Web of Things (WoT) Thing Description

W3C Editor's Draft 10 June 2020

This version:
https://w3c.github.io/wot-thing-description/

Latest published version:
https://www.w3.org/TR/wot-thing-description/

Latest editor's draft:
https://w3c.github.io/wot-thing-description/

Implementation report:
https://w3c.github.io/wot-thing-description/testing/report.html

Editors:
Sebastian Kaebisch (Siemens AG)
Takuki Kamiya (Fujitsu Laboratories of America)
Michael McCool (Intel)
Victor Charpenay (Siemens AG)
Matthias Kovatsch (Huawei)

Participate:
GitHub w3c/wot-thing-description
File a bug
Commit history
Pull requests

Contributors:
In the GitHub repository

Repository:
We are on GitHub
File a bug

Abstract

This document describes a formal model and a common representation for a Web of Things (WoT) Thing Description. A Thing Description describes the metadata and interfaces of Things, where a Thing is an abstraction of a physical or virtual entity that provides interactions to and participates in the Web of Things. Thing Descriptions provide a set of interactions based on a small vocabulary that makes it possible both to integrate diverse devices and to allow diverse applications to interoperate. Thing Descriptions, by default, are encoded in a JSON format that also allows JSON-LD processing. The latter provides a powerful foundation to represent knowledge about Things in a machine-understandable way. A Thing Description instance can be hosted by the Thing itself or hosted externally when a Thing has resource restrictions (e.g., limited memory space) or when a Web of Things-compatible legacy device is retrofitted with a Thing Description.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document was published by the Web of Things Working Group as an Editor's Draft.

GitHub Issues are preferred for discussion of this specification. Alternatively, you can send comments to our mailing list. Please send them to public-wot-wg@w3.org (archives).

Please see the Working Group's implementation report.

Publication as an Editor's Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 March 2019 W3C Process Document.

Table of Contents

1. Introduction §

This section is non-normative.

The WoT Thing Description (TD) is a central building block in the W3C Web of Things (WoT) and can be considered as the entry point of a Thing (much like the *index.html* of a Web site). A TD instance has four main components: textual metadata about the Thing itself, a set of Interaction Affordances that indicate how the Thing can be used, schemas for the data exchanged with the Thing for machine-understandability, and, finally, Web links to express any formal or informal relation to other Things or documents on the Web.

The Interaction Model of W3C WoT defines three types of Interaction Affordances: Properties (`PropertyAffordance` class) can be used for sensing and controlling parameters, such as getting the current value or setting an operation state. Actions (`ActionAffordance` class) model invocation of physical (and hence time-consuming) processes, but can also be used to abstract RPC-like calls of existing platforms. Events (`EventAffordance` class) are used for the push model of communication where notifications, discrete events, or streams of values are sent asynchronously to the receiver. See [WOT-ARCHITECTURE] for details.

In general, the TD provides metadata for different Protocol Bindings identified by URI schemes [RFC3986] (e.g., `http`, `coap`, etc. [IANA-URI-SCHEMES]), content types based on media types [RFC2046] (e.g., `application/json`, `application/xml`, `application/cbor`, `application/exi`, etc. [IANA-MEDIA-TYPES]), and security mechanisms (for authentication, authorization, confidentiality, etc.). Serialization of TD instances is based on JSON [RFC8259], where JSON names refer to terms of the TD vocabulary, as defined in this specification document. In addition the JSON serialization of TDs follows the syntax of JSON-LD 1.1 [JSON-LD11] to enable extensions and rich semantic processing.

Example 1 shows a TD instance and illustrates the Interaction Model with Properties, Actions, and Events by describing a lamp Thing with the title *MyLampThing*.

EXAMPLE 1: Thing Description Sample

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "id": "urn:dev:ops:32473-WoTLamp-1234",
    "title": "MyLampThing",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in":"header"}
    },
    "security": ["basic_sc"],
    "properties": {
        "status" : {
            "type": "string",
            "forms": [{"href": "https://mylamp.example.com/status"}]
        }
    },
    "actions": {
        "toggle" : {
            "forms": [{"href": "https://mylamp.example.com/toggle"}]
        }
    },
    "events":{
        "overheating":{
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

From this TD example, we know there exists one Property affordance with the title *status*. In addition, information is provided to indicate that this Property is accessible via (the secure form of) the HTTP protocol with a GET method at the URI `https://mylamp.example.com/status` (announced within the `forms` structure by the `href` member), and will return a string-based status value. The use of the GET method is not stated explicitly, but is one of the default assumptions defined by this document.

In a similar manner, an Action affordance is specified to toggle the switch status using the POST method on the `https://mylamp.example.com/toggle` resource, where POST is again a default assumption for invoking Actions.

The Event affordance enables a mechanism for asynchronous messages to be sent by a Thing. Here, a subscription to be notified upon a possible overheating event of the lamp can be obtained by using HTTP with its long polling subprotocol on `https://mylamp.example.com/oh`.

This example also specifies the `basic` security scheme, requiring a username and password for access. Note that a security scheme is first given a name in `securityDefinitions` and then activated by specifying that name in a `security` section. In combination with the use of the HTTP protocol this example demonstrates the use of HTTP Basic Authentication. Specification of at least one security scheme at the top level is mandatory, and gives the default access requirements for every resource. However, security schemes can also be specified per-form, with configurations given at the form level overriding configurations given at the `Thing` level, allowing for the specification of fine-grained access control. It is also possible to use a special `nosec` security scheme to indicate that no access control mechanisms are used. Additional examples will be provided later.

The Thing Description offers the possibility to add contextual definitions in some namespace. This mechanism can be used to integrate additional semantics to the content of the Thing Description instance, provided that formal knowledge, e.g., logic rules for a specific domain of application, can be found under the given namespace. Contextual information can also help specify some configurations and behavior of the underlying communication protocols declared in the `forms` field. Example 2 extends the TD sample from Example 1 by introducing a second defintion in the `@context` to declare the prefix `saref` as referring to SAREF, the Smart Appliance Reference Ontology [SMARTM2M]. This IoT ontology includes terms interpreted as semantic labels that can be set as values of the `@type` field, giving the semantics of Things and their Interaction Affordances. In the example below, the Thing is labelled with `saref:LightSwitch`, the `status` Property is labelled with `saref:OnOffState` and the `toggle` Action with `saref:ToggleCommand`.

EXAMPLE 2: Thing Description with TD Context Extension for semantic annotations

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        { "saref": "https://w3id.org/saref#" }
    ],
    "id": "urn:dev:ops:32473-WoTLamp-1234",
    "title": "MyLampThing",
    "@type": "saref:LightSwitch",
    "securityDefinitions": {"basic_sc": {
        "scheme": "basic",
        "in": "header"
    }},
    "security": ["basic_sc"],
    "properties": {
        "status": {
            "@type": "saref:OnOffState",
            "type": "string",
            "forms": [{
                "href": "https://mylamp.example.com/status"
            }]
        }
    },
    "actions": {
        "toggle": {
            "@type": "saref:ToggleCommand",
            "forms": [{
                "href": "https://mylamp.example.com/toggle"
            }]
        }
    },
    "events": {
        "overheating": {
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh"
            }]
        }
    }
}
```

The declaration mechanism inside some `@context` is specified by JSON-LD. A TD instance complies to version 1.1 of that specification [json-ld11]. Hence, a TD instance can be also processed as an RDF document (for details about semantic processing, please refer to Appendix § D. JSON-LD Context Usage and the documentation under the namespace IRIs, e.g., https://www.w3.org/2019/wot/td).

2. Conformance §

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, *RECOMMENDED*, *SHOULD*, and *SHOULD NOT* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

A Thing Description instance complies with this specification if it follows the normative statements in § 5. TD Information Model and § 6. TD Representation Format regarding Thing Description serialization.

A JSON Schema [JSON-SCHEMA] to validate Thing Description instances is provided in Appendix § B. JSON Schema for TD Instance Validation.

3. Terminology §

This section is non-normative.

The fundamental WoT terminology such as ***Thing***, ***Consumer***, ***Thing Description*** (***TD***), ***Interaction Model***, ***Interaction Affordance***, ***Property***, ***Action***, ***Event***, ***Protocol Binding***, ***Servient***, ***WoT Interface***, ***WoT Runtime***, etc. is defined in Section 3 of the WoT Architecture specification [WOT-ARCHITECTURE].

In addition, this specification introduces the following definitions:

TD Context Extension
: A mechanism to extend Thing Descriptions with additional Vocabulary Terms. It is the basis for semantic annotations and extensions to core mechanisms such as Protocol Bindings, Security Schemes, and Data Schemas.

TD Information Model
: Set of Class definitions constructed from pre-defined Vocabularies on which constraints apply, thus defining the semantics of these Vocabularies. Class definitions are typically expressed in

terms of a Signature (a set of Vocabulary Terms) and functions over that Signature. The TD Information Model also includes Default Values, defined as a global function over Classes.

TD Processor
: A system that can serialize some internal representation of a Thing Description in a given format and/or deserialize it from that format. A TD Processor must detect semantically inconsistent Thing Descriptions, that is, Thing Descriptions that cannot satisfy constraints on the Instance Relation of the `Thing` class. For that purpose, a TD Processor may compute canonical forms of Thing Descriptions in which all possible Default Values are assigned. A TD Processor is typically a sub-system of a WoT Runtime. Implementations of a TD Processor may be a TD producer only (able to serialize to TD Documents) or a TD consumer only (able to deserialize from TD Documents).

TD Serialization* or *TD Document
: Textual or binary representation of Thing Descriptions that can be stored and exchanged between Servients. A TD Serialization follows a given representation format, identified by a media type when exchanged over the network. The default representation format for Thing Descriptions is JSON-based as defined by this specification.

Vocabulary
: A collection of Vocabulary Terms, identified by a namespace IRI.

Term* and *Vocabulary Term
: A character string. When a Term is part of a Vocabulary, i.e., prefixed by a namespace IRI, it is called a Vocabulary Term. For the sake of readability, Vocabulary Terms present in this document are always written in a compact form and not as full IRIs.

These definitions are further developed in § 5.2 Preliminaries.

4. Namespaces §

The version of the TD Information Model defined in § 5. TD Information Model of this specification is identified by the following IRI:

