT client implementations MAY ignore them. The properties and links objects SHOULD be defined for every WT, even though they MAY be empty.

Basic WT - Example

```
{
	"id":"WebObject-2423529879879875",
	"name":"Shopping Cart",
	"description":"Shopping Cart that updates its location as it moves",
	"tags":["cart","device","test"],
	"customFields":{"size":"20","color":"blue"},
	"configurations":{...},
	"links":{...}
}
```

2. “configurations” Object

Properties are static information about the WT, some of which have a type (well-known properties we can create schemas for, such as device information, network, IP, etc.). Properties should not be versioned, as they only represent the current information about the device. They can change anytime over the lifecycle of the WT, but their history will not be accessible.

The following property objects SHOULD be defined:

- ***system***: an object with various system-level information about the WT (such as the last reboot time, the OS or firmware version, etc.)
- ***connections***: an object with the various network interfaces and their public parameters.
- ***subscriptionOptions***: an object defining which subscription methods are available, if any. WT could support either HTTP callbacks or WebSockets.
- ***identifiers***: the various identifiers used by this WT in other reference systems (e.g. an EPC code, etc.).
- ***customFields***: extension point to define WT specific properties, these must be key-value pairs (where values MUST be strings, booleans, or numbers – not objects).

The properties object MAY define any custom key value pairs, but the keys should always start with the underscore symbol (“_”) to denote that it’s a user-defined custom property and to avoid collision with well-known properties. Custom-defined properties should always include the “schema” field with a URL to where the schema of those properties is defined.

Properties Object of a WT - Data Model

```
"properties":{
	"system":<OBJECT>,
```

```
        "connections":<OBJECT>,
        "subscriptionOptions":<OBJECT>,
        "identifiers":<OBJECT>,
        "customFields":<OBJECT>,
        "_myCustomProperty1":<OBJECT>,
        "_myCustomProperty2":<OBJECT>
}
```

Properties of a WT - Example

```
"meta":{
        "connections":{
                "ip": "198.39.3.2",
                "port": 8585,
                "rootUrl": "http://localhost:8585/",
                "publicUrl": "http://devices.webofthings.io/"
        },
        "system":{
                "softwareVersion": "4.0" ,
                "lastConnection": "2015-10-31T23:59:59.000Z"
        },
        "identifiers":{
                "serialNumber":"AX2332-00021",
                "ean":"9399392392"
        },
        "subscriptionOptions":{
                "http.callback":{},
                "http.websocket":{}
        },
        "customFields":{
                "creator":"COMPOSE Consortium Hardware Team"
        },
        "_myCustomMeta1":{},
        "_myCustomMeta2":{}

}
```

3. “links” object

The “links” object is the collection of all the “resources” available on that WT, therefore all the elements in the links MUST be addressable. This is particularly useful because of

HATEOAS, as this object contains all the outgoing links from the objects (it's sensors, subscriptions, etc.).

Thanks to the "links" object (https://tools.ietf.org/html/rfc5988), one can automatically follow links to the various resources. Looking at the Link header in HTTP (see Section 5 in https://tools.ietf.org/html/rfc5988), one can see that all links have the following schema:

```
"Link" ":" "<" URI-Reference ">" *( ";" link-param )
```

Examples:

```
Link: <http://me.com/Book/chapter2>; rel="previous"; title="previous chapter"
```

The fields we are interested in are the following:

- ***link***: link to the resource (sections 5.1 & 5.2 of RFC), normally should be
- ***rel***: pointer to additional information of the Web Thing, how it works and how to use it if needed. It could be either human or machine readable (Section 5.3 of RFC).
- ***title***: pointer to additional information of the Web Thing, how it works and how to use it if needed. It could be either human or machine readable.

Links Object of a WT - Data Model

```
"links":{
      "<TYPE1>":<OBJECT>,
      "<TYPE2>":<OBJECT>,
      "<TYPE3>":<OBJECT>
}
```

The links object is a collection of "link" elements, all of which must follow the following model.

