WoT Thing Description

IRTF T2TRG Workshop on IoT Semantic/Hypermedia Interoperability

Authors:
Victor Charpenay (Siemens AG)
Sebastian Käbisch (Siemens AG)
Matthias Kovatsch (Siemens AG)

Abstract

The WoT Thing Description (TD) is the central building block of the W3C Web of Things (WoT). It provides metadata of the possible interactions, data model, communication, as well as security mechanisms of the Thing. For this, it uses a simple, but common interaction model and pluggable domain-specific vocabularies for semantic annotations. Its serializations range from JSON-based to concise binary formats. A popular serialization is JSON-LD, which can be consumed by classic JSON parsers, but also conveyes rich semantics through machine-understandable triples.

Status of This Document

This document is an excerpt of the Web of Things Interest Group Current Practices, which is maintained on GitHub

Table of Contents

1. Introduction

The central idea of WoT is that Things can describe their capabilities and metadata in a machine-understandable format: the WoT Thing Description (TD), which is rooted in RDF. The TD can be served by a Thing itself or hosted elsewhere on the Web. This way, a TD can also be retrofitted on existing devices, and thereby complement existing IoT platforms and standards with rich metadata to enable interoperability across platforms. WoT also supports flexible Protocol Bindings that enable the mapping of a semantically described Thing interaction to different protocols, since each domain has different requirements here. Finally, WoT envisions to provide a common runtime environment for IoT-related apps. Sensors and actuators can be improved by installing better data processing algorithms or by adding completely new features through software. With a common runtime environment, aggregation apps can be instantiated in the cloud and then relocated to local hubs, or even directly onto a powerful Thing, to meet certain quality of service requirements.

(...)

2. Terminology

This document uses the following terms defined elsewhere.

CoAP
: Constrained Application Protocol [RFC7252]

JSON-LD
: A JSON document that is augmented with support for Linked Data by providing an `@context` property with a defining URI [JSON-LD]

JWT
: JSON Web Token [RFC7519]

Protocol Binding
: A mapping from operations on the WoT resource model to specific operations of a protocol (see 3.1.2 Protocol Bindings)

RDF
: The Resource Description Framework (RDF) of the Semantic Web [rdf11-concepts]

Repository
: A registry for TDs that provides a Web interface to register TDs and look them up, for intance using SPARQL queries

Scripting API
: Programming interface that allows scripts to discover Things through a Discovery API, issue requests through a Client API, provide resources through a Server API, and access directly attached hardware through a Physical API (see)

Servient
: The addressable application endpoint of a Thing that makes it interactive by providing a WoT Interface and means to execute application logic

SPARQL
: A query language for semantic data

Thing
: The abstract concept of a physical entity that can either be a real-world artifact, such as a device, or a virtual entity that represents physicality, such as a room or group of devices

Thing Description
: An RDF document (currently serialized in JSON-LD by default) that contains semantic and functional descriptions of a Thing (see 3.2 Thing Description for details)

WoT Interface

Resource-oriented Web interface (often called "Web API") that allows access to servients over the network using different Protocol Bindings (see 3.1 WoT Interface)

3. Concepts & Building Blocks

(...)

3.2 Thing Description

The WoT Thing Description (TD) provides the semantic metadata of a Thing as well as a functional description of its WoT Interface. For this, it relies on the Resource Description Framework (RDF) [rdf11-concepts] as an underlying data model. For now, [JSON-LD] is used as the default TD serialization format. The WoT IG defined a minimal vocabulary to express the capabilities of a Thing in terms of different interaction patterns: *Properties*, *Actions*, and *Events*. In addition, the TD provides metadata for the different communication bindings (e.g., HTTP, CoAP, etc.), mediaTypes (e.g., "application/json", "application/exi", etc.), and security policies (authentication, authorization, etc.). Figure 3 Concepts of the Thing Description (TD) gives an overview of the relevant content defined in a TD.

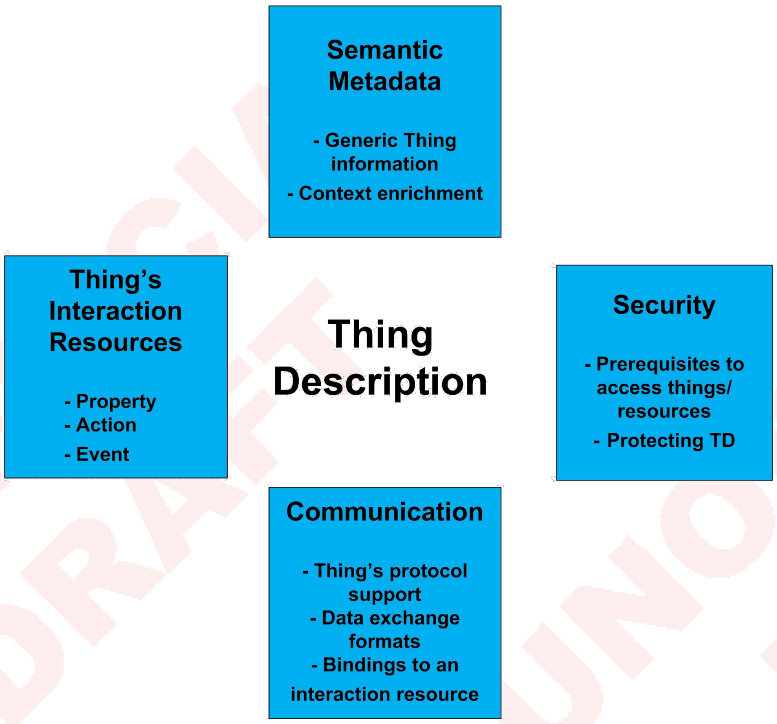

Semantic Metadata
- Generic Thing information
- Context enrichment
Thing's Interaction Resources
- Property
- Action
- Event
Thing Description
Security
- Prerequisites to access things/ resources
- Protecting TD
Communication
- Thing's protocol support
- Data exchange formats
- Bindings to an interaction resource

Figure 3 Concepts of the Thing Description (TD)

The subsequent subsection will give a brief introduction to the TD as JSON-LD samples. This is followed by more detailed explainations of the TD elements, a number of more complex TD examples, and finally the considerations for use in production.

3.2.1 Quick Start: TD Samples

In the following, we introduce three minimal TD samples to show the prospects of the W3C Thing Descriptions. Example 1 shows a simple TD that describes a temperature Thing with the name *MyTemperatureThing*.

EXAMPLE 1: Data Only

```
{
  "@context": ["http://w3c.github.io/wot/w3c-wot-td-context.jsonld"],
  "@type": ["Thing"],
  "name": "MyTemperatureThing",
  "interaction": [
    {
      "@type": ["Property"],
      "name": "temperature",
      "outputData": { "type": "number" },
      "writable": false,
      "link": [{
        "href" : "coap://mytemp.example.com:5683/temp",
        "mediaType": "application/json"
        }]
    }
  ]
}
```

Based on this content, we know there exists one *Property* interaction resource with the name *temperature*. In addition, information is provided such as that this Property is accessable over the CoAP protocol with a GET method (see CoAP protocol binding here) at `coap://mytemp.example.com:5683/temp` (announced within the endoind structure by the *href* kye), which will return a number (TD type system) inside a JSON structure (JSON as payload format is announced by the *mediaType* field).

In practice, a Thing provides further details about what kind of Thing it is and what the interactions mean. This additional information is the semantic context of the Thing. JSON-LD provides a means to extend a TD with an external (semantic) context, which allows to reuse existing models, thereby enhancing semantic interoperability. Through a context, the meaningless strings turn into semantically defined terms that are part of a linkable vocabulary. Example 2 shows the integration of the `sensor` namespace and the resulting semantic enrichment of the temperature Property with an additional entry within the `@type` and unit assignment.