```
https://www.w3.org/2019/wot/td/v1
```

This IRI [RFC3987], which is also a URI [RFC3986], can be dereferenced to obtain a JSON-LD context file [json-ld11], allowing the compact strings in TD Documents to be expanded to full IRI-based Vocabulary Terms. However, this processing is only required when transforming JSON-based TD Documents to RDF, an optional feature of TD Processor implementations.

In the present specification, Vocabulary Terms are always presented in their compact form. Their expanded form can be accessed under the namespace IRI of the Vocabulary they belong to. These

namespaces follow the structure of § 5.3 Class Definitions. Each Vocabulary used in the TD Information Model has its own namespace IRI, as follows:

Vocabulary	*Namespace IRI*
Core	`https://www.w3.org/2019/wot/td#`
Data Schema	`https://www.w3.org/2019/wot/json-schema#`
Security	`https://www.w3.org/2019/wot/security#`
Hypermedia Controls	`https://www.w3.org/2019/wot/hypermedia#`

The Vocabularies are independent from each other. They may be reused and extended in other W3C specifications. Every breaking change in the design of a Vocabulary will require the assignment of a new year-based namespace URI. Note that to maintain the general coherence of the TD Information Model, the associated JSON-LD context file is versioned such that every version has its own URI (`v1`, `v1.1`, `v2`, ...) to also identify non-breaking changes, in particular the addition of new Terms.

Because a Vocabulary under some namespace IRI can only undergo non-breaking changes, its content can be safely cached or embedded in applications. One advantage of exposing relatively static content under a namespace IRI is to optimize payload sizes of messages exchanged between constrained devices. It also avoids any privacy leakage resulting from devices accessing publicly available vocabularies from private networks (see also § 9.1 Context Fetching Privacy Risk).

5. TD Information Model §

This section introduces the TD Information Model. The TD Information Model serves as the conceptual basis for the processing of Thing Descriptions and their serialization, which is described separately in § 6. TD Representation Format.

5.1 Overview §

The TD Information Model is built upon the following, independent Vocabularies:

- the *core* TD Vocabulary, which reflects the Interaction Model with the Properties, Actions, and Events Interaction Affordances [WOT-ARCHITECTURE]

- the *Data Schema* Vocabulary, including (a subset of) the terms defined by JSON Schema [JSON-SCHEMA]
- the *WoT Security* Vocabulary, identifying security mechanisms and requirements for their configuration
- the *Hypermedia Controls* Vocabulary, encoding the main principles of RESTful communication using Web links and forms

Each of these Vocabularies is essentially a set of Terms that can be used to build data structures, interpreted as objects in the traditional object-oriented sense. Objects are instances of classes and have properties. In the context of W3C WoT, they denote Things and their Interaction Affordances. A formal definition of objects is given in § 5.2 Preliminaries. The main elements of the TD Information Model are then presented in § 5.3 Class Definitions. Certain object properties may be omitted in a TD when Default Values exist. A list of defaults is given in § 5.4 Default Value Definitions.

The UML diagram shown next gives an overview of the TD Information Model. It represents all classes as tables and the associations that exist between classes, starting from the class `Thing`, as directed arrows. For the sake of readability, the diagram was split in four parts, one for each of the four base Vocabularies.

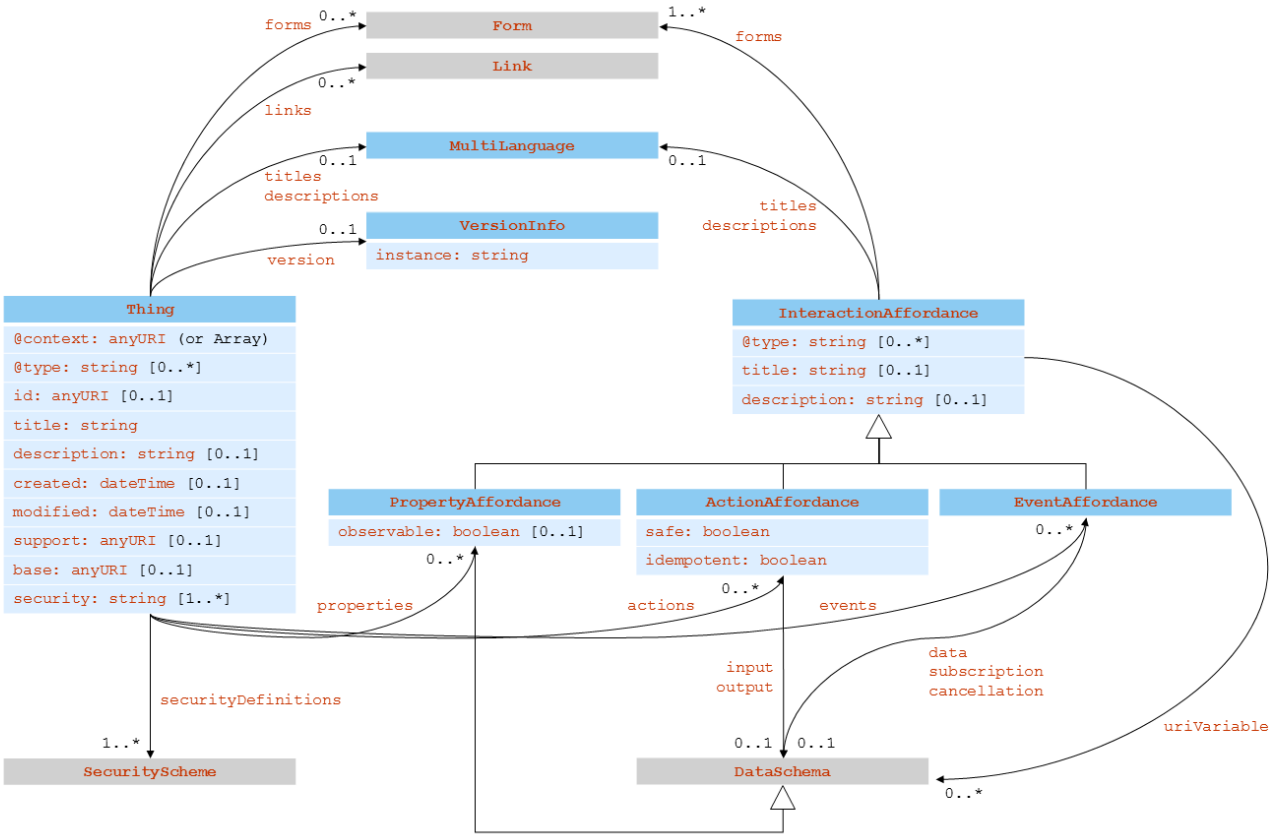

Figure 1 TD core vocabulary

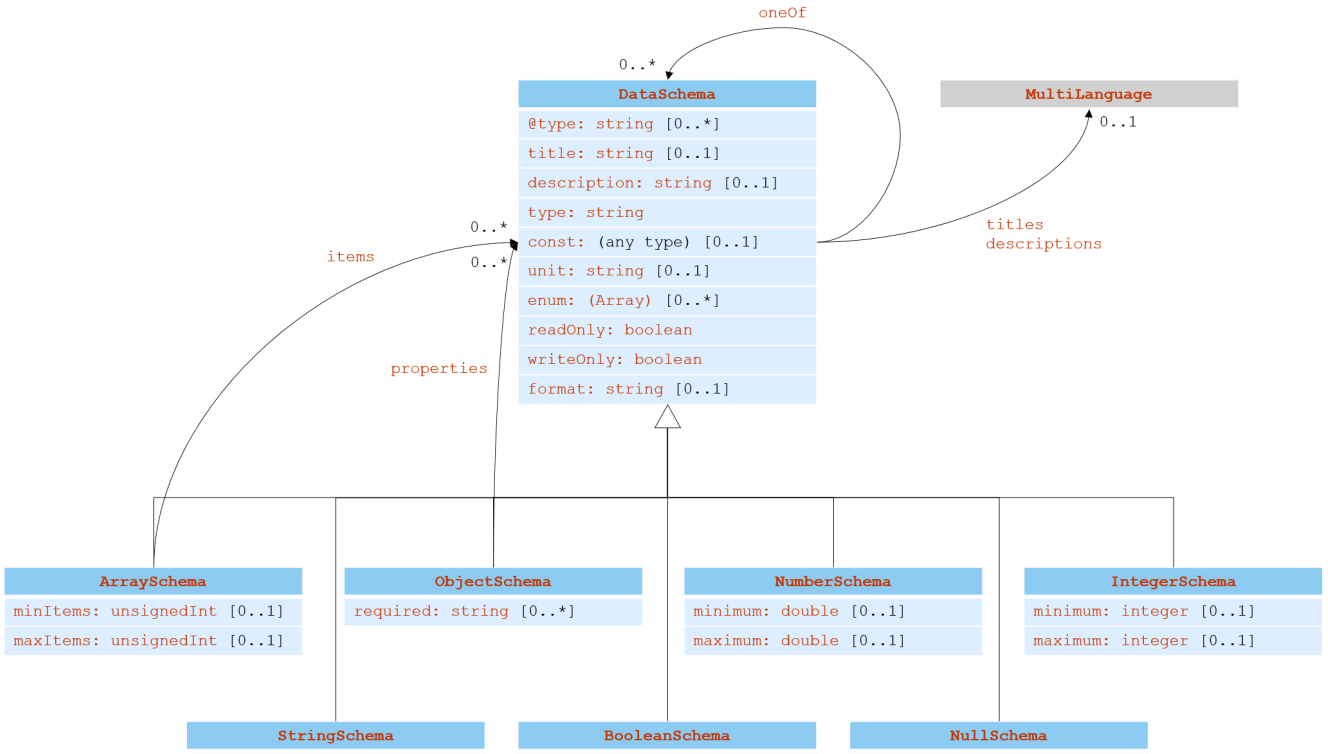

Figure 2 Data schema vocabulary

✧ ✧ ✧

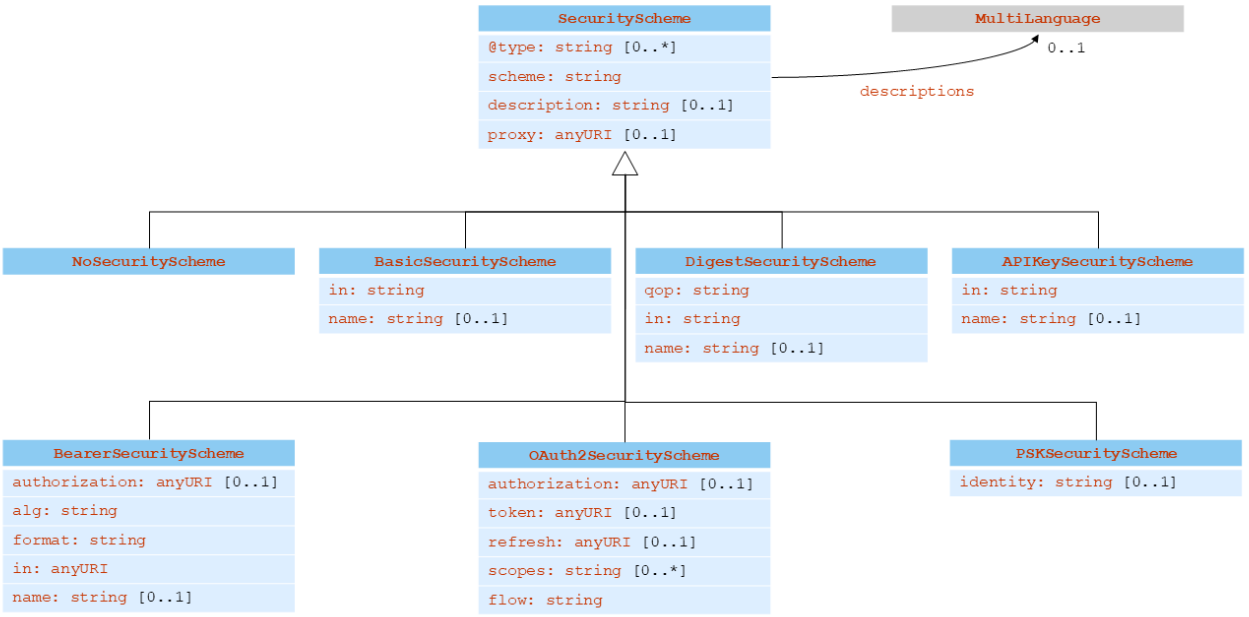

Figure 3 WoT security vocabulary

✧ ✧ ✧

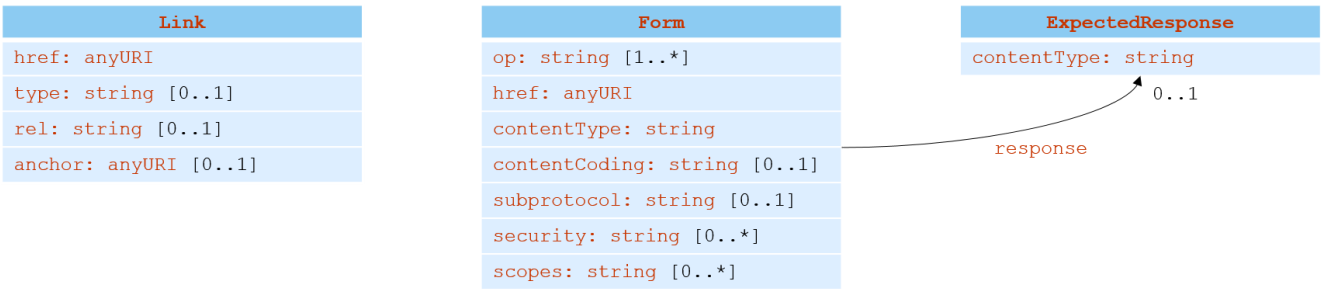

Figure 4 Hypermedia controls vocabulary

5.2 Preliminaries §

To provide a model that can be easily processed by both, simple rules on a tree-based document (i.e., raw JSON processing) and rich Semantic Web tooling (i.e., JSON-LD processing), this document defines the following formal preliminaries to construct the TD Information Model accordingly.

All definitions in this section refer to *sets*, which intuitively are collections of elements that can themselves be sets. All arbitrarily complex data structures can be defined in terms of sets. In particular, an ***Object*** is a data structure recursively defined as follows:

- a Term, which may or may not belong to a Vocabulary, is an Object.
- a set of name-value pairs where the name is a Term and the value is another Object, is also an Object.

Though this definition does not prevent Objects to include multiple name-value pairs with the same name, they are generally not considered in this specification. An Object whose elements only have numbers as names is called an ***Array***. Similarly, an Object whose elements only have Terms (that do not belong to any Vocabulary) as names is called a ***Map***. All names appearing in some name-value pair in a Map are assumed to be *unique* within the scope of the Map.

Moreover, Objects can be instances of some ***Class***. A Class, which is denoted by a Vocabulary Term, is first defined by a set of Vocabulary Terms called a ***Signature***. A Class whose Signature is empty is called a ***Simple Type***.

The Signature of a Class allows to construct two functions that further define Classes: an ***Assignment Function*** and a ***Type Function***. The Assignment Function of a Class takes a Vocabulary Term of the Class's Signature as input and returns either `true` or `false` as output. Intuitively, the Assignment Function indicates whether an element of the Signature is mandatory or optional when instantiating the Class. The Type Function of a Class also takes a Vocabulary Term of the Class's Signature as input and returns another Class as output. These functions are *partial*: their domain is limited to the Signature of the Class being defined.

On the basis of these two functions, an ***Instance Relation*** can be defined for a pair composed of an Object and a Class. This relation is defined as constraints to be satisfied. That is, an Object is an instance of a Class if the two following constraints are *both* satisfied:

- if for every Term for which the Assignment Function of the Class returns `true`, the Object includes a name-value pair with the Vocabulary Term as name.
- if for every Vocabulary Term in the Signature of the Class used as name in some name-value pair of the Object, the value of that pair is an instance of the Class returned by the Type Function of the Class for the given Vocabulary Term.

According to the definition above, an Object would be an instance of every Simple Type, regardless of its structure. Instead, another definition for the Instance Relation is introduced for Simple Types: an Object is an instance of a Simple Type if it is a Term with a given lexical form (e.g., `true`, `false` for the `boolean` type, `1`, `2`, `3`, ... for the `unsignedInt` type, etc.).

Moreover, additional Classes, called ***Parameterized Classes***, can be derived from the generic Map and Array structures. An Object is a Map of some Class, that is, an instance of the Map type *parameterized* with some Class, if it is a Map such that the value in all the name-value pairs it contains is an instance of this Class. The same applies to Arrays.

Finally, a Class is a ***Subclass*** of some other Class if every instance of the former is also an instance of the latter.

Given all definitions above, the TD Information Model is to be understood as a set of Class definitions, which include a Class name (a Vocabulary Term), a Signature (a set of Vocabulary Terms), an Assignment Function, and a Type Function. These Class definitions are provided as tables in § 5.3 Class Definitions. For each table, the values "mandatory" (respectively, "optional") in the assignment column indicates that the Assignment Function returns `true` (respectively, `false`) for the corresponding Vocabulary Term.

By convention, Simple Types are denoted by names starting with lowercase. The TD Information Model references the following Simple Types defined in XML Schema [XMLSCHEMA11-2-20120405]: `string`, `anyURI`, `dateTime`, `integer`, `unsignedInt`, `double`, and `boolean`. Their definition (i.e., the specification of their lexical form) is outside of the scope of the TD Information Model.

In addition, the TD Information Model defines a global function on pairs of Vocabulary Terms. The function takes a Class name and another Vocabulary Term as input and returns an Object. If the returned Object is different from `null`, it represents the ***Default Value*** for some assignment on the input Vocabulary Term in an instance of the input Class. This function allows to relax the constraint defined above on the Assignment Function: an Object is an instance of a Class if it includes all mandatory assignments *or* if Default Value exist for the missing assignments. All Default Values are given in the table of § 5.4 Default Value Definitions. In each table of § 5.3 Class Definitions, the assignment column contains the value "with default" if a Default Value is available for the corresponding combination of Class and Vocabulary Term in the TD Information Model.

The formalization introduced here does not consider the possible relation between Objects as abstract data structures and physical world objects such as Things. However, care was given to the possibility of re-interpreting all Vocabulary Terms involved in the TD Information Model as RDF resources, so as to integrate them in a larger model of the physical world (an ontology). For details about semantic