Link of a WT - Detailed Data Model

```
"links":{
      "<TYPE>":{
        "link": <STRING>,
        "title": <STRING>,
        "schema": <STRING>
      },
      ...
}
```

A link element MAY have any fields, but all MUST contain a type as key, and an object with the following key-values: link, title, schema (optional).

The following relation types are formally defined by IANA (see section 6.2.2):

- ***meta***: link to the schema definition of the particular WT. Some aspects of the WT are not standardized as they depend on the concrete type, therefore the schema of the missing parts must be provided here.
- ***help***: pointer to additional information of the WT, how it works and how to use it if needed. It could be either human or machine readable.
- ***ui***: pointer to a web based interface of the Web Thing if available.
- ***self:*** the base URL to access the object (which MUST start with http:// or https://), can be local (LAN URL) or public (Web address, resolvable via DNS).

Additionally, Web Streams define the following “well-known” relation types:

- ***streams***: the streams offered by the WT (see Section 3.1.4).
- ***actions***: the actions offered by the WT (see Section 3.1.5).
- ***entities***: the other Web entities accessible through this WT (therefore acting as a gateway).

Well-known links

There are three types of resource that all Web Things can have

- `{thingURI}/actions/`
- `{thingURI}/events/`
- `{thingURI}/properties/`
- `{thingURI}/things/`

- `{thingURI}/product/`
- `{thingURI}/help/`
- `{thingURI}/meta/`
- `{thingURI}/ui/`

If you use this default URI for these resources, you do not need to list them in the “links” object, as clients can assume these links exist (but it’s their respo to check if they exist or not). Clients can make implicit assumptions: e.g., /properties.
This enables for instance to map existing APIs to this model (e.g., when a “thing” is called “device”).

Developers MAY define any custom links and relation type, in which case these SHOULD start with “_” to avoid collision, and SHOULD contain the optional “schema” key that points to a document that describes the “meaning” of that relation.

Links Object of a WT - Example

```
"links":{
       "properties": {
         "link": "properties/",
         "title": "List of Properties"
       },
       "events": {
         "link": "events/",
         "title": "List of Events"
       },
       "actions": {
         "link": "actions/",
         "title": "List of Actions"
       },
       "meta": {
         "link": "http://w3c.org/schemas/webthing/",
         "title": "Metadata"
       },
       "self": {
         "link": "/",
         "title": "Self"
       },
       "help": {
         "link": "http://webofthings.io/docs/pi/",
         "title": "Documentation"
       },
       "ui": {
         "link": "ui/",
         "title": "User Interface"
       },
       "_myCustomLinkRelType":{
         "link": "custom/",
         "schema": " http://webofthings.io/schemas/custom.html",
         "title": "My custom resource"
       }
}
```

Link Type #1: Streams

Any WT can have 0-n sensors that collect data periodically or sporadically and possibly store that data locally. Each sensor SHOULD have an associated data stream (a “sensor

stream") where sensor readings are pushed and made available (each reading is called a sensor update). There might be additional streams (a "virtual" stream) for other purposes, such as debugging, etc., therefore we refer to streams as any piece of information, the value of which changes over time and is exposed by the WT.

Streams Link Object of a WT - Model

```
"streams":["location","acceleration","temperature","myCustomStream"]
```

Each **Stream** must have a name, and the value MUST be a JSON object that follows this schema:

Stream Object of a WT - Model

```
"<URI-NAME>":{
	"name":"<STRING>",
	"description":"<STRING>",
	"lastUpdate":"<TIMESTAMP>",
	"unit":<SI_STRING>,
	"value":{
		"a":<CHANNEL>,
		"b":<CHANNEL>,
		"c":<CHANNEL>
	},
	"tags":<ARRAY>,
	"customFields":<OBJECT>
}
```

A stream object MUST define the following fields

- ***name***: a given name to the stream.
- ***lastUpdate***: this value is simply the timestamp (Unix epochs) of last time the readings have been updated. Updates are synchronized by stream; all the channels will share the update time.
- ***channels***: each stream might have one or more dimensions, which are dependent on the specific domain. We call these dimensions channels. The definition of the channels is domain specific, but typically, each channel will have a set of custom fields that are multiple key-value pairs for that particular reading. No model is suggested/enforced/validated (other than it must be a set of key value pairs) - the semantics should be established by the WT creator.