EXAMPLE 2: Semantic Annotations

```
{
"@context": ["http://w3c.github.io/wot/w3c-wot-td-context.jsonld",
    { "sensor": "http://example.org/sensors#" }
   ],
  "@type": ["Thing"],
  "name": "MyTemperatureThing",
  "interaction": [
    {
      "@type": ["Property","sensor:Temperature"],
      "name": "temperature",
      "sensor:unit": "sensor:Celsius",
      "outputData":  { "type": "number" },
      "writable": false,
      "link": [{
        "href" : "coap://mytemp.example.com:5683/",
        "mediaType": "application/json"
        }]
    }
  ]
}
```

Example 3 shows a more advanced TD reflecting an LED Thing that supports multiple protocols (CoAP and HTTP), mediaTypes (JSON and EXI), and security policies (it requires a JWT for interaction). This example also shows the definition of the endpoint information in a global manner. More precisely, a global defined *base* URI is valid for defined interaction resources. Indivdualiziation of the URI's resource is done by the *href* key within the interaction definitions.

EXAMPLE 3: More Capabilities

```
{
 "@context": [
  "http://w3c.github.io/wot/w3c-wot-td-context.jsonld",
  { "actuator": "http://example.org/actuator#" }
 ],
 "@type": ["Thing"],
 "name": "MyLEDThing",
 "base": "coap://myled.example.com:5683/",
 "security": {
   "cat": "token:jwt",
   "alg": "HS256",
   "as": "https://authority-issuing.example.org"
  },
 "interaction": [
  {
   "@type": ["Property","actuator:onOffStatus"],
   "name": "status",
   "outputData": { "type": "boolean" },
   "writable": true,
   "link": [{
    "href" : "pwr",
    "mediaType": "application/exi"
   },
   {
    "href" : "http://mytemp.example.com:8080/status",
    "mediaType": "application/json"
   }]
  },
  {
   "@type": ["Action","actuator:fadeIn"],
   "name": "fadeIn",
   "inputData": { "type": "integer" },
   "link": [{
    "href" : "in",
    "mediaType": "application/exi"
   },
   {
    "href" : "http://mytemp.example.com:8080/in",
    "mediaType": "application/json"
   }]
  },
  {
   "@type": ["Action","actuator:fadeOut"],
   "name": "fadeOut",
   "inputData": { "type": "integer" },
   "link": [{
    "href" : "out",
    "mediaType": "application/exi"
   },
   {
    "href" : "http://mytemp.example.com:8080/out",
    "mediaType": "application/json"
   }]
  },
  {
   "@type": ["Event","actuator:alert"],
   "name": "criticalCondition",
   "outputData": { "type": "string" },
   "link": [{
              "href" : "ev",
              "mediaType": "application/exi"
            }]
    ]
  }
```

Based on this TD, one is able to know that a JSON Web Token (JWT) is required to interact with the resources of the Thing. They are issued by `https://authority-issuing.example.org` and signed using HMAC SHA-256 (HS256). The status of the Thing can be requested using a CoAP GET on `coap://myled.example.com:5683/pwr` (*base* URI from the global scope endpoint definition plus the href) or an HTTP GET on `http://mything.example.com:8080/myled/status`. It can also modified using a PUT on the respective URIs, since Property is writable. The Actions can be invoked through a POST to the resources fadeIn (`/in` for CoAP and `/myled/in` for HTTP) and fadeOut (`/out` for CoAP and `/myled/out` for HTTP) with an integer value in milliseconds serialized as JSON or EXI. Please note, all this protocol binding assumptions are explained here. MyLEDThing also serves for the CoAP protocol an Event called criticalCondition, which enables clients to be informed about problems (enabled by the CoAP GET with the Observe option to `coap://myled.example.com:5683/ev`). This event sample also shows the local scope definition of the endpoint which overwrite global defined endpoint information. Such a mechanism allows to restrict or expand endpoint oppertunities of indivdual interaction resources.

3.2.2 Semantic Metadata

3.2.2.1 TD Context

JSON-LD is a serialization format that adds a semantic layer on top of the JSON specification: the terms that appear in a JSON document should be associated with uniquely identified concepts from shared vocabularies. This principle is part of a set of practices to publish data on the Web called Linked Data, where concepts are usually identified with URIs and originate from RDF vocabularies.

The association between terms and concept URIs has to be declared in preamble of the JSON document with the keyword `@context`. The expected value for `@context` can be of different kinds. A first option is to use a JSON object where keys are terms and values are concepts URIs, e.g.:

EXAMPLE 4

```
{
  "@context": {
    "name": "https://w3c.github.io/wot/w3c-wot-td-ontology.owl#name",
  "base": "https://w3c.github.io/wot/w3c-wot-td-ontology.owl#associatec
Uri",
  "unit": "http://purl.oclc.org/NET/ssnx/qu/qu-rec20#unit",
  "Thing": "https://w3c.github.io/wot/w3c-wot-td-ontology.owl#Thing",
    ...
 }
}
```

It is also possible to declare namespaces in the context instead of terms. In the example above, three of the four URIs have the same prefix: `http://www.w3c.org/wot/td#`. This common part could be given a short name, refered to as a namespace (as in XML). URIs could then be shortened by concatenating namespace with `:` and a local name (anywhere in the JSON-LD document). E.g.:

EXAMPLE 5

```
{
  "@context": {
  "wot": "https://w3c.github.io/wot/w3c-wot-td-ontology.owl#",
    "name": "wot:name",
  "base": "wot:associatedUri",
  "unit": "http://purl.oclc.org/NET/ssnx/qu/qu-rec20#unit",
  "Thing": "wot:Thing",
    ...
 }
}
```

For the sake of reusability, it is also possible to define an external JSON-LD context and simply give its URI as value of `@context`. All JSON terms that are defined in the present document have been put in an external document, available at `http://w3c.github.io/wot/w3c-wot-td-context.jsonld`. It is highly recommended (but not mandatory) to include this standard context in a Thing Description. A basic Thing Description would contain the following declaration:

EXAMPLE 6

```
{
  "@context": "http://w3c.github.io/wot/w3c-wot-td-context.jsonld",
 ...
}
```

A third option is to declare an array, in case a single document involves several contexts. Array elements are either objects or strings, as explained above. This option proves relevant if one wants to extend the existing TD context without modifying it. For instance:

EXAMPLE 7

```
{
  "@context": ["http://w3c.github.io/wot/w3c-wot-td-context.jsonld",
               {"sensor": "http://example.org/sensors#"}],
  ...
}
```

Section 3.2.6.1 Extending Thing Description with Other Semantic Models provides a more concrete example on that topic.

3.2.2.2 Security

EXAMPLE 8

```
{
 ...
 "security": {"cat":"token:jwt", "alg":"HS256", "as":"https://authorit
y-issuing.org"},
 ...
}
```

The **(optional)** security field can be used to provide access metadata (self-contained) information of the Thing for securely transmitting information via all its resources. Also see Section 4. Security Considerations.

Here as a example, JSON Web Token (JWT) type is assigned (cat), the corresponding hashing algorithm "HS256" (alg), and issuing authority of the security token (as).

3.2.2.3 Thing Metadata

A TD will provide some generic metadata vocabularies that can be used, e.g., to assign a name or what kind of protocols does a servient support.