processing, please refer to § D. JSON-LD Context Usage and the documentation under the namespace IRIs, e.g., https://www.w3.org/2019/wot/td.

5.3 Class Definitions §

A TD Processor *MUST* satisfy the Class instantiation constraints on all Classes defined in § 5.3.1 Core Vocabulary Definitions, § 5.3.2 Data Schema Vocabulary Definitions, § 5.3.3 Security Vocabulary Definitions, and § 5.3.4 Hypermedia Controls Vocabulary Definitions.

5.3.1 Core Vocabulary Definitions §

5.3.1.1 `Thing` §

An abstraction of a physical or a virtual entity whose metadata and interfaces are described by a WoT Thing Description, whereas a virtual entity is the composition of one or more Things.

Vocabulary term	*Description*	*Assignment*	*Type*
`@context`	JSON-LD keyword to define short-hand names called terms that are used throughout a TD document.	mandatory	`anyURI` or Array
`@type`	JSON-LD keyword to label the object with semantic tags (or types).	optional	`string` or Array of `string`
`id`	Identifier of the Thing in form of a URI [RFC3986] (e.g., stable URI, temporary and mutable URI, URI with local IP address, URN, etc.).	optional	`anyURI`

Vocabulary term	*Description*	*Assignment*	*Type*
`title`	Provides a human-readable title (e.g., display a text for UI representation) based on a default language.	mandatory	`string`
`titles`	Provides multi-language human-readable titles (e.g., display a text for UI representation in different languages).	optional	`MultiLanguage`
`description`	Provides additional (human-readable) information based on a default language.	optional	`string`
`descriptions`	Can be used to support (human-readable) information in different languages.	optional	`MultiLanguage`
`version`	Provides version information.	optional	`VersionInfo`
`created`	Provides information when the TD instance was created.	optional	`dateTime`
`modified`	Provides information when the TD instance was last modified.	optional	`dateTime`
`support`	Provides information about the TD maintainer as URI scheme (e.g., `mailto` [RFC6068], `tel` [RFC3966], `https`).	optional	`anyURI`

Vocabulary term	*Description*	*Assignment*	*Type*
`base`	Define the base URI that is used for all relative URI references throughout a TD document. In TD instances, all relative URIs are resolved relative to the base URI using the algorithm defined in [RFC3986]. `base` does not affect the URIs used in `@context` and the IRIs used within Linked Data [LINKED-DATA] graphs that are relevant when semantic processing is applied to TD instances.	optional	`anyURI`
`properties`	All Property-based Interaction Affordances of the Thing.	optional	Map of `PropertyAffordance`
`actions`	All Action-based Interaction Affordances of the Thing.	optional	Map of `ActionAffordance`
`events`	All Event-based Interaction Affordances of the Thing.	optional	Map of `EventAffordance`
`links`	Provides Web links to arbitrary resources that relate to the specified Thing Description.	optional	Array of `Link`

Vocabulary term	*Description*	*Assignment*	*Type*
`forms`	Set of form hypermedia controls that describe how an operation can be performed. Forms are serializations of Protocol Bindings. In this version of TD, all operations that can be described at the Thing level are concerning how to interact with the Thing's Properties collectively at once.	optional	Array of `Form`
`security`	Set of security definition names, chosen from those defined in `securityDefinitions`. These must all be satisfied for access to resources.	mandatory	`string` or Array of `string`
`securityDefinitions`	Set of named security configurations (definitions only). Not actually applied unless names are used in a `security` name-value pair.	mandatory	Map of `SecurityScheme`

The `@context` name-value pair *MUST* contain the anyURI `https://www.w3.org/2019/wot/td/v1` either directly when of type `anyURI` or as first element when of type Array. When `@context` is an Array, the anyURI `https://www.w3.org/2019/wot/td/v1` *MAY* be followed by elements of type `anyURI` or type Map in any order, while it is *RECOMMENDED* to include only one Map with all the name-value pairs in the `@context` Array. Maps contained in an `@context` Array *MAY* contain name-value pairs, where the value is a namespace identifier of type `anyURI` and the name a Term or prefix denoting that

namespace. One Map contained in an `@context` Array *SHOULD* contain a name-value pair that defines the default language for the Thing Description, where the name is the Term `@language` and the value is a well-formed language tag as defined by [BCP47] (e.g., `en`, `de-AT`, `gsw-CH`, `zh-Hans`, `zh-Hant-HK`, `sl-nedis`).

The computation of the base direction of all human-readable text strings is defined by the following set of rules:

- If no language tag is given, the base direction *SHOULD* be inferred through first-strong heuristics or detection algorithms such as the CLDR Likely Subtags [LDML].
- Outside of `MultiLanguage` Maps, the base direction *MAY* be inferred from the language tag of the default language.
- Inside of `MultiLanguage` Maps, the base direction of each value of the name-value pairs *MAY* be inferred from the language tag given in the corresponding name.
- In cases where a language can be written in more than one script with different base directions, the corresponding language tag given in `@language` or `MultiLanguage` Maps *MUST* include a script subtag, so that an appropriate base direction can be inferred. An example is Azeri, which is written LTR when Latin script is used (specified using `az-Latn`) and RTL when Arabic script is used (specified using `az-Arab`).

TD Processors should be aware of certain special cases when processing bidirectional text. They should take care to use bidi isolation when presenting strings to users, particularly when embedding in surrounding text (e.g., for Web user interface). Mixed direction text can occur in any language, even when the language is properly identified.

TD producers should attempt to provide mixed direction strings in a way that can be displayed successfully by a naive user agent. For example, if an RTL string begins with an LTR run (such as a number or a brand or trade name in Latin script), including an RLM character at the start of the string or wrapping opposite direction runs in bidi controls can assist in proper display.

Strings on the Web: Language and Direction Metadata [string-meta] provides some guidance and illustrates a number of pitfalls when using bidirectional text.

In addition to the explicitly provided Interaction Affordances in the `properties`, `actions`, and `events` Arrays, a Thing can also provide meta-interactions, which are indicated by `Form` instances in its optional `forms` Array. When the `forms` Array of a Thing instance contains `Form` instances, the string values assigned to the name `op`, either directly or within an Array, *MUST* be one of the following *operation types*: `readallproperties`, `writeallproperties`, `readmultipleproperties`, or `writemultipleproperties`. (See an example for an usage of `form` in a Thing instance.)

The data schema for each of these meta-interactions is constructed by combining the data schemas of each `PropertyAffordance` instance in a single `ObjectSchema` instance, where the `properties` Map of the `ObjectSchema` instance contains each data schema of the `PropertyAffordances` identified by the name of the corresponding `PropertyAffordances` instance.

If not specified otherwise (e.g., through a TD Context Extension), the request data of the `readmultipleproperties` operation is an Array that contains the intended `PropertyAffordances` instance names, which is serialized to the content type specified by the `Form` instance.

5.3.1.2 `InteractionAffordance` §

Metadata of a Thing that shows the possible choices to Consumers, thereby suggesting how Consumers may interact with the Thing. There are many types of potential affordances, but W3C WoT defines three types of Interaction Affordances: Properties, Actions, and Events.

Vocabulary term	*Description*	*Assignment*	*Type*
`@type`	JSON-LD keyword to label the object with semantic tags (or types).	optional	`string` or Array of `string`
`title`	Provides a human-readable title (e.g., display a text for UI representation) based on a default language.	optional	`string`
`titles`	Provides multi-language human-readable titles (e.g., display a text for UI representation in different languages).	optional	`MultiLanguage`
`description`	Provides additional (human-readable) information based on a default language.	optional	`string`

Vocabulary term	*Description*	*Assignment*	*Type*
`descriptions`	Can be used to support (human-readable) information in different languages.	optional	`MultiLanguage`
`forms`	Set of form hypermedia controls that describe how an operation can be performed. Forms are serializations of Protocol Bindings.	mandatory	Array of `Form`
`uriVariables`	Define URI template variables as collection based on DataSchema declarations.	optional	Map of `DataSchema`

The class `InteractionAffordance` has the following subclasses:

- `PropertyAffordance`
- `ActionAffordance`
- `EventAffordance`

5.3.1.3 `PropertyAffordance` §

An Interaction Affordance that exposes state of the Thing. This state can then be retrieved (read) and optionally updated (write). Things can also choose to make Properties observable by pushing the new state after a change.

Vocabulary term	*Description*	*Assignment*	*Type*
`observable`	A hint that indicates whether Servients hosting the Thing and Intermediaries should provide a Protocol Binding that supports the `observeproperty` operation for this Property.	optional	`boolean`

NOTE