A stream object SHOULD define the following fields:

- ***description***: a human readable description of the stream.
- ***type***: an open field to classify the stream if needed. If not specified, the default value is “sensor”.

A stream object MAY define the following fields:

- ***customFields***: each stream has a customFields attribute, which is simply a set of key-value pairs to store current information about the sensor (not historical, updates are not stored). A sensor can have multiple channels and each channel has only one dimension (k-v pair, with its own customFields).

Each **Channel** of a stream MUST have a name and the value MUST be a JSON object that follows this schema:

Stream Channel Object of a WT - Model

```
"<NAME>":{
	"name":"<STRING>",
	"description":"<STRING>",
	"unit":<SI_STRING>,
	"type":"<STRING>",
	"current-value":"<STRING|NUM|BOOL>"
}
```

Stream Object of a WT - Example

```
"location":{
	"name":"Location",
	"description":"Location of the shopping cart",
	"lastUpdate":998239224,
	"value":{
		"long":{
		"name":"longitude",
		"unit":"degrees",
		"type":"numeric",
		"value":30.4
		},
		"lat":{
		"name":"latitude",
		"unit":"degrees",
```

```
                "type":"numeric",
                "value":70
                }
        },
        "customFields":{
                "model":"ModelX245-A Location Sensor",
                "measureError":"1%",
                "technology":"BLE"
        }
}

GET .../location/value
200 OK
{lat:22,long:15}

GET .../temperature/value
{temp:22} ---> default json

GET .../temperature/value
Accept: text/plain
22 ---> if you ask for text
```

Link Type #2 - Actions

An action is a pre-defined commands that WT clients can send to instruct a WT to do something. Actions should be used for anything that can be executed by a Web device, for example unlocking a door, displaying a text on a screen, restarting a device, etc. Unlike streams which are used as a dynamic set of values representing the current state of a thing (streams are sent/updated by a WT and read by clients), an action is an atomic command that a Web Thing can react to (by executing it) and are sent by external clients to a thing.

Each WT could support 0-m actions, representing predefined commands that can be sent for the WT to perform some kind of activity. These actions will have normally a set of parameters that can be configured in a key-value fashion. The description of these parameters depends on the WT and their schema should be provided in the schema definition. Actions are usually defined by the manufacturer and map to the actuation capabilities of a device. The Actions Object MUST follow the following model:

Actions Object of a WT - Model

```
"actions":{
	"lcd":{},
	"camera":{},
	"leds":{},
	"myActionType":{}
}
```

Each **Action** must have a (unique) name (<NAME>) and its value MUST be a JSON object that follows this schema:

Action Object of a WT - Model

```
"<NAME>":{
	"name":"<STRING>",
	"description":"<STRING>",
	"type":"<STRING>",
	"lastUpdate":"<TIMESTAMP>",
	"customFields":<OBJECT>,
	"parameters": {
		"a":<OBJECT>,
		"b":<OBJECT>,
		"c":<OBJECT>
	}
}
```

An action object MUST define the following fields:

- ***name***: Human-readable name of this action (display), for example (“Reboots the device”)

An action object SHOULD define the following fields:

- ***description***: human readable description of the action
- ***lastUpdate***: the timestamp when the action was invoked the last time.
- ***parameters***: set of parameters that are needed for the action. The definition of the parameters is dependent of the action.

An action object MAY define the following fields:

- ***customFields***: a set of key-value pairs to define various information configurations about this action type, how it's displayed, used, etc.
- ***status***: defines if the action is being executed and its last status (“success”, “failed”,”in progress”)

Each **Parameter** of an action MUST have a name and its value MUST be a JSON object that follows this schema:

Action Parameter Object of a WT - Model

```
"<NAME>":{
	"name":"<STRING>",
	"description":"<STRING>",
	"type":"<STRING>",
	"unit":"<STRING>",
	"optional":"<BOOLEAN>",
	"minValue":"<NUMERIC>",
	"maxValue":"<NUMERIC>",
	"maxLength":"<INTEGER>"
}
```