EXAMPLE 9

```
{
 ...
  "name": "MyLED",
  "base": "coap://myled.example.com:5683/"
 ...
}
```

There are three mandatory and one optional (shown as italic text in JSON snippet) vocabulary terms defined within metadata:

- **name:** Name of the Thing (string-based)
- **base: [optional]** Define base URI that is valid for all defined local interaction resources. All other URIs in the TD must then be resolved using the algorithm defined in [RFC3986]
- **security: [optional]** The security field can be used to provide access metadata (self-contained) information of the Thing for securely transmitting information via all its resources. Also see Section 4. Security Considerations. Above, the security field is used to announce that JSON Web Token (JWT) has to be used to interact with the resources of the Thing. Thereby, type is assigned by the cat field, the corresponding hashing algorithm "HS256" by the alg field, and the issuing authority of the security token by the as field.
- **@type: [optional]** RDF type of this Thing Descripton that defines the semantics through a Linked Data vocabulary (Thing is the default entry) within an array structure. Additional vocabulary can be added to the array structure to further enrich the Array with semantics.

Note: Besides these pre-defined terms in the TD context, additional characteristics can be added such as product ID, firmware version, location, etc. These terms should then appear in the context of the Thing (as detailed in 3.2.2.1 TD Context).

3.2.3 Interaction Patterns

An interaction in the context of the Web of Things is an exchange of data between a Web client and a Thing. This data can be either given as input by the client, returned as output by the Thing or both. Three interaction patterns have been defined so far: Property, Action and Event.

3.2.3.1 Property

Property provides readable and/or writeable data that can be static (e.g., supported mode, rated output voltage, etc.) or dynamic (e.g., current fill level of water, minimum recorded temperature, etc.).

EXAMPLE 10

```
{
 ...
 "interaction": [
    {
      "@type": ["Property"],
      "name": "temperature",
      "outputData":{ "type": "number" },
      "writable": false,
      "link": [{
       "href" : "coap://mytemp.example.com:5683/temp",
       "mediaType": "application/json"
      }],
      "stability": 10
    },
 ...
}
```

There are four mandatory and three optional (italic in JSON sample) terms defined for the Property pattern:

- **@type:** RDF type of the interaction that defines the semantics through a Linked Data vocabulary (Property is the default value) within an array structure. Additional vocabulary can be added to the array structure to further enrich the Property with semantics.
- **name:** Name of the Property, which can also be used for simple semantic disambiguation
- **outputData:** Which data type is associated with this Property (see3.2.4 Type System for details)
- **writable:** Is this Property writable (`true`/`false`)
- **link:** Local link definitions (as array) with href and mediaType definition

- **stability: [optional]** Expected period in ms the Property value is expected not to change (>0=estimated period; 0=irregular change; -1=static value)
- **security: [optional]** Access metadata (self-contained) for protecting this Property and securely transmitting information. Compared to the security field that can be found in the Thing metadata, this field here can be used to apply specific security requirements that is only valid for this resource.

Absolute URIs in `hrefs` or endoint can point to an external resource that is not hosted on the Thing directly. This can be useful to link data into the context of the Thing, for which there is not enough space in the device itself. Absolute URIs could also point to a proxy or gateway that allows access to legacy systems that might belong to a Thing that does not have the capability to translate by itself (but provides other, WoT-compatible resources). This mechanism is also interesting for virtual Things that combine the services of multiple other Things (e.g., a room lighting servient that has Properties for each individual light, while these linked Properties are hosted directly on those lights).

If the property is writable (`writable=true`), then the Property accepts the same format(s) as input as described for its output.

The stability field provides a hint for caching and polling. This value should also be included in the cache control information of protocols, e.g., the Cache-Control header field of HTTP or Max-Age option of CoAP.

3.2.3.2 Action

The Action interaction pattern targets changes or processes on a Thing that take a certain time to complete (i.e., actions cannot be applied instantaneously like property writes). Examples include an LED fade in, moving a robot, brewing a cup of coffee, etc. Usually, ongoing Actions are modelled as Task resources, which are created when an Action invocation is received by the Thing.

EXAMPLE 11

```
{
 ...
  "interaction": [
    {
      "@type": ["Action", "actuator:fadeIn"],
      "name": "fadeIn",
      "inputData": { "type": "integer" },
      "link": [{
       "href" : "coap://mytemp.example.com:5683/in",
       "mediaType": "application/json"
      }]
    }
  ]
 ...
}
```

There are one mandatory and four optional terms defined within the`Action` type:

- **@type:** RDF type of the interaction that defines the semantics through a Linked Data vocabulary (Action is the default entry).
- **name:** Name of the Action, which can also be used for simple semantic disambiguation.
- **inputData: [optional]** The call parameters associated with this Action.
- **outputData: [optional]** Which data is resulting from this Action (same condition as `inputData` above).
- **link:** Local link definitions (as array) with href and mediaType definition.
- **security: [optional]** Access metadata (self-contained) for protecting this Action and securely transmitting information.

The hypertext reference (`href`) field works similar to Properties.

Usually, invoking an Action results in a response that indicates a new (sub-)resource, where the started Task can be monitored and also controlled: Updating this Task resource may allow for modification of the process (e.g., when it is still queued and not started yet, but also during runtime if the process supports that). Deleting this Task resource may allow for cancellation of the Action execution. Once the Task completes, the (sub-)resource may be removed by the server; or it is marked as completed, but kept for traceability. It is also possible that calling an action produces a so-called action result that is not manifested in any temporary resource nor other changes to the server (e.g., a conversion process or dry run).

3.2.3.3 Event

The `Event` interaction pattern enables a mechanism to be notified by a Thing on a certain condition. While some protocols such as CoAP can provide such a mechanism natively, others do not. Furthermore, Events might need a specific configuration that requires data sent and stored on the Thing in a standard way. There are are two mandatory and two optional terms defined within the `Event` pattern:

EXAMPLE 12

```
{
 ...
  "interaction": [
    {
      "@type": ["Event", "actuator:alert"],
      "name": "criticalCondition",
      "outputData":  { "type": "string" },
      "link": [{
       "href" : "coap://mytemp.example.com:5683/ev",
       "mediaType": "application/json"
      }]
    }
  ]
 ...
}
```

- **@type: [optional]** RDF type of the interaction that defines the semantics through a Linked Data vocabulary (Event is the default entry) within an array structure. Additional vocabulary can be added to the array structure to further enrich the Event with semantics.
- **name:** Name of the Event, which can also be used for simple semantic disambiguation
- **link:** Local link definitions (as array) with href and mediaType definition
- **outputData: [optional]** Configuration data associated with this Event. This field works similar to the one of the Action pattern.
- **security: [optional]** Access metadata (self-contained) for protecting this Event and securely transmitting information.

EDITOR'S NOTE

So far, the concept of events has not been evaluated yet during the PlugFests. Thus, there has been little discussion and no common practice has emerged. The following text is a proposal to trigger discussion.

The interaction with Events works similar to Actions with the difference that no side effects are intended on the Thing. A client issues a request to the URI of an Event, which may include a payload that represents the configuration data for the Event mechanism. Upon reception, the Thing creates a (sub-)resource, which serves as a handle for the Event subscription. It can be monitored to receive notifications: an HTTP client would need to poll this resource, while a CoAP client would simply observe it. Technically, multiple clients could use the subscription resource created by another client when they are interested in the same Event configuration. The Event configuration can also be updated and ultimatively deleted to clean up internal notification hooks.