Property instances are also instances of the class DataSchema. Therefore, it can contain the `type`, `unit`, `readOnly` and `writeOnly` members, among others.

`PropertyAffordance` is a Subclass of the `InteractionAffordance` Class and the `DataSchema` Class. When a Form instance is within a `PropertyAffordance` instance, the value assigned to `op` *MUST* be one of `readproperty`, `writeproperty`, `observeproperty`, `unobserveproperty` or an Array containing a combination of these terms.

5.3.1.4 `ActionAffordance` §

An Interaction Affordance that allows to invoke a function of the Thing, which manipulates state (e.g., toggling a lamp on or off) or triggers a process on the Thing (e.g., dim a lamp over time).

Vocabulary term	*Description*	*Assignment*	*Type*
`input`	Used to define the input data schema of the Action.	optional	`DataSchema`
`output`	Used to define the output data schema of the Action.	optional	`DataSchema`
`safe`	Signals if the Action is safe (=true) or not. Used to signal if there is no internal state (cf. resource state) is changed when invoking an Action. In that case responses can be cached as example.	with default	`boolean`
`idempotent`	Indicates whether the Action is idempotent (=true) or not. Informs whether the Action can be called repeatedly with the same result, if present, based on the same input.	with default	`boolean`

`ActionAffordance` is a Subclass of the `InteractionAffordance` Class. When a Form instance is within an `ActionAffordance` instance, the value assigned to op *MUST* be `invokeaction`.

5.3.1.5 `EventAffordance` §

An Interaction Affordance that describes an event source, which asynchronously pushes event data to Consumers (e.g., overheating alerts).

Vocabulary term	*Description*	*Assignment*	*Type*
`subscription`	Defines data that needs to be passed upon subscription, e.g., filters or message format for setting up Webhooks.	optional	`DataSchema`
`data`	Defines the data schema of the Event instance messages pushed by the Thing.	optional	`DataSchema`
`cancellation`	Defines any data that needs to be passed to cancel a subscription, e.g., a specific message to remove a Webhook.	optional	`DataSchema`

`EventAffordance` is a Subclass of the `InteractionAffordance` Class. When a Form instance is within an `EventAffordance` instance, the value assigned to `op` *MUST* be either `subscribeevent`, `unsubscribeevent`, or both terms within an Array.

5.3.1.6 `VersionInfo` §

Metadata of a Thing that provides version information about the TD document. If required, additional version information such as firmware and hardware version (term definitions outside of the TD namespace) can be extended via the TD Context Extension mechanism.

Vocabulary term	*Description*	*Assignment*	*Type*

Vocabulary term	*Description*	*Assignment*	*Type*
`instance`	Provides a version indicator of this TD instance.	mandatory	`string`

It is recommended that the values within instances of the `VersionInfo` Class follow the semantic versioning pattern, where a sequence of three numbers separated by a dot indicates the major version, minor version, and patch version, respectively. See [SEMVER] for details.

5.3.1.7 `MultiLanguage` §

A Map providing a set of human-readable texts in different languages identified by language tags described in [BCP47]. See § 6.3.2 Human-Readable Metadata for example usages of this container in a Thing Description instance.

Each name of the `MultiLanguage` Map *MUST* be a language tag as defined in [BCP47]. Each value of the `MultiLanguage` Map *MUST* be of type `string`.

5.3.2 Data Schema Vocabulary Definitions §

The data schema vocabulary definition is reflecting a very common subset of the terms defined by JSON Schema [JSON-SCHEMA]. It is noted that data schema definitions within Thing Description instances are not limited to this defined subset and may use additional terms found in JSON Schema using a TD Context Extension for the additional terms as described in § 7. TD Context Extensions, otherwise these terms are semantically ignored by TD Processors (for details about semantic processing, please refer to § D. JSON-LD Context Usage and the documentation under the namespace IRIs, e.g., https://www.w3.org/2019/wot/td).

A data schema is an abstract notation for data contained in data formats. In a TD, concrete data formats are specified in Forms (see § 5.3.4.2 `Form`) using content types. When the value of a content type in an instance of the Form is `application/json`, the data schema can be processed directly by JSON Schema processors. Otherwise, Web of Things (WoT) Binding Templates [WOT-BINDING-TEMPLATES] defines data schema's available mappings to other content types such as XML [xml]. If the content type in an instance of the Form is not `application/json` and if no mapping is defined for the content type, specifying a data schema does not make sense for the content type.

5.3.2.1 `DataSchema` §

Metadata that describes the data format used. It can be used for validation.

Vocabulary term	*Description*	*Assignment*	*Type*
`@type`	JSON-LD keyword to label the object with semantic tags (or types)	optional	`string` or Array of `string`
`title`	Provides a human-readable title (e.g., display a text for UI representation) based on a default language.	optional	`string`
`titles`	Provides multi-language human-readable titles (e.g., display a text for UI representation in different languages).	optional	`MultiLanguage`
`description`	Provides additional (human-readable) information based on a default language.	optional	`string`
`descriptions`	Can be used to support (human-readable) information in different languages.	optional	`MultiLanguage`
`type`	Assignment of JSON-based data types compatible with JSON Schema (one of boolean, integer, number, string, object, array, or null).	optional	`string` (one of `object`, `array`, `string`, `number`, `integer`, `boolean`, or `null`)
`const`	Provides a constant value.	optional	any type

Vocabulary term	*Description*	*Assignment*	*Type*
`unit`	Provides unit information that is used, e.g., in international science, engineering, and business.	optional	`string`
`oneOf`	Used to ensure that the data is valid against one of the specified schemas in the array.	optional	Array of `DataSchema`
`enum`	Restricted set of values provided as an array.	optional	Array of any type
`readOnly`	Boolean value that is a hint to indicate whether a property interaction / value is read only (=true) or not (=false).	with default	`boolean`
`writeOnly`	Boolean value that is a hint to indicate whether a property interaction / value is write only (=true) or not (=false).	with default	`boolean`
`format`	Allows validation based on a format pattern such as "date-time", "email", "uri", etc. (Also see below.)	optional	`string`

The class `DataSchema` has the following subclasses:

- `ArraySchema`
- `BooleanSchema`
- `NumberSchema`
- `IntegerSchema`
- `ObjectSchema`

- `StringSchema`
- `NullSchema`

The `format` string values are known from a fixed set of values and their corresponding format rules defined in [JSON-SCHEMA] (Section 7.3 Defined Formats in particular). Servients *MAY* use the `format` value to perform additional validation accordingly. When a value that is not found in the known set of values is assigned to `format`, such a validation *SHOULD* succeed.

5.3.2.2 `ArraySchema` §

Metadata describing data of type Array. This Subclass is indicated by the value `array` assigned to `type` in `DataSchema` instances.

Vocabulary term	*Description*	*Assignment*	*Type*
`items`	Used to define the characteristics of an array.	optional	`DataSchema` or Array of `DataSchema`
`minItems`	Defines the minimum number of items that have to be in the array.	optional	`unsignedInt`
`maxItems`	Defines the maximum number of items that have to be in the array.	optional	`unsignedInt`

5.3.2.3 `BooleanSchema` §

Metadata describing data of type `boolean`. This Subclass is indicated by the value `boolean` assigned to `type` in `DataSchema` instances.

5.3.2.4 `NumberSchema` §

Metadata describing data of type `number`. This Subclass is indicated by the value `number` assigned to `type` in `DataSchema` instances.

Vocabulary term	*Description*	*Assignment*	*Type*
`minimum`	Specifies a minimum numeric value. Only applicable for associated number or integer types.	optional	`double`
`maximum`	Specifies a maximum numeric value. Only applicable for associated number or integer types.	optional	`double`

5.3.2.5 `IntegerSchema` §

Metadata describing data of type `integer`. This Subclass is indicated by the value `integer` assigned to `type` in `DataSchema` instances.

Vocabulary term	*Description*	*Assignment*	*Type*
`minimum`	Specifies a minimum numeric value. Only applicable for associated number or integer types.	optional	`integer`
`maximum`	Specifies a maximum numeric value. Only applicable for associated number or integer types.	optional	`integer`

5.3.2.6 `ObjectSchema` §

Metadata describing data of type `object`. This Subclass is indicated by the value `object` assigned to `type` in `DataSchema` instances.

Vocabulary term	*Description*	*Assignment*	*Type*

Vocabulary term	*Description*	*Assignment*	*Type*
`properties`	Data schema nested definitions.	optional	Map of `DataSchema`
`required`	Defines which members of the object type are mandatory.	optional	Array of `string`

5.3.2.7 `StringSchema` §

Metadata describing data of type `string`. This Subclass is indicated by the value `string` assigned to `type` in `DataSchema` instances.

5.3.2.8 `NullSchema` §

Metadata describing data of type `null`. This Subclass is indicated by the value `null` assigned to `type` in `DataSchema` instances. This Subclass describes only one acceptable value, namely `null`. It can be used as part of a `oneOf` declaration, where it is used to indicate, that the data can also be `null`.