An Action Parameter Object MUST define the following fields:

- ***name***: Human-readable name of this parameter (display), for example (“Reboot delay”)
- ***type***: the type of this parameter. The possible values are “integer”, “float”, “string”, or “boolean”. Can be a URL to a remote parameter definition (a combination of the following parameters)

An Action Parameter Object SHOULD define the following fields:

- ***description***: human readable description of what this parameter is.
- ***optional***: whether this parameter is optional or not
- ***unit***: the unit of this parameter, usually as a SI unit. Can be a URL to a remote schema that defines a specific unit

An Action Parameter Object MAY define the following fields:

- ***minValue***: the minimum value this parameter should have (only if type is “numeric”)
- ***maxValue***: the maximum value this parameter should have (only if type is “numeric”)
- ***maxLength***: the maximum length this parameter should have (only if type is “string”)

Action Parameter Object of a WT - Example

```
"delay":{
	"type":"integer",
	"optional":"false",
	"minValue":"0",
	"maxValue":"100",
```

```
        "maxLength":"20",
        "unit":"milliseconds"
}
```

Link Type #3 - Entities

In some cases, a WT can act as a proxy for other WTs, in which case it is said to be a gateway. As an example, a WT can be a home router with a ZigBee interface to which various lamps are connected. Although the lamps themselves are not directly accessible over the Web, the gateway exposes them over the Web.

Link Type #4 – Subscriptions

Information provided by a WT could be accessed in many ways, and the various streams can always be polled individually by WT clients. However, in many situations it might be simpler and more efficient for WT clients to subscribe to one or several streams manually, so they can receive notifications (push) only when those streams are updated. These subscriptions, regardless of the actual underlying mechanism (be it entirely HTTP-based or not), SHOULD be visible and accessible via the Web Streams protocol.

As we have seen above, a special property is the “subscriptionOptions” object that contains the list (and protocol-specific) subscription mechanisms supported by the device. Besides, each WT has a special link object called “subscriptions”, that contains the current active subscriptions across those mechanisms.

Ideally, most WT SHOULD support at least HTTP callbacks as a subscription type. WT clients can then subscribe are made against streams or actions in a WT. The Subscriptions Object MUST follow the following model:

Subscriptions Link Object of a WT - Schema

```
"subscriptions":{
        "id1":{},
        "id2":{},
        ...
}
```

Subscription Object of a WT - Schema

```
"2349":{
        "subscriberId":"ServiceObject-123213213",
        "subscriptionId":"ServiceObject-123213213#location",
```

```
        "type":"http.callback",
        "delay":0,
        "expire":10000,
        "forceUpdates":false,
        "stream":"location",
        "customFields":{

"callbackUrl":"http://www.compose.eu/so/ServiceObject-123213213/callback"
        }
}
```

A subscription object MUST contain the following attributes:

- ***subscriberId***: an identifier of the subscriber.
- ***subscriptionId***: an identifier for the specific subscription to the stream.
- ***type***: defines the type of the subscription, mainly the mechanism for the notification (HTTP callbacks, WebSockets…).
- ***stream***: the stream to which the subscription is connected.

A subscription object SHOULD contain the following attributes:

- ***delay***: the minimal time in seconds between two notifications. Notifications are sent only when a new sensor reading is available, but at most once every delay seconds. If set to 0, then notifications are sent as soon as a new sensor reading has been done.
- ***expire***: it is the duration (in seconds) the subscription will be valid. The value can be between 0 (never) and 31536000 (365 days, max allowed).
- ***forceUpdates***: if set to false, notifications will be sent only if the sensor reading has changed since last notification. If the value is true: notifications will be sent each time (either periodically or sporadically).
- ***customFields***: here the specificities of the type of subscription can be added as a key value set of properties, for example, the url where to push the notifications can be described here. Or additional authentication tokens if needed.

Web Streams Protocol

The Web Streams Protocol defines an HTTP REST based protocol that governs the access to information provided by Web Things. Many operations apply as well to the interaction between Service Objects and Services in COMPOSE platform as the provided functionality is the same. The difference between them lies in the parts of the communication and the operations that can be done, sometimes in a slightly different way, but still following the same protocol.