3.2.4 Type System

All of the three interaction types `Property`, `Action`, and `Event` can specify the type of the value acceptable as an input or expected as an output of the interaction. JSON schema [draft-zyp-json-schema-04] [draft-fge-json-schema-validation-00] provides a standard way to describe the structure and datatypes of data, and we use JSON schema as the notation for value types of the interaction types.

Note: In the embodiment of type system described in this section, JSON schema is used as an abstract description system for structured data. JSON schema is something that is already out there being used and well-known, therefore gives us a chance to get started quickly to experiment with type system idea for further contemplation. Through discussion, the following points have been noted as feedback for improvements.

- JSON schema specification was designed for JSON, therefore, may not be an optimal solution as an encoding-neutral description system.
- JSON schema does not provide a way to associate data elements with citations or semantic concepts defined elsewhere.
- JSON schema is relatively verbose.

Note that there is an idea (Proposal for the type system for Things) that was incubated independently and was put forward to the IG for consideration.

For `Property` and `Event` interaction types, each property or event can specify its value type using `outputData` with JSON schema definition as its content. In the following example, `temperature` property and `criticalCondition` event are defined as `number` and `string`, respectively.

EXAMPLE 13

```
{
 ...
  "properties": [
    {
   ...
      "name": "temperature",
      "outputData": { "type": "number" },
   ...
    }
  ]
 ...
  "events": [
    {
   ...
      "name": "criticalCondition",
      "outputData": { "type": "string" },
   ...
    }
  ]
 ...
}
```

Similarly, for `Action` interaction type, each action can specify its input and/or output value type with JSON schema definition as its content. The following is an example value type definition for an action. In the example, `fadeIn` action is defined to have `inputData` of type `number` and `outputData` of type `boolean`.

EXAMPLE 14

```
{
 ...
  "actions": [
    {
   ...
      "name": "fadeIn",
      "inputData": { "type": "number" },
      "outputData":{ "type": "boolean" },

   ...
    }
  ]
 ...
}
```

3.2.4.1 Simple Data

With value types described by means of JSON schema, serialization of data exchanged between servients is straightforward when it is in JSON format. JSON Schema allows the following definitions in the `type` field:

- boolean
- integer
- number
- string
- array
- object

Consider the following `inputData` definition which defines the value to be an `integer` within the value range of [0 ... 255].

EXAMPLE 15

```
"inputData": {
 "type": "integer",
 "minimum": 0,
 "maximum": 255,
}
```

When the integer being exchanged is 123, data serialization in JSON format will look like the following:

EXAMPLE 16

```
123
```

The same data (i.e. a number of 123) will look like the following when the data is exchanged in XML.

EXAMPLE 17

```
<integer>123</integer>
```

NOTE: Wrapping single values in a JSON object

W3C WoT now follows RFC 7159, which allows sending simple types in the root of the document. There is no wrapping of values into an object.

3.2.4.2 Structured Data

In the previous section, we used an example inputData definition consisting of a single integer.

Since we are using JSON schema to describe inputData, it is also possible to define value types that have more than one literal value. JSON provides two distinct constructs to define a structure that can have multiple literal values. One is JSON object, and the other is JSON array.

3.2.4.2.1 JSON Object

The following is an example inputData definition that defines the value to be an object that consists of two named literals id (of type integer) and name (of type string) where id is required to be present.

EXAMPLE 18

```
"inputData": {
    "type": "object",
    "properties": {
        "id": {
            "type": "integer"
        },
        "name": {
            "type": "string"
        }
    },
    "required": ["id"]
}
```

When the `id` number and the `name` string values being exchanged are 12345 and "Web of Things", data serialization in JSON format will look like the following.

EXAMPLE 19

```
{
 "id": 12345,
 "name": "Web of Things"
}
```

The above data will look the following when the data is exchanged in XML.

EXAMPLE 20

```
<object>
 <id>
  <integer>12345</integer>
 </id>
 <name>
  <string>Web of Things</string>
 </name>
</object>
```

3.2.4.2.2 JSON Array

The following is an example `inputData` definition that defines the value to be an `array` that consists of exactly three number literals with each value within the range of [0 ... 255].

EXAMPLE 21

```
"inputData": {
    "type": "array",
    "items": {
        "type" : "number",
        "minimum": 0,
        "maximum": 255
    },
    "minItems" : 3,
    "maxItems" : 3
}
```

When the numbers being exchanged are 208, 32 and 144, data serialization in JSON format will look like the following.

EXAMPLE 22

```
[
 208,
 32,
 144
]
```

The above data will look the following when the data is exchanged in XML.

EXAMPLE 23

```
<array>
 <number>208</number>
 <number>32</number>
 <number>144</number>
</array>
```

3.2.4.3 Mapping to XML Schema

In the previous section, examples showed what those data whose value type are described using JSON schema look like when serialized to XML in parallel to corresponding JSON serializations.

This section describes how JSON schema definitions can be mapped to XML schema definitions by using the same examples. Given JSON schema definitions, providing the mapping to XML schema allows XML tools to directly validate serialized XML data, for example.

The XML structure is based on EXI4JSON [exi-for-json]. The structure works uniformly well for both schema-less and schema-informed use cases.

EDITOR'S NOTE

A complete "JSON Schema" to "XML Schema" mapping needs to be defined.

3.2.4.3.1 JSON Object Definition to XML Schema

Shown below is the JSON schema `object` definition used as the `inputData` in Section JSON Object. The `object` consists of two named literals `id` (of type `integer`) and `name` (of type `string`) where `id` is required to be present.

EXAMPLE 24

```
{
    "type": "object",
    "properties": {
        "id": {
            "type": "integer"
        },
        "name": {
            "type": "string"
        }
    },
    "required": ["id"]
}
```

When the `object` is anonymous (i.e. it is the root, or participates in an `array` definition), the above `object` definition transforms to the following XML Schema element definition.

EXAMPLE 25

```
<xs:element name="object" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType>
        <xs:all>
            <xs:element name="id">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="integer" type="xs:integer" />
                  </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="name" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="string" type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:all>
    </xs:complexType>
</xs:element>
```

EDITOR'S NOTE

JSON schema `object` does NOT define any order. Therefore, in order to capture the constraints of JSON schema `object`, we need to use xsd:all constructs instead of xsd:sequence.

Otherwise (i.e. the `object` is a member of another `object` definition, thus has a name), the `object` definition transforms to the following XML schema element definition. Note *__name* should be replaced by the actual name of the `object`.

EXAMPLE 26

```
<xs:element name="__name" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="object">
                <xs:complexType>
                    <xs:all>
                        <xs:element name="id">
                            <xs:complexType>
                              <xs:sequence>
                                <xs:element name="integer" type="xs:i
nteger" />
                              </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                        <xs:element name="name" minOccurs="0">
                            <xs:complexType>
                              <xs:sequence>
                                <xs:element name="string" type="xs:str
ing" />
                              </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                    </xs:all>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

3.2.4.3.2 JSON Array Definition to XML Schema

Shown below is the JSON schema `array` definition used as the `inputData` in Section JSON Array. The `array` consists of exactly three number literals with each value within the value range of [0 ... 255].

EXAMPLE 27

```
{
    "type": "array",
    "items": {
        "type" : "number"
        "minimum": 0,
        "maximum": 255,
    },
    "minItems" : 3,
    "maxItems" : 3
}
```

When the `array` is anonymous (i.e. it is the root, or participates in another`array` definition), the above `array` definition transforms to the following XML Schema element definition.

EXAMPLE 28

```
<xs:element name="array" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="double" minOccurs="3" maxOccurs="3">
                <xs:simpleType name="minInclusive">
                    <xs:restriction base="xs:double">
                        <xs:minInclusive value="0"/>
                        <xs:maxInclusive value="255"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Otherwise (i.e. the `array` is a member of an `object` definition, thus has a name), the `array` definition transforms to the following XML schema element definition. Note__*name* should be replaced by the actual name of the `array`.