5.3.3 Security Vocabulary Definitions §

This specification provides a selection of well-established security mechanisms that are directly built into protocols eligible as Protocol Bindings for W3C WoT or are widely in use with those protocols. The current set of HTTP security schemes is partly based on OpenAPI 3.0.1 (see also [OPENAPI]). However while the HTTP security schemes, Vocabulary, and syntax given in this specification share many similarities with OpenAPI, they are not compatible.

5.3.3.1 `SecurityScheme` §

Metadata describing the configuration of a security mechanism. The value assigned to the name `scheme` *MUST* be defined within a Vocabulary included in the Thing Description, either in the standard Vocabulary defined in § 5. TD Information Model or in a TD Context Extension.

Vocabulary term	*Description*	*Assignment*	*Type*

Vocabulary term	*Description*	*Assignment*	*Type*
`@type`	JSON-LD keyword to label the object with semantic tags (or types).	optional	`string` or Array of `string`
`scheme`	Identification of the security mechanism being configured.	mandatory	`string` (e.g., `nosec`, `basic`, `digest`, `bearer`, `psk`, `oauth2`, or `apikey`)
`description`	Provides additional (human-readable) information based on a default language.	optional	`string`
`descriptions`	Can be used to support (human-readable) information in different languages.	optional	`MultiLanguage`
`proxy`	URI of the proxy server this security configuration provides access to. If not given, the corresponding security configuration is for the endpoint.	optional	`anyURI`

The class `SecurityScheme` has the following subclasses:

- `NoSecurityScheme`
- `BasicSecurityScheme`
- `DigestSecurityScheme`
- `APIKeySecurityScheme`
- `BearerSecurityScheme`
- `PSKSecurityScheme`
- `OAuth2SecurityScheme`

5.3.3.2 `NoSecurityScheme` §

A security configuration corresponding to identified by the Vocabulary Term `nosec` (i.e., `"scheme": "nosec"`), indicating there is no authentication or other mechanism required to access the resource.

5.3.3.3 `BasicSecurityScheme` §

Basic Authentication [RFC7617] security configuration identified by the Vocabulary Term `basic` (i.e., `"scheme": "basic"`), using an unencrypted username and password. This scheme should be used with some other security mechanism providing confidentiality, for example, TLS.

Vocabulary term	*Description*	*Assignment*	*Type*
`in`	Specifies the location of security authentication information.	with default	`string` (one of `header`, `query`, `body`, or `cookie`)
`name`	Name for query, header, or cookie parameters.	optional	`string`

5.3.3.4 `DigestSecurityScheme` §

Digest Access Authentication [RFC7616] security configuration identified by the Vocabulary Term `digest` (i.e., `"scheme": "digest"`). This scheme is similar to basic authentication but with added features to avoid man-in-the-middle attacks.

Vocabulary term	*Description*	*Assignment*	*Type*
`qop`	Quality of protection.	with default	`string` (one of `auth`, or `auth-int`)
`in`	Specifies the location of security authentication information.	with default	`string` (one of `header`, `query`, `body`, or `cookie`)

Vocabulary term	*Description*	*Assignment*	*Type*
`name`	Name for query, header, or cookie parameters.	optional	`string`

5.3.3.5 `APIKeySecurityScheme` §

API key authentication security configuration identified by the Vocabulary Term `apikey` (i.e., `"scheme": "apikey"`). This is for the case where the access token is opaque and is not using a standard token format.

Vocabulary term	*Description*	*Assignment*	*Type*
`in`	Specifies the location of security authentication information.	with default	`string` (one of `header`, `query`, `body`, or `cookie`)
`name`	Name for query, header, or cookie parameters.	optional	`string`

5.3.3.6 `BearerSecurityScheme` §

Bearer Token [RFC6750] security configuration identified by the Vocabulary Term `bearer` (i.e., `"scheme": "bearer"`) for situations where bearer tokens are used independently of OAuth2. If the `oauth2` scheme is specified it is not generally necessary to specify this scheme as well as it is implied. For `format`, the value `jwt` indicates conformance with [RFC7519], `jws` indicates conformance with [RFC7797], `cwt` indicates conformance with [RFC8392], and `jwe` indicates conformance with [RFC7516], with values for `alg` interpreted consistently with those standards. Other formats and algorithms for bearer tokens *MAY* be specified in vocabulary extensions.

Vocabulary term	*Description*	*Assignment*	*Type*

Vocabulary term	*Description*	*Assignment*	*Type*
`authorization`	URI of the authorization server.	optional	`anyURI`
`alg`	Encoding, encryption, or digest algorithm.	with default	`string` (e.g., `MD5`, `ES256`, or `ES512-256`)
`format`	Specifies format of security authentication information.	with default	`string` (e.g., `jwt`, `cwt`, `jwe`, or `jws`)
`in`	Specifies the location of security authentication information.	with default	`string` (one of `header`, `query`, `body`, or `cookie`)
`name`	Name for query, header, or cookie parameters.	optional	`string`

5.3.3.7 `PSKSecurityScheme` §

Pre-shared key authentication security configuration identified by the Vocabulary Term `psk` (i.e., `"scheme": "psk"`).

Vocabulary term	*Description*	*Assignment*	*Type*
`identity`	Identifier providing information which can be used for selection or confirmation.	optional	`string`

5.3.3.8 `OAuth2SecurityScheme` §

OAuth2 authentication security configuration for systems conformant with [RFC6749] and [RFC8252], identified by the Vocabulary Term `oauth2` (i.e., `"scheme": "oauth2"`). For the `code` flow both `authorization` and `token` *MUST* be included. If no `scopes` are defined in the `SecurityScheme` then they are considered to be empty.

Vocabulary term	*Description*	*Assignment*	*Type*
`authorization`	URI of the authorization server.	optional	`anyURI`
`token`	URI of the token server.	optional	`anyURI`
`refresh`	URI of the refresh server.	optional	`anyURI`
`scopes`	Set of authorization scope identifiers provided as an array. These are provided in tokens returned by an authorization server and associated with forms in order to identify what resources a client may access and how. The values associated with a form should be chosen from those defined in an `OAuth2SecurityScheme` active on that form.	optional	`string` or Array of `string`
`flow`	Authorization flow.	mandatory	`string` (e.g., `code`)

5.3.4 Hypermedia Controls Vocabulary Definitions §

The present model provides a representation for (typed) Web links and Web forms exposed by a Thing. The `Link` class definition is reflecting a very common subset of the terms defined in Web Linking [RFC8288]. The defined terms can be used, e.g., to describe the relation to another Thing such as a *Lamp Thing* is controlled by a *Switch Thing*. The `Form` class corresponds to a newly introduced form of hypermedia control to manipulate the state of Things (and other Web resources).

5.3.4.1 `Link` §

A link can be viewed as a statement of the form "***link context*** has a ***relation type*** resource at ***link target***", where the optional ***target attributes*** may further describe the resource.

Vocabulary term	*Description*	*Assignment*	*Type*
`href`	Target IRI of a link or submission target of a form.	mandatory	`anyURI`
`type`	Target attribute providing a hint indicating what the media type [RFC2046] of the result of dereferencing the link should be.	optional	`string`
`rel`	A link relation type identifies the semantics of a link.	optional	`string`
`anchor`	Overrides the link context (by default the Thing itself identified by its `id`) with the given URI or IRI.	optional	`anyURI`

5.3.4.2 `Form` §

A form can be viewed as a statement of "To perform an ***operation type*** operation on ***form context***, make a ***request method*** request to ***submission target***" where the optional ***form fields*** may further describe the required request. In Thing Descriptions, the ***form context*** is the surrounding Object, such as Properties, Actions, and Events or the Thing itself for meta-interactions.

Vocabulary term	*Description*	*Assignment*	*Type*

Vocabulary term	*Description*	*Assignment*	*Type*
`op`	Indicates the semantic intention of performing the operation(s) described by the form. For example, the Property interaction allows get and set operations. The protocol binding may contain a form for the get operation and a different form for the set operation. The op attribute indicates which form is for which and allows the client to select the correct form for the operation required. op can be assigned one or more interaction verb(s) each representing a semantic intention of an operation.	with default	`string` or Array of `string` (one of `readproperty`, `writeproperty`, `observeproperty`, `unobserveproperty`, `invokeaction`, `subscribeevent`, `unsubscribeevent`, `readallproperties`, `writeallproperties`, `readmultipleproperties`, or `writemultipleproperties`)
`href`	Target IRI of a link or submission target of a form.	mandatory	`anyURI`
`contentType`	Assign a content type based on a media type (e.g., `text/plain`) and potential parameters (e.g., `charset=utf-8`) for the media type [RFC2046].	with default	`string`

Vocabulary term	*Description*	*Assignment*	*Type*
`contentCoding`	Content coding values indicate an encoding transformation that has been or can be applied to a representation. Content codings are primarily used to allow a representation to be compressed or otherwise usefully transformed without losing the identity of its underlying media type and without loss of information. Examples of content coding include "gzip", "deflate", etc. .	optional	`string`

Vocabulary term	*Description*	*Assignment*	*Type*