As we are based on REST principles, the Web Things have a view of resources. Each group of properties is treated as resources too, triggering the functionality depending on the operations performed on these resources, mainly CRUD operations. The following table summarizes the operations that can be done for each type of object:

Operations of the Web Stream Protocol

Resource	Operations
{wt}	POST
	GET
	PUT
	DELETE
/thing/{id}/properties	POST
	GET
	PUT
	DELETE
/thing/{id}/streams/	-
	GET
	-
	-
/thing/{id}/streams/{id}/channels	-
	GET
	-
	-
/thing/{id}/actions	POST
	GET
	-
	DELETE

/thing/{id}/subscriptions	POST
	GET
	-
	DELETE
/thing/{id}/links	-
	GET
	-
	-

Web Things Details

Server-based Web Things support an embedded HTTP server, therefore provide additional REST capabilities. The operations allowed for working with server-based Web Things are as follows:

Working with Web Things

Getting the root resource of a Web Thing

As described above, a Web Thing must always return a basic representation of its root resource as a response to a GET request to the root URL of the Web Thing.

GET {rootURL}

```
GET /WebObject-2423529879879875 HTTP/1.1
Host: www.compose.eu
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

{
	"id":"WebObject-2423529879879875",
	"name":"Shopping Cart",
	"description":"Shopping Cart that updates location information as it moves",
	"tags":["cart","device","test"],
```

```
        "properties":{...},
        "links":{...}
}
```

Working with Properties

Getting the properties from a Web Thing

The properties of a Web Thing can be accessed by making GET requests to the following URL:

GET {rootURL}/properties/{propertyName}

Individual properties can also be accessed by using their name in the URL

REQUEST
GET /WebObject-2423529879879875/properties/connections HTTP/1.1 Host: www.compose-project.eu Accept: application/json
RESPONSE

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 639
"system":{
"softwareVersion": "4.0" ,
"lastConnection": "2015-10-31T23:59:59.000Z"
}

Working with Actions

Similar to streams in their structure, actions are mostly used to proxy actuators of the device and commands that can be sent to the device. As actions are also resources, the same URL structure applies.

```
{rootURL}/actions/{actionName}
```

Get list of actions of a Web Thing

```
GET /WebObject-2423529879879875/actions HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675
{
       "lcd":{...},
       "camera":{...},
       "leds":{...},
       "myActionType":{...}
}
```

Get Details of a Specific Action Type

`GET {rootURL}/actions/{actionName}`

REQUEST
GET /WebObject-2423529879879875/actions/myActionType HTTP/1.1 Host: www.compose-project.eu Accept: application/json
RESPONSE

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675
"myActionType":{
"name":"My Custom Action",
"description":"Reboots the device",
"lastUpdate":23123123,
"customFields":{
},
"parameters": {
"delay":{...},
"requester":{...},
}
}

Each parameter should define the following fields:

- ***type:*** or string, float, boolean, etc. OR URL to a Web-based semantic one (e.g. webofthings.io/schemas/temperature) - which can be templates/configs of the above

Send action command to a Web Thing

```
POST {rootURL}/actions/{actionName}
```

Example - Send a “reboot” command to a robot

```
POST /WebObject-2423529879879875/actions/reboot HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
Content-type: application/json
// ADD sample payload here


HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 165
Location: www.compose-project.eu/WebObject-2423529879879875/actions/reboot/12
// ADD sample response payload here
```

Cancel an action

```
DELETE {rootURL}/actions/{actionName}/{actionId}
```

Depending on the type of action, the Web Thing might allow the cancelation of it if still in progress. This is done by sending a delete request on the action-id received when the action is posted to the WT.

```
DELETE /WebObject-2423529879879875/actions/reboot/ServiceObject-1412413241#reboot
HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
HTTP/1.1 200 OK
```

Working with Streams

A stream of a WT is seen as a resource from the REST perspective so it can be accessed by appending the corresponding name to the URL. The properties of the streams and channels are equally accessible following this pattern.