EXAMPLE 29

```
<xs:element name="__name" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="array">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="double" minOccurs="3" maxOcc
urs="3" >
                            <xs:simpleType name="minInclusive">
                                <xs:restriction base="xs:double">
                                    <xs:minInclusive value="0"/>
                                    <xs:maxInclusive value="255"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

3.2.5 TD Examples

In the following, we give three examples of a full TD. The first one, a temperature sensor, shows how to define relations between Thing Properties. As a second example, we modeled an actuator (LED lamp) where Actions have all been characterized semantically, so that machines could unambiguously interpret them in an automated manner. At last, association between Things is shown by defining a master switch controlling other LED lamps.

3.2.5.1 Temperature Sensor

EXAMPLE 30

```
{
  "@context": ["https://w3c.github.io/wot/w3c-wot-td-context.jsonld",
               "https://w3c.github.io/wot/w3c-wot-common-context.jsonlc
],
  "@type": "Sensor",
  "name": "myTempSensor",
 "base" : "coap:///www.example.com:5683/temp/",
  "interaction": [
    {
      "@id": "val",
      "@type": ["Property","Temperature"],
      "unit": "celsius",
      "reference": "threshold",
      "name": "myTemp",
      "outputData": { "type": "number" },
      "writable": false,
      "link": [{
       "href" : "val",
       "mediaType": "application/json"
      }]
    }, {
      "@id": "threshold",
      "@type": ["Property","Temperature"],
      "unit": "celsius",
      "name": "myThreshold",
      "outputData": { "type": "number" },
      "writable": true,
      "link": [{
       "href" : "thr",
       "mediaType": "application/json"
      }]
    },
    {
      "@type": ["Event"],
      "outputData": { "type": "number" },
      "name": "myChange",
      "property": "temp",
      "link": [{
       "href" : "val/changed",
       "mediaType": "application/json"
      }]
    }, {
      "@type": ["Event"],
      "outputData":  { "type": "number" },
      "name": "myWarning",
      "link": [{
       "href" : "val/high",
       "mediaType": "application/json"
      }]
    }
  ]
}
```

The Thing `myTempSensor` defines two Properties: `myTemp` and `myThreshold`. Both are defined as temperatures, with the same unit (`celsius`). A client that is able to parse a TD only needs to know the predicate `reference` to understand that the threshold acts as a reference value for `myTemp` while the latter is the actual value measured by the temperature sensor. Here, `reference` points to `http://schema.org/valueReference`.

Moreover, one of the Events of the Thing is linked to the measured value (with the predicate `property`). It means in that context that an event should be triggered each time `myTemp` changes. The other Event does not define further semantics, it could be used either in a closed system (where clients are aware of its meaning) or by a human but an external agent would not have sufficient information to interpret it.

3.2.5.2 LED Master Switch

EXAMPLE 31

```
{
  "@context": ["https://w3c.github.io/wot/w3c-wot-td-context.jsonld",
               "https://w3c.github.io/wot/w3c-wot-common-context.jsonlc
],
  "@type": "Lamp",
  "name": "myMasterLED",
  "interaction": [
    {
      "@type": ["Actions", "Toggle"],
      "name": "myMasterOnOff",
      "inputData": { "type": "boolean" },
      "link": [
      {
        "href" : "coap://www.example.com:5683/master",
        "mediaType": "application/json"
      },{
        "href" : "http://www.example.com:80/master",
        "mediaType": "application/json"
      }]
    }
  ],
  "associations": [
    { "href": "coap://www.example.com:5683/0" },
    { "href": "coap://www.example.com:5683/1" },
    { "href": "coap://www.example.com:5683/2" },
    { "href": "coap://www.example.com:5683/3" },
    { "href": "coap://www.example.com:5683/4" },
    { "href": "coap://www.example.com:5683/5" },
    { "href": "coap://www.example.com:5683/6" },
    { "href": "coap://www.example.com:5683/7" }
  ]
}
```

In this last example, we illustrate the use of `associations`. The Thing we modeled here acts as a master switch for eight lamps similar to that of . It means switching on and off `myMasterOnOff` will propagate to all associated Things by toggling their Action that is also of type `Toggle`.

No precise semantics for `associations` have been defined yet and there might exist many other kinds of dependency between Things than simply parent/child relation. This issue will be addressed soon. Until then, Thing associations could be useful for discovery.

3.2.6 Usage

3.2.6.1 Extending Thing Description with Other Semantic Models

As the TD context we have developed is intended to be minimal, it is strongly recommended to extend it for each Thing by reusing other vocabularies or ontologies and/or defining application-specific terms. In the following example, in addition to our standard context, the Thing Description declares a namespace for the Smart Appliance Reference Ontology (SAREF):

EXAMPLE 32

```
{
  "@context": ["http://w3c.github.io/wot/w3c-wot-td-context.jsonld",
               {"saref": "http://uri.etsi.org/m2m/saref#"}],
  "@type": [ "Thing", "saref:BuildingSpace" ],
                  ...
}
```

This way, the TD can refer to the concept of building space, as defined by SAREF, with the CURIE `saref:BuildingSpace`. Formally, a building space is something that has geo-coordinates and that contains building objets (like doors or windows) and devices (like sensors) [smartM2M]. The TD states that the Thing it models is also a building space. One can assume this thing exposes properties such as room temperature, brightness, number of persons, ...

During early experimentations with TD files, a few concepts and terms appeared to be recurrent, such as `Temperature`, `Toggle` or `unit`. To ease experimentation with the modeling of Things, a context that gathers such common terms has been defined at `http://w3c.github.io/wot/w3c-wot-common-context.jsonld`. Its content simply defines aliases for some SAREF classes and properties. This file is not intended to become a reference. It should always be used along with the standard TD context, as follows:

EXAMPLE 33

```
{
  "@context": ["http://w3c.github.io/wot/w3c-wot-td-context.jsonld",
               "http://w3c.github.io/wot/w3c-wot-common-context.jsonld"
],
  "@type": [ "Thing", "BuildingSpace" ],
                    ...
}
```

The table below shows SAREF concepts that can be used along with either `Thing`, `Property` or `Action` as `@type` annotations, like in the examples above. The common context contains mapping for both terms with and without prefix, for the convenience of the user. This means that e.g. `BuildingSpace` and `saref:BuildingSpace` map to the same SAREF concept.

Thing	Property	Action
`BuildingSpace`	`Temperature`	`ToggleCommand`
`Door`	`Humidity`	`StartCommand`
`Window`	`Energy`	`StopCommand`
`Switch`	`Light`	`StepUpCommand`
`Sensor`	`Motion`	`StepDownCommand`
`Meter`	`Occupancy`	`SetLevelCommand`
	`Power`	`OnCommand`
	`Pressure`	`OffCommand`
	`MultiLevelState`	`OpenCommand`
	`OnOffState`	`CloseCommand`
	`OpenCloseState`	
	`StartStopState`	

Moreover, our common context introduces a new term, `actsUpon`, to express on which property an action is acting, if both are modeled in the same Thing Description. For instance, a thermostat that can regulate the temperature of a room can be modeled as follows (note the use of the additional JSON-LD keyword `@type`):

EXAMPLE 34