`subprotocol`	Indicates the exact mechanism by which an interaction will be accomplished for a given protocol when there are multiple options. For example, for HTTP and Events, it indicates which of several available mechanisms should be used for asynchronous notifications such as long polling (`longpoll`), WebSub [websub] (`websub`), Server-Sent Events (`sse`) [html] (also known as EventSource). Please note that there is no restriction on the subprotocol selection and other mechanisms can also be announced by this subprotocol term.	optional	`string` (e.g., `longpoll`, `websub`, or `sse`)
`security`	Set of security definition names, chosen from those defined in `securityDefinitions`. These must all be satisfied for access to resources.	optional	`string` or Array of `string`

Vocabulary term	*Description*	*Assignment*	*Type*
`scopes`	Set of authorization scope identifiers provided as an array. These are provided in tokens returned by an authorization server and associated with forms in order to identify what resources a client may access and how. The values associated with a form should be chosen from those defined in an `OAuth2SecurityScheme` active on that form.	optional	`string` or Array of `string`
`response`	This optional term can be used if, e.g., the output communication metadata differ from input metadata (e.g., output contentType differ from the input contentType). The response name contains metadata that is only valid for the response messages.	optional	`ExpectedResponse`

Possible values for the `contentCoding` property can be found, e.g., in the IANA HTTP content coding registry.

The list of possible operation types of a form is fixed. As of this version of the specification, it only includes the well-known types necessary to implement the WoT interaction model described in [WOT-ARCHITECTURE]. Future versions of the standard may extend this list but operations types *SHOULD NOT* be arbitrarily set by servients.

The optional `response` name-value pair can be used to provide metadata for the expected response message. With the core vocabulary, it only includes content type information, but TD Context

Extensions could be applied. If no `response` name-value pair is provided, it *MUST* be assumed that the content type of the response is equal to the content type assigned to the Form instance. Note that `contentType` within an `ExpectedResponse` Class does not have a Default Value. For instance, if the value of the content type of the form is `application/xml` the assumed value of the content type of the response will be also `application/xml`.

In some use cases, input and output data might be represented in a different form, for instance an Action that accepts JSON, but returns an image. In such a case, the optional `response` name-value pair can describe the content type of the expected response. If the content type of the expected response differs from the content type of the form, the `Form` instance *MUST* include a name-value pair with the name `response`. For instance, an `ActionAffordance` could only accept `application/json` for its input data, while it will respond with an `image/jpeg` content type for its output data. In that case the content types differ and the `response` name-value pair has to be used to provide response content type (`image/jpeg`) information to the Consumer.

5.3.4.3 `ExpectedResponse` §

Communication metadata describing the expected response message.

Vocabulary term	*Description*	*Assignment*	*Type*
`contentType`	Assign a content type based on a media type (e.g., `text/plain`) and potential parameters (e.g., `charset=utf-8`) for the media type [RFC2046].	mandatory	`string`

5.4 Default Value Definitions §

When assignments in a TD are missing, a TD Processor *MUST* follow the Default Value assignments expressed in the table of § 5.4 Default Value Definitions.

The following table gives all Default Values defined in the TD Information Model.

Class	*Vocabulary Term*	*Default Value*	*Comment*

Class	*Vocabulary Term*	*Default Value*	*Comment*
`Form`	`contentType`	`application/json`	
`DataSchema`	`readOnly`	`false`	
`DataSchema`	`writeOnly`	`false`	
`ActionAffordance`	`safe`	`false`	
`ActionAffordance`	`idempotent`	`false`	
`Form`	`op`	Array of `string` with the elements `readproperty` and `writeproperty`	If defined within an instance of `PropertyAffordance`
`Form`	`op`	`invokeaction`	If defined within an instance of `ActionAffordance`
`Form`	`op`	`subscribeevent`	If defined within an instance of `EventAffordance`
`BasicSecurityScheme`	`in`	`header`	
`DigestSecurityScheme`	`in`	`header`	
`BearerSecurityScheme`	`in`	`header`	
`APIKeySecurityScheme`	`in`	`query`	
`DigestSecurityScheme`	`qop`	`auth`	
`BearerSecurityScheme`	`alg`	`ES256`	
`BearerSecurityScheme`	`format`	`jwt`	

6. TD Representation Format §

WoT Thing Descriptions represent Things and are modeled and structured based on § 5. TD Information Model. This section defines a JSON-based representation format for Things, a serialization of instances of the Class `Thing` defined by the TD Information Model.

A TD Processor *MUST* be able to serialize Thing Descriptions into the JSON format [RFC8259] and/or deserialize Thing Descriptions from that format, according to the rules noted in § 6.1 Mapping to JSON Types and § 6.3 Information Model Serialization.

The JSON serialization of the TD Information Model is aligned with the syntax of JSON-LD 1.1 [json-ld11] in order to streamline semantic evaluation. Hence, the TD representation format can be processed either as raw JSON or with a JSON-LD 1.1 processor (for details about semantic processing, please refer to § D. JSON-LD Context Usage and the documentation under the namespace IRIs, e.g., https://www.w3.org/2019/wot/td).

In order to support interoperable internationalization, TDs *MUST* be serialized according to the requirements defined in Section 8.1 of RFC8259 [RFC8259] for open ecosystems. In summary, this requires the following:

- TDs *MUST* be encoded using UTF-8 [RFC3629].
- Implementations *MUST NOT* add a byte order mark (U+FEFF) to the beginning of a TD document.
- TD Processors *MAY* ignore the presence of a byte order mark rather than treating it as an error.

6.1 Mapping to JSON Types §

The TD Information Model is constructed, so that there is an easy mapping between model Objects and JSON types. Every Class instances maps to a JSON object, where each name-value pair of the Class instance is a member of the JSON object.

Every Simple Type mentioned in § 5.3 Class Definitions (i.e., `string`, `anyURI`, `dateTime`, `integer`, `unsignedInt`, `double`, and `boolean`) maps to a primitive JSON type (string, number, boolean), as per the rules listed below. These rules apply to values in name-value pairs:

- Values that are of type `string` or `anyURI` *MUST* be serialized as JSON strings.
- Values that are of type `dateTime` *MUST* be serialized as JSON strings following the "date-time" format specified by [RFC3339]. Examples would include `2019-05-24T13:12:45Z` and `2015-07-11T09:32:26+08:00`. Values that are of type `dateTime` *SHOULD* use the literal `Z` representing the UTC time zone instead of an offset.

- Values that are of type `integer` or `unsignedInt` *MUST* be serialized as JSON numbers without a fraction or exponent part.
- Values that are of type `double` *MUST* be serialized as JSON number.
- Values that are of type `boolean` *MUST* be serialized as JSON boolean.

Every complex type of the TD Information Model (i.e., Arrays, Maps, and Class instances) maps to a structured JSON type (array and object), as per the rules listed below:

- A value of type Array *MUST* be serialized as JSON array, with each value of the name-value pairs as element of the JSON array ordered by the numeric name of the pair.
- A value of type Map *MUST* be serialized as JSON object, with each name-value pair as member of the JSON object.
- A Class instance *MUST* be serialized as JSON object, following the detailed rules given individually in § 6.3 Information Model Serialization.

6.2 Omitting Default Values §

A Thing Description serialization may omit Vocabulary Term for which Default Values are defined, as listed in the table given in § 5.4 Default Value Definitions.

The following example shows the TD instance from Example 1 with a checkbox to also include the members with Default Values (=checkbox checked). These members can be omitted (=checkbox unchecked) to simplify the TD serialization. Note that a TD Processor interprets these omitted members identically as if they were explicitly present with a given Default Value.

> EXAMPLE 3
>
> ☐ *with Default Values*

Please note that, depending on the Protocol Binding used, additional protocol-specific Vocabulary Terms may apply. They may also have associated Default Values, and hence can also be omitted as explained in this subsection. Further information can be found in § 8.3 Protocol Bindings.

6.3 Information Model Serialization §

6.3.1 Thing Root Object §

A Thing Description is a data structure rooted at an Object of type `Thing`. In turn, a JSON serialization of the Thing Description is a JSON object, which is the root of a syntax tree constructed from the TD Information Model.

The root element of a TD Serialization *MUST* be a JSON object that includes a member with the name `@context` and a value of type string or array that equals or respectively contains `https://www.w3.org/2019/wot/td/v1`.

In general, this URI is used to identify the TD representation format version defined by this specification. For JSON-LD processing [json-ld11], this URI specifies the Thing Description context file. An `@context` of type array indicates TD Context Extensions (see § 7. TD Context Extensions for details).

EXAMPLE 4

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    ...
}
```