Get the list of All streams of a WT

GET {rootURL}/streams/

REQUEST
GET /WebObject-2423529879879875/streams HTTP/1.1 Host: www.compose-project.eu Accept: application/json
RESPONSE

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675
"streams":{
"location":{...},
"acceleration":{...},
"temperature":{...},
"myCustomStream":{...}
}

Get a specific stream of a WT (the “location” stream)

GET {rootURL}/streams/location

REQUEST
GET /WebObject-2423529879879875/streams/location HTTP/1.1 Host: www.compose-project.eu Accept: application/json
RESPONSE

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675
{
"name":"Location",
"description":"Location of the shopping cart",
"type":"sensor",
"lastUpdate":998239224,
"channels":{
"long":{
"name":"longitude",

```
                "unit":"degrees",
                "type":"numeric",
                "current-value":30
        },
        "lat":{
                "name":"latitude",
                "unit":"degrees",
                "type":"numeric",
                "current-value":70
        }
        },
        "customFields":{
        "model":"ModelX245-A Location Sensor",
        "measureError":"1%",
        "technology":"BLE"
        }
}
```

Get channels available in a stream

`GET {rootURL}/streams/location/channels`

REQUEST

```
GET /WebObject-2423529879879875/streams/location/channels
HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675
{
        "long":{
        "name":"longitude",
        "unit":"degrees",
        "type":"numeric",
        "current-value":30
        },
```

```
        "lat":{
        "name":"latitude",
        "unit":"degrees",
        "type":"numeric",
        "current-value":70
        }
}
```

Some considerations about the channels:

- Each stream is independent; there is no implicit synchronization between streams by default.
- All channels are synchronized, they have the same timestamp (not channel-specific timestamp). The timestamp is stream specific.
- Channel type is dependent on the implementation of the Web Thing. They can be defined as a simple string or numeric value, as well as a more complex byte array or binary data.

Working with Subscriptions

Web Things might provide subscription mechanisms to the streams of data they update. Subscribing to a stream updates are done by posting a subscription request to the subscriptions resource of a Web Thing.

`POST {rootURL}/subscriptions`

Create subscriptions to streams – HTTP callback

REQUEST

```
POST /WebObject-2423529879879875/subscriptions HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
Content-type: application/json
{
        "subscriberId":"ServiceObject-123213213",
        "type":"http.callback",
        "delay":0,
        "expire":10000,
        "forceUpdates":false,
        "stream":"location"
        "customFields":{

"callbackUrl":"http://www.compose-project.eu/so/ServiceObje
ct-123213213/callback"
```

```
        }
}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675
{

"subscriptionId":"ServiceObject-123213213#subscription"

}
```

Create subscriptions to streams - WebSockets

REQUEST

```
POST
/WebObject-2423529879879875/streams/location/subscriptions
HTTP/1.1
Host: www.compose-project.eu
Accept: application/json
Content-type: application/json
{
        "subscriberId":"ServiceObject-123213213",
        "type":"http.websocket",
        "delay":0,
        "expire":10000,
        "forceUpdates":false,
        "stream":"location"
}
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1675
{

"subscriptionId":"ServiceObject-123213213#subscription",
        "customFields":{
```

```
"websocket":"wss://www.compose-project.eu/WebObject-2423529
879879875/streams/location/"
        }
}
```

When opening a WebSockets channel for notifications, the sequence of activities is as follows:

- The subscriber POSTs s subscription request stating that the type of subscription corresponds to a Web Socket.
- If the Web Thing accepts the subscription, it provides the WebSocket address of the stream and sends as stated in the subscription the corresponding updates as JSON objects.

Get/Delete/Update information about subscriptions to sensor data

As subscriptions are created they become resources, accessible using the same strategy as any other resource. Sending a GET request on a subscriptionId the details of the subscription can be retrieved. Accordingly, sending DELETE and PUT requests will cancel and update the subscription respectively.

[1] http://stateless.co/hal_specification.html

[2] http://amundsen.com/media-types/collection/

[3] http://json-ld.org/