```
{
  "@context": ["http://w3c.github.io/wot/w3c-wot-td-context.jsonld",
               "http://w3c.github.io/wot/w3c-wot-common-context.jsonld"
],
  "@type": [ "Thing" ],
  "interaction": [
                  {
                    "@id": "_:temp",
                    "@type": ["Property", "Temperature"],
                    "outputData":  { "type": "number" },
                    ...
                  }, {
                    "@type": ["Action", "SetLevelCommand"],
                    "actsUpon": "_:temp",
                    "inputData": { "type": "number" },
                    ...
                  }
  ]
}
```

In the following, we give examples of modeling for some devices that appeared in past PlugFests (the full Thing Descriptions can be found on Github):

Device	Semantic Modeling
LED lamp	`Switch` that has two actions: a `OnCommand` and a `OffCommand`.
LED lamp (with dimming)	`Switch` that has one writable `Light` property (that is also of type `MultiLevelState`) and two actions: a `StepUpCommand` and a `StepDownCommand`, where each `actsUpon` the `Light` property.
Brightness sensor	`BuildingSpace` that has one `Light` property.
Human detection sensor	`BuildingSpace` that has one `Occupancy` property.
Home Air Conditioner	`BuildingSpace` that has one writable `Temperature` property (that is also of type `MultiLevelState`) and two actions: a `StepUpCommand` and a `StepDownCommand`, where each `actsUpon` the `Temperature` property.
Weather station	`geo:SpatialThing`* that has four readable properties: `Temperature`, `Humidity`, `Pressure` and `Light`.

(*) the prefix `geo` is introduced here. It also includes terms to specify the geo-location of a spatial entity: `geo:long` and `geo:lat`, also used in the examples on Github.

Beside SAREF, many vocabularies are of interest for WoT, such as W3C SSN [vocab-ssn-20170504], oneM2M Base Ontology, iot.schema.org and vocabularies developed in the context of research projects such as the Vicinity core ontology, BIG IoT's domain models or the SEAS ontology.

3.2.6.2 Discovery

EDITOR'S NOTE

This section describes in general how a Thing (i.e., its TD) can be discovered. In particular, the current practices at the PlugFests should become clear. Technology-specific mechanisms such as BLE Beacons or UPnP multicast requests should go into the corresponding sub-sections of 3.1.2 Protocol Bindings.

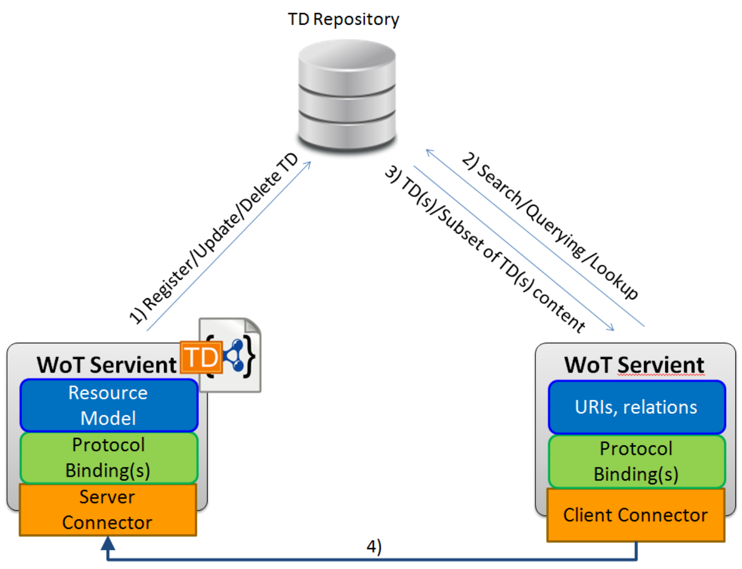
Discovering a Thing means acquiring a link to its TD, which then contains all the information to interact with it and understand its data. The URI of the link may point to the Thing (technically the servient) itself, as Things often host their TD directly, or to any other location on the Web. There are several approaches to aquire such links. Some work independent from the Protocol Binding, others rely on features of a specific protocol.

3.2.6.2.1 Manual Discovery

The link to the TD is provided by the developer at programming time, the operator through device management, or the user through a UI.

3.2.6.2.2 Repository

The Thing (or a commissioning tool) registers the TD with a well-known repository, which also provides a look-up mechanism (potentially supporting filtering). An implementation of the repository is available here (if the server is not online, you can also run your own instance: see our Github repository).



The figure shows the basic concept of the usage of a repository: 1) A servient registers its TD to a known TD repository. 2) Another servient does a lookup/search/querying (e.g., via SPARQL) at the TD repository. 3) The repository answers with one ore more relevant TDs. Also a subset of TD's content can be in the result set (e.g., only relevant properties, actions, and/or events are included). 4) Based on the repository's result set, a connection can be initiated.

3.2.6.2.3 Local Discovery

The Thing is able to broadcast a discovery request locally (e.g., a CoAP multicast request for `/.well-known/core`) or to receive announcements from its proximity (e.g., BLE Beacons). The response to a discovery request can include the TD directly or just a link. For announcements, it is more common to only provide a link, since broadcasting TDs can be expensive.

3.2.6.3 Security

@TODO Add howto about security metadata and how to use it
(...)

4. Security Considerations

This section outlines the conception behind the WoT security model. The following security concerns apply:

- Enforcing security at the network interface of the Thing
- Protecting TD objects

TD objects in plain form can easily be manipulated or faked by attackers. This could result in security or safety breaches. To establish the authenticity of TD objects signature mechanisms are needed. They allow to corroborate the source of the TD information and to assure the integrity of its contents.

- Describing prerequistes for accessing Things
- Components that expose Things (aka servients) may expect callers to present credentials and/or to use secure communications in order to access them. Specific TD object contents are needed to express such requirements.

These two security concerns are orthogonal: signing TD objects may be needed for TD objects which do not express any security-related expectation for accessing Things. On the other hand TD objects may express such expectations without being signed.

4.1 Security Enforcement at the WoT Interface

The purpose of WoT is to expose private resources (Things themselves, information they produce, or information about them) at public-facing endpoints (e.g., Internet). This implies that components that expose Things (i.e., servients) must enforce security: credentials have to be presented, communication exchanges have to be encrypted/signed etc. There are two basic clusters of security features which are relevant for servients:

- Authorization and authentication

  When receiving instructions, servients must authorize such calls/requests i.e. determine whether to accept them for processing. This implies the need to authenticate the properties of callers that are used when performing authorization.
- Secure communications

  When exchanging sensitive resources/information over a shared network, servients must demand encrypting/signing corresponding message exchanges.

These fundamental security services depend on underpinnings esp. the provisioning of metadata (identifiers, attributes, assignments/affiliations) about system actors and the establishment of keying relations resp. shared secrets among actors that are supposed to interact in protected fashion. This section does not further elaborate on the provisioning of metadata and establishment of keys/secrets needed to implement authoriztion and authentication resp. secure communications.

Authorization and Authentication

The authorization of requested actions (instructions/requests) depends on the authentication of those properties of a caller that are relevant for authorization and that are being submitted or claimed by the caller. This can e.g. be identifiers, attributes (e.g. location), assignments (e.g. roles), affiliations (e.g. group memberships) or permissions (e.g. resource access rights). There are two basic strategies for the initial authentication of callers:

- Internalization

  The servient challenges the caller for initial authentication credentials, validates them, determines the properties of the authenticated caller, uses them to render an authorization decision and enforces this decision.
- Externalization to online TTPs

  The servient delegates the challenging for initial authentication credentials, their validation, the determination of caller properties and evtl. the rendering of an authorization decision to an online TTP. In course of that the servient then receives a report of the TTP (corresponding objects are called 'security token' or 'access token') and proceeds with it

The internalization approach has a number of issues including: lack of SSO (a concern when the caller is a user agent, no real issue otherwise), tight coupling of initial authentication credentials and application protocol (a painpoint when multiple schemes/dynamics are to be covered), forcing servients into the implementation of complex, non-core functionality. For these reasons the internalization strategy is uncommon - even in unconstrained office/enterprise/Cloud IT. Since WoT is concerned with components that are (severly) constrained, it is anticipated that WoT solutions will externalize complex processing tasks in caller authentication to online TTP components by default.
The externalization of initial caller authentication to TTPs allows to allocate the authorization decision making as follows:

- Internalization

  The servient component performs authorization decision making and enforcement
- Externalization according push