All name-value pairs of an instance of `Thing`, where the name is a Vocabulary Term in the Signature of `Thing`, *MUST* be serialized as JSON members of the root object.

A TD snippet for a serialized root object including all mandatory and optional members is given below:

EXAMPLE 5: Sample of Thing serializations

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "@type": "Thing",
    "id": "urn:dev:ops:32473-Thing-1234",
    "title": "MyThing",
    "titles": {...},
    "description": "Human readable information.",
    "descriptions": {...},
    "support": "mailto:support@example.com",
    "version" : {...},
    "created" : "2018-11-14T19:10:23.824Z",
    "modified" : "2019-06-01T09:12:43.124Z",
    "securityDefinitions": {...},
    "security": ...,
    "base": "https://servient.example.com/",
    "properties": {...},
    "actions": {...},
    "events": {...},
    "links": [...],
    "forms": [...]
}
```

All values assigned to `version`, `securityDefinitions`, `properties`, `actions`, and `events` in an instance of the Class `Thing` *MUST* be serialized as JSON objects.

All values assigned to `links`, and `forms` in an instance of the Class `Thing` *MUST* be serialized as JSON arrays containing JSON objects as defined in § 6.3.8 `links` and § 6.3.9 `forms`, respectively.

The value assigned to `security` in an instance of Class `Thing` *MUST* be serialized as JSON string or as JSON array whose elements are JSON strings.

6.3.2 Human-Readable Metadata §

JSON members named `title` and `description` are used within a TD document to provide human-readable metadata. They can be used as comments for developers inspecting a TD document or as display texts for user interface.

As defined in § 5.3.1.1 `Thing`, the base text direction used to display human-readable metadata can either be estimated using heuristics such as the first-strong rule or inferred from language information.

In TD documents the default language is defined by a value assigned to `@language` in the `@context`, and this, along with a script subtag if necessary, can be used to determine a base text direction. However, when interpreting human-readable text, each human-readable string value *MUST* be processed independently. In other words, a TD Processor cannot carry forward changes in direction from one string to another, or infer direction for one string from another one elsewhere in the TD.

> NOTE
>
> Strings on the Web [STRING-META] suggests both strong-first and language-based inferencing as means to determine the base text direction. Given that the Thing Description format is based on JSON-LD 1.1 [json-ld11], which currently lacks explicit direction metadata, these approaches are currently considered appropriate at the time of this publication. However, if JSON-LD 1.1 adopts support for explicit base direction metadata as recommended by [STRING-META], the Thing Description format should be updated to take advantage of that feature.

A TD snippet using `title` and `description` is shown below. The default language is set to `en` through the definition of the `@language` member within a JSON object in the `@context` array.

EXAMPLE 6

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        { "@language" : "en" }
    ],
    "title": "MyThing",
    "description": "Human readable information.",
    ...
    "properties": {
        "on": {
            "title" : "On/Off",
            "type": "boolean",
            "forms": [...]
        },
        "status": {
            "title" : "Status",
            "type": "object",
            ...
            "forms": [...]
        }
    },
    ...
}
```

The JSON members named `titles` and `descriptions` are used within the TD document to provide human-readable metadata in multiple languages within a single TD document. All name-value pairs of a `MultiLanguage` Map *MUST* be serialized as members of a JSON object, where the name is a well-formed language tag as defined by [BCP47] and the value is a human-readable string in the language indicated by the tag. See § 5.3.1.7 `MultiLanguage` for details. All `MultiLanguage` object within a TD document *SHOULD* contain the same set of language members.

A TD snippet using `titles` and `descriptions` at different levels is given below:

EXAMPLE 7

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "title": "MyThing",
    "titles": {
        "en":"MyThing",
        "de": "MeinDing",
        "ja" : "私の物",
        "zh-Hans" : "我的东西",
        "zh-Hant" : "我的東西"
    },
    "descriptions": {
        "en":"Human readable information.",
        "de": "Menschenlesbare Informationen.",
        "ja" : "人間が読むことができる情報",
        "zh-Hans" : "人们可阅读的信息",
        "zh-Hant" : "人們可閱讀的資訊"
    },
    ...
    "properties": {
        "on": {
            "titles": {
                "en": "On/Off",
                "de": "An/Aus",
                "ja": "オンオフ",
                "zh-Hans": "开关",
                "zh-Hant": "開關" },
            "type": "boolean",
            "forms": [...]
        },
        "status": {
            "titles": {
                "en": "Status",
                "de": "Zustand",
                "ja": "状態",
                "zh-Hans": "状态",
                "zh-Hant": "狀態" },
            "type": "object",
            ...
            "forms": [...]
        }
```

```
    },
    ...
}
```

TD instances may also combine the use of `title` and `description` with `titles` and `descriptions`. When `title` and `titles` or `description` and `descriptions` are present within the same JSON object, the values of `title` and `description` *MAY* be seen as the default text. When `title` and `titles` or `description` and `descriptions` are present in a TD document, each `title` and `description` member *SHOULD* have a corresponding `titles` and `descriptions` member, respectively. The language of the default text is indicated by the default language, which is usually set by the creator of the Thing Description instance.

EXAMPLE 8

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        { "@language" : "de" }
    ],
    "title": "MyThing",
    "titles": {
        "en":"MyThing",
        "de": "MeinDing",
        "ja" : "私の物",
        "zh-Hans" : "我的东西",
        "zh-Hant" : "我的東西"
    },
    "description": "Menschenlesbare Informationen.",
    "descriptions": {
        "en":"Human readable information.",
        "de": "Menschenlesbare Informationen.",
        "ja" : "人間が読むことができる情報",
        "zh-Hans" : "人们可阅读的信息",
        "zh-Hant" : "人們可閱讀的資訊"
    },
    ...
    "properties": {
        "on": {
            "title" : "An/Aus",
            "titles": {
                "en": "On/Off",
                "de": "An/Aus",
                "ja": "オンオフ",
                "zh-Hans": "开关",
                "zh-Hant": "開關" },
            "type": "boolean",
            "forms": [...]
        },
        "status": {
            "title" : "Zustand",
            "titles": {
                "en": "Status",
                "de": "Zustand",
                "ja": "状態",
                "zh-Hans": "状态",
                "zh-Hant": "狀態" },
```

```
            "type": "object",
            ...
            "forms": [...]
        }
    },
    ...
}
```

Another possibility to set the default language is through a language negotiation mechanism, such as the `Accept-Language` header field of HTTP. In cases where the default language has been negotiated, an `@language` member *MUST* be present to indicate the result of the negotiation and the corresponding default language of the returned content. When the default language has been negotiated successfully, TD documents *SHOULD* include the appropriate matching values for the members `title` and `description` in preference to `MultiLanguage` objects in `titles` and `descriptions` members. Note however that Things *MAY* choose to not support such dynamically-generated TDs nor to support language negotiation (e.g., because of resource constraints).

6.3.3 `version` §

All name-value pairs of an instance of `VersionInfo`, where the name is a Vocabulary Term included in the Signature of `VersionInfo`, *MUST* be serialized as JSON members with the Vocabulary Term as name.

A TD snippet of a version information object is given below:

EXAMPLE 9

```
{
    ...
    "version": { "instance": "1.2.1" },
    ...
}
```

The `version` member is intended as container for additional application- and/or device-specific version information based on TD Context Extensions. See § 7.1 Semantic Annotations for details.

6.3.4 `securityDefinitions` and `security` §

In a `Thing` instance, the value assigned to `securityDefinitions` is a Map of instances of `SecurityScheme`. All name-value pairs of a Map of `SecurityScheme` instances *MUST* be serialized as members of the JSON object that results from serializing the Map; the name of a pair *MUST* be serialized as a JSON string and the value of the pair, an instance of `SecurityScheme`, *MUST* be serialized as a JSON object.

All name-value pairs of an instance of one of the Subclasses of `SecurityScheme`, where the name is a Vocabulary Term included in the Signature of that Subclass or in the Signature of `SecurityScheme`, *MUST* be serialized as members of the JSON object that results from serializing the `SecurityScheme` Subclass's instance, with the Vocabulary Term as name.

The following TD snippet shows a simple security configuration specifying basic username/password authentication in the header. The value given for `in` is actually the Default Value (`header`) and could be omitted. A named security configuration must be given in the `securityDefinitions` map. That definition must be activated by including its JSON name in the `security` member, which can be of type string when only one definition is activated.

EXAMPLE 10

```
...
"securityDefinitions": {
    "basic_sc": {
        "scheme": "basic",
        "in": "header"
    }
},
"security": "basic_sc",
...
```

Here is a more complex example: a TD snippet showing digest authentication on a proxy combined with bearer token authentication on the Thing. In the `digest` scheme, the Default Value of `in` (i.e., `header`) is omitted, but still applies. Note that the corresponding private security configuration such as username/password and tokens must be configured in the Consumer to interact successfully. When activating multiple security definitions, the `security` member becomes an array.