  The servient component performs authorization decision enforcement. Authorization decision making is done by the TTP that performs the initial authentication of callers (resource access control information travels in-band with the security token asserting the authenticated identity of the caller)
- Externalization according pull

  The servient component performs authorization decision enforcement. Authorization decision making is done by a backend component (called by the servient).

The externalization of initial caller authentication and the externalization of the authorization decision making according push to TTPs matches the architectural proposition of the IETF ACE working group (see "An architecture for authorization in constrained environments" at https://www.ietf.org/id/draft-ietf-ace-actors).

The anticipated default strategy in WoT results in online TTP components as well as special-purpose objects: security tokens by which online TTPs report the results of their work back to servients. These are cryptographically protected objects that report properties of the caller (identifiers, attributes, assignments, affiliations, permissions etc) which are asserted by the online TTP. Such objects may contain resource access control information. Such objects are short-lived (normally: minutes/hours) and may be re-used during their lifetime. These objects must be signed to prevent counterfeiting and may be encrypted to preserve confidentiality.

- Bearer security model

  In case of bearer tokens submitters of security tokens do not have to provide any proof-of-possession. A real-life analogy is a cinema or concert ticket - the recipient only wants to make sure it is no counterfeit. This implies a risk that illegitimate callers may present valid security tokens. In the digital World SAML Web SSO is a prominent example of an online TTP scheme that is implementing a bearer model.
- PoP security model

  In case of PoP (aka HoK) tokens submitters are required to supply fresh authenticator objects in addition/conjunction with security tokens. A real-life analogy is a passport - the recipient wants to make sure it is no counterfeit and is presented by the right person. In the digital World Kerberos is the most prominent example of an online TTP scheme that is implementing a PoP model.

Note that OAuth started with PoP (OAuth 1.x), moved to bearer (OAuth 2.0 according RFC 6749/6750) to foster adaptation and currently adds PoP (current OAuth 2.0 drafts) to include high-value assets.

In WoT no one-size-fits-all security token should be expected. Specific WoT domains such as building automation, sports/health care, industrial control systems have their own specific needs when it comes to the expression of caller properties. Moreover specific protocol stacks also have their own specific constraints that affect security token contents.

In addition to security tokens (that are domain and protocol stack-specific) protocols are needed to acquire and supply security tokens. The security token supply between callers and servients is the straight-forward part of this task:

- Application requests

The application protocol requests must allow the transfer of security tokens (bearer) and/or authenticators (PoP) in a standardized way. For HTTP such standards exist (RFC 6750 for bearer tokens) resp. emerge (PoP tokens); for CoAP they do emerge. A critical issue are large security tokens/authenticators that can not be supplied inline with arbitrary application requests. The common trick to address this concern is to introduce a dedicated endpoint to which security tokens and/or authenticators can be sent in application PDUs (dedicated to the supply of security tokens and/or authenticators) during in an upfront exchange.

- Application responses

  The application protocol (error) responses must allow to inform clients about expected security token issuing authority, expected security token type/category and protection. Current and emerging standards do not yet provide an adequate coverage for this concern.

The security token acquisition is the more complex part of this trick. The servient components trigger exchanges between callers and their online TTP but are not involved during their execution. The requesting party may also introduce further online TTP components to support e.g. constrained callers or clients. Obviously the online TTPs need to respect the capabilities of the actual caller or its proxy. In addition to that a continental divide exists between following scenarios:

- Consumer goods, owned by individual end users

  Authorization is often conducted according a lazy policing model: access requests happen first, then the resource owner (an individual) is asked for an authorization (whether she can be assumed to be online does matter). This results in an authorization policy that is build-up dynamically
- Capital goods, owned by legal entities

  Authorization is mostly conducted according a preemtive policing model: a (static) authorization policy is provided, then access request happen and authorization decisions are rendered without assuming any interaction with the resource ownwer or a representative

Both cases distribute work unevenly over servients and their callers; both sides have fundamentally different working tasks and workloads in authorization and authentication:

- Servient tasks

  Understand protection needs (public, private) of served resources, challenge for security tokens (bearer) and/or authenicators (PoP), validate security tokens (bearer) and/or authenicators (PoP) and match their content against application request contents
- Caller tasks

React on application protocol error responses (demanding the supply of security tokens), acquire security token from online TTPs, perform initial authentication by means of a security protocol (possibly different from the application protocol, evtl. mediated by an online TTP on side of the requesting party), and supply security tokens (bearer) and/or authenticators (PoP) as part of the application protocol

For client-side and server-side support APIs this obvious results in fundamentally different security task that have to be facilitated by WoT Interfaces. The security-part of the W3C WoT PlugFest in Nice showed the ability to create mutually interoperable and security-enabled WoT component implementations (by different vendors). The security-enabling at the PlugFest focussed on the authorization of actions (sent to servients) and the authenticator of actors (WoT clients). It utilized online third-party components to which the servients and clients delegated the complex processing tasks in authorization and authentication and utilized trusted assertions (aka access/security tokens) to report back to servients and clients. This follows architetural models and protocols which emerge from IETF working groups (including OAuth and ACE). See https://www.w3.org/WoT/IG/wiki/F2F_meeting_2016,_January,_26th_%E2%80%93_28th,_France,_Nice#Security for more info about the security-enabling and its results for this PlugFest

Secure Communications

Communication security can be implemented in form of transient, transport-level security (e.g. TLS, DTLS) and/or persistent, application-level security (e.g. JOSE, COSE). Both approaches result in symmetry with respect to the work-split across servients and their callers; both sides basically have the same working tasks and workload

Transport-level security has a long heritage in IT and presents a well-understood means in Web security which is carrying critical use cases. If TLS resp. DTLS match the constraints of WoT deployments they should be used to secure communications.

Some WoT deployments have requirements that do not match TLS and DTLS properties, for instance multicast resp. group-oriented communications. It is also possible that (severely) constrained WoT components can not bear the overhead that is implied by the TLS and DTLS protocols. Then persistent, application-level security can help to achieve communication security.
In contrast to transport-level security, application-level security allows to apply cryptographic transformations in a specific/granular manner (affecting some but not all exchanged data objects) and thereby reducing the security processing burden. In case security tokens are to be used in cases where transport-level security can not be used, application-level security offers means to provide PoP for security tokens as well as authenticity for message exchanges.

4.2 Describing Prerequistes for Accessing Things

When private resources (Things themselves, information they produce or information about them) get exposed at public-facing endpoints (e.g. Internet) then the components that are exposing theThings (i.e., servients) must enforce security: credentials have to be presented, communication exchanges have to be encrypted/signed etc. There are two basic strategies for demanding such security mechanisms:

- A priori

  Callers know beforehand and make requests according the expectations of the callee. A prominent example for this strategy is the 'https' access scheme in URLs. It triggers clients to employ SSL/TLS in order to send HTTP requests (HTTP-over-TLS, RFC 2818).
- A posteriori

  Callers do not know beforehand, make a request and are being told by the callee in the response or during the exchanges. An example for this strategy is the negotiation of the use of SSL/TLS as part of HTTP exchanges (TLS-in-HTTP, RFC 2817).

The a posteriori-strategy places requirements on the application protocol esp. the contents of its error responses, not TD. In case of WoT, the a priori-strategy places requirements on TD: TD needs to be able to express security mechanisms that callers have to fulfill. This expression shall be optional because certain WoT deployments might prefer the a posteriori-strategy and do (intentionally) not want to reveal this information in TD objects.

Authorization and Authentication

This section assumes that servients externalize complex processing tasks around call authorization and caller authentication to security components (representing online TTPs). See below for a rationale.
Following items are needed to inform callees about security credentials/tokens that they have to present for specific resources:

- Issuing authority of the security token

  Callers need to know about the online TTP component where they need to apply for security tokens. This includes information about their configuration encompassing endpoints and supported protocols.
- Type/category of security token

  Callers need to know which type(s) or categories of security tokens they need to apply for. An individual security token type or category is assumed to expressed by a URN (in the namespace of the security token issuer).

- Protection model for security token

  Callers need to know the protection model of the security tokens they need to apply for. Note that bearer token are submitted in a opaque fashion, for PoP tokens an authenticator has to be created by the caller.

Secure Communications

Transient Protection, Transport-Level Security

The URL access scheme allows to express the need for secure communications (SSL/TLS or DTLS). This is part of the resource endpoint URL and there is no need for TD to specify any additional means. Note that 'http' vs. 'https' resp. 'coap' vs. 'coaps' carries only boolean information. The suite of parameters that determine SSL/TLS or DTLS sessions is much larger (encompassing authentication modes and related information, encryption/signature strategies). Hence the URL access scheme does not provide a full or even rich announcement of the required settings. Since it is the best current practice in IT to rely on a simple a priori trigger (in URL access scheme) and do the rest of the work accordng the a posteriori-apporach (inband with the security protocol) there is no need for TD resp. WoT to go beyond.

Persited Protection, Application-Level Security

Application-level security providing persisted protection is usually handled by specifications/conventions in the application domain. This uses specific media types such as application/json for JSON-plain and application/jose for protected JSON objects (JWS/JWE). Note that application/cbor does exist but there is not yet a IANA registration for "application/cose" (as of 2016-02-21, see http://www.iana.org/assignments/media-types).

TODO (@TF TD): the datatype part in XML Schema (https://www.w3.org/TR/xmlschema11-2/) seems to be short with respect to being able to speak about cryptographically transformed data (see RFC 7193 for e.g. application/cms). It seems hard to impossible to allow callees to express security requirements wrt to persitent application-level security based on that. I suggest to either disclaim this case or allow richer data types

4.3 Protecting TD Objects

TD objects may be protected by means of signature. Signing TD objects is optional.

If TD objects are signed then cryptographic checksums (aka signatures) are added to establish the authenticity of TD objects. Such checksums are created by the producers/issuers of TD objects and validated by consumers of TD objects (which should reject signed TD objects whose signatures are invalid). They use keying associations between the producer of the protected TD objects and its consumer(s). The checksum resp. keying association can be asymmetric (producer signs with a private key, consumers validate with the corresponding public key) or symmetric (producer and consumer use a shared secret key).
For TD objects expressed in JSON, IETF JOSE (JSON Object Signature and Encryption, see https://datatracker.ietf.org/wg/jose/documents/) provides the standards for computation and validation of signatures and their representation as network transfer objects (RFC 7515).

Signing is straight-forward on the level of cryptographic primitives i.e. algorithms to generate and validate checksums. But the cryptographic algorithm that computes the checksum only reduces the amount of data that has to be protected - from potentially large (TD) to small (key) objects. Sound key management practices are needed behind the scene. Key management largely contributes to the overall price-tag of the security solution. It is also not straight-forward on the level of cryptographic objects i.e. the expression and organization of signed data and signature metadata (information about the embedding/location and transformation/normalization of signed data, information about the signature algorithm as well as the keying association etc). The solution design needs to reflect given or anticipated WoT system dynamics with respect to granularity/pooling of TD object signing. This presents a not yet addressed problem.

On top of a commonly signature object specification such as XML Signature or CMS lots of profiling may be needed in order to make a signature mechanism meaningful for a specific domain. This is illustrated by the IETF ltans effort: IETF ltans addressed the long-term archival of digital data objects and produced a number of RFCs (RFC 4810/4998/5276/5698/6238) on top of XML Signature resp. CMS that were needed to do the trick. Another example is XAdES (https://www.w3.org/TR/XAdES/). Note: this does not try to suggest that the signing of TD objects will have the same added complexity as IETF ltans or W3C XAdES, only that specific conventions/profiles for the use of JWS or XML Signature in the TD domain will be needed.

C. References

C.1 Normative references

[JSON-LD]
JSON-LD 1.0. Manu Sporny; Gregg Kellogg; Markus Lanthaler. W3C. 16 January 2014. W3C Recommendation. URL: https://www.w3.org/TR/json-ld/

[RFC3986]
Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee; R. Fielding; L. Masinter. IETF. January 2005. Internet Standard. URL: https://tools.ietf.org/html/rfc3986

[RFC6750]
The OAuth 2.0 Authorization Framework: Bearer Token Usage. M. Jones; D. Hardt. IETF. October 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6750

[RFC7252]
The Constrained Application Protocol (CoAP). Z. Shelby; K. Hartke; C. Bormann. IETF. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7252

[RFC7517]
JSON Web Key (JWK). M. Jones. IETF. May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7517

[RFC7519]
JSON Web Token (JWT). M. Jones; J. Bradley; N. Sakimura. IETF. May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7519

[RFC7591]
OAuth 2.0 Dynamic Client Registration Protocol. J. Richer, Ed.; M. Jones; J. Bradley; M. Machulak; P. Hunt. IETF. July 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7591

[draft-fge-json-schema-validation-00]
JSON Schema: interactive and non interactive validation. Kris Zyp; Gary Court. IETF. Internet-Draft. URL: https://tools.ietf.org/html/draft-fge-json-schema-validation-00

[draft-zyp-json-schema-04]
JSON Schema: core definitions and terminology. Kris Zyp; Gary Court. IETF. Internet-Draft. URL: https://tools.ietf.org/html/draft-zyp-json-schema-04

[exi-for-json]
EXI for JSON (EXI4JSON). Daniel Peintner; Don Brutzman. W3C. 23 August 2016. W3C Working Draft. URL: https://www.w3.org/TR/exi-for-json/

[rdf11-concepts]
RDF 1.1 Concepts and Abstract Syntax. Richard Cyganiak; David Wood; Markus Lanthaler. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf11-concepts/

[smartM2M]
ETSI TS 103 264: SmartM2M; Smart Appliances; Reference Ontology and oneM2M Mapping. European Telecommunications Standards Institute. November 2015. ETSI Technical Specification. URL:

http://www.etsi.org/deliver/etsi_ts/103200_103299/103264/01.01.01_60/ts_103264v010101p.pdf

C.2 Informative references

[vocab-ssn-20170504]
Semantic Sensor Network Ontology. Armin Haller; Krzysztof Janowicz; Danh Le Phuoc; Kerry Taylor; Maxime Lefrançois. W3C. 4 May 2017. W3C Working Draft. URL: https://www.w3.org/TR/2017/WD-vocab-ssn-20170504/

↑ ↑

→