EXAMPLE 11

```
...
"securityDefinitions": {
    "proxy_sc": {
        "scheme": "digest",
        "proxy": "https://portal.example.com/"
    },
    "bearer_sc": {
        "in":"header",
        "scheme": "bearer",
        "format": "jwt",
        "alg": "ES256",
        "authorization": "https://servient.example.com:8443/"
    }
},
"security": ["proxy_sc", "bearer_sc"],
...
```

Security configuration in the TD is mandatory. At least one security definition *MUST* be activated through the `security` array at the Thing level (i.e., in the TD root object). This configuration can be seen as the default security mechanism required to interact with the Thing. Security definitions *MAY* also be activated at the form level by including a `security` member in form objects, which overrides (i.e., completely replace) all definitions activated at the Thing level.

The `nosec` security scheme is provided for the case that no security is needed. The minimal security configuration for a Thing is activation of the `nosec` security scheme at the Thing level, as shown in the following example:

EXAMPLE 12

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "id": "urn:dev:ops:32473-Thing-1234",
    "title": "MyThing",
    "description": "Human readable information.",
    "support": "https://servient.example.com/contact",
    "securityDefinitions": { "nosec_sc": { "scheme": "nosec" }},
    "security": "nosec_sc",
    "properties": {...},
    "actions": {...},
    "events": {...},
    "links": [...]
}
```

To give a more complex example, suppose we have a Thing where all Interaction Affordances require basic authentication except for one, for which no authentication is required. For the `status` Property and the `toggle` Action, `basic` authentication is required and defined at the Thing level. For the `overheating` Event, however, no authentication is required, and hence the security configuration is overridden at the form level.

EXAMPLE 13

```
{
    ...
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic"},
        "nosec_sc": {"scheme": "nosec"}
    },
    "security": ["basic_sc"],
    ...
    "properties": {
        "status": {
            ...
            "forms": [{
                "href": "https://mylamp.example.com/status"
            }]
        }
    },
    "actions": {
        "toggle": {
            ...
            "forms": [{
                "href": "https://mylamp.example.com/toggle"
            }]
        }
    },
    "events": {
        "overheating": {
            ...
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "security": ["nosec_sc"]
            }]
        }
    }
}
```

Security configurations can also can be specified for different forms within the same Interaction Affordance. This may be required for devices that support multiple protocols, for example HTTP and CoAP [RFC7252], which support different security mechanisms. This is also useful when alternative authentication mechanisms are allowed. Here is a TD snippet demonstrating three possible ways to activate a Property affordance: via HTTPS with basic authentication, with digest authentication, with

bearer token authentication. In other words, the use of different security configurations within multiple forms provides a way to combine security mechanisms in an "OR" fashion. In contrast, putting multiple security configurations in the same `security` member combines them in an "AND" fashion, since in that case they would all need to be satisfied to allow activation of the Interaction Affordance. Note that activating one (default) configuration at the Thing level is still mandatory.

EXAMPLE 14

```
{
    ...
    "securityDefinitions": {
        "basic_sc": { "scheme": "basic" },
        "digest_sc": { "scheme": "digest" },
        "bearer_sc": { "scheme": "bearer" }
    },
    "security": ["basic_sc"],
    ...
    "properties": {
        "status": {
            ...
            "forms": [{
                "href": "https://mylamp.example.com/status"
            }, {
                "href": "https://mylamp.example.com/status",
                "security": ["digest_sc"]
            }, {
                "href": "https://mylamp.example.com/status",
                "security": ["bearer_sc"]
            }]
        }
    },
    ...
}
```

As another more complex example, OAuth2 makes use of scopes. These are identifiers that may appear in tokens and must match with corresponding identifiers in a resource to allow access to that resource (or Interaction Affordance in the case of W3C WoT). For example, in the following, the `status` Property can be read by Consumers using bearer tokens containing the scope `limited`, but the `configure` Action can only be invoked with a token containing the `special` scope. Scopes are not identical to roles, but are often associated with them; for example, perhaps only those in an administrative role are authorized to perform "special" interactions. Tokens can have more than one

scope. In this example, an administrator would probably be issued tokens with both the `limited` and `special` scopes, while ordinary users would only be issued tokens with the `limited` scope.

EXAMPLE 15

```
{
    ...
    "securityDefinitions": {
        "oauth2_sc": {
            "scheme": "oauth2",
            ...
            "flow": "code",
            "authorization": "https://example.com/authorization",
            "token": "https://example.com/token",
            "scopes": ["limited", "special"]
        }
    },
    "security": ["oauth2_sc"],
    ...
    "properties": {
        "status": {
            ...
            "forms": [{
                "href": "https://scopes.example.com/status",
                "scopes": ["limited"]
            }]
        }
    },
    "actions": {
        "configure": {
            ...
            "forms": [{
                "href": "https://scopes.example.com/configure",
                "scopes": ["special"]
            }]
        }
    },
    ...
}
```

6.3.5 `properties` §

The value assigned to `properties` in a `Thing` instance is a Map of instances of `PropertyAffordance`. All name-value pairs of a Map of `PropertyAffordance` instances *MUST* be serialized as members of the JSON object that results from serializing the Map; the name of a pair *MUST* be serialized as a JSON string and the value of the pair, an instance of `PropertyAffordance`, *MUST* be serialized as a JSON object.

All name-value pairs of an instance of `PropertyAffordance`, where the name is a Vocabulary Term included in (one of) the Signatures of `PropertyAffordance`, `InteractionAffordance`, or `DataSchema`, *MUST* be serialized as members of the JSON object that results from serializing the `PropertyAffordance` instance, with the Vocabulary Term as name. See § 6.3.10 Data Schemas for details on serializing `DataSchema` instances.

The value assigned to `forms` in an instance of `PropertyAffordance` *MUST* be serialized as a JSON array containing one or more JSON object serializations as defined in § 6.3.9 `forms`.

A snippet for two Property affordances is given below:

EXAMPLE 16: Sample of Property serializations

```
...
"properties": {
    "on": {
        "type": "boolean",
        "forms": [...]
    },
    "status": {
        "type": "object",
        "properties": {
            "brightness": {
                "type": "number",
                "minimum": 0.0,
                "maximum": 100.0
            },
            "rgb": {
                "type": "array",
                "items" : {
                    "type" : "number",
                    "minimum": 0,
                    "maximum": 255
                },
                "minItems": 3,
                "maxItems": 3
            }
        },
        "required": ["brightness", "rgb"],
        "forms": [...]
    }
},
...
```

6.3.6 `actions` §

In a `Thing` instance, the value assigned to `actions` is a Map of instances of `ActionAffordance`. All name-value pairs of a Map of `ActionAffordance` instances *MUST* be serialized as members of the JSON object that results from serializing the Map; the name of a pair *MUST* be serialized as a JSON string and the value of the pair, an instance of `ActionAffordance`, *MUST* be serialized as a JSON object.

All name-value pairs of an instance of `ActionAffordance`, where the name is a Vocabulary Term included in (one of) the Signatures of `ActionAffordance` or `InteractionAffordance`, *MUST* be serialized as members of the JSON object that results from serializing the `ActionAffordance` instance, with the Vocabulary Term as name.

The values assigned to `input` and `output` in an instance of `ActionAffordance` *MUST* be serialized as JSON objects. They rely on the Class `DataSchema`, whose serialization is defined in § 6.3.10 Data Schemas.

The value assigned to `forms` in an instance of `ActionAffordance` *MUST* be serialized as a JSON array containing one or more JSON object serializations as defined in § 6.3.9 `forms`.

A TD snippet of an Action affordance is given below:

EXAMPLE 17: Sample of an Action serialization

```
...
"actions": {
    "fade" : {
        "title": "Fade in/out",
        "description": "Smooth fade in and out animation.",
        "input": {
            "type": "object",
            "properties": {
                "from": {
                    "type": "integer",
                    "minimum": 0,
                    "maximum": 100
                },
                "to": {
                    "type": "integer",
                    "minimum": 0,
                    "maximum": 100
                },
                "duration": {"type": "number"}
            },
            "required": ["to","duration"],
        },
        "output": {"type": "string"},
        "forms": [...]
    }
},
...
```

6.3.7 `events` §

In a `Thing` instance, the value assigned to `events` is a map of instances of `EventAffordance`. All name-value pairs of a Map of `EventAffordance` instances *MUST* be serialized as members of the JSON object that results from serializing the Map; the name of a pair *MUST* be serialized as a JSON string and the value of the pair, an instance of `EventAffordance`, *MUST* be serialized as a JSON object.

All name-value pairs of an instance of `EventAffordance`, where the name is a Vocabulary Term included in (one of) the Signatures of `EventAffordance` or `InteractionAffordance`, *MUST* be

serialized as members of the JSON object that results from serializing the `EventAffordance` instance, with the Vocabulary Term as name.

The values assigned to `subscription`, `data`, and `cancellation` in an instance of `EventAffordance` *MUST* be serialized as JSON objects. They rely on the Class `DataSchema`, whose serialization is defined in § 6.3.10 Data Schemas.

The value assigned to `forms` in an instance of `EventAffordance` *MUST* be serialized as a JSON array containing one or more JSON object serializations as defined in § 6.3.9 `forms`.

A TD snippet of an Event object is given below:

EXAMPLE 18: Sample of an Event serialization

```
...
"events": {
    "overheated": {
        "data" : {
            "type": "string"
        },
        "forms": [...]
    }
},
...
```

Event affordances have been defined in a flexible manner, in order to adopt existing (e.g., WebSub [websub]) or customer-oriented event mechanisms (e.g., Webhooks). For this reason, `subscription` and `cancellation` can be defined according to the desired mechanism. Please find further details in [WOT-BINDING-TEMPLATES]. Example § A.3 Webhook Event Example illustrates how Events can use `subscription` and `cancellation` to describe Webhooks.

6.3.8 `links` §

All name-value pairs of an instance of `Link`, where the name is a Vocabulary Term included in the Signature of `Link`, *MUST* be serialized as members of the JSON object that results from serializing the `Link` instance, with the Vocabulary Term as name.

A TD snippet of a link object in the `links` array is given below:

EXAMPLE 19: Sample of a Link serialization

```
...
"links": [{
    "rel": "controlledBy",
    "href": "https://servient.example.com/things/lampController",
    "type": "application/td+json"
}]
...
```

6.3.9 `forms` §

All name-value pairs of an instance of `Form`, where the name is a Vocabulary Term included in the Signature of `Form`, *MUST* be serialized as members of the JSON object that results from serializing the `Form` instance, with the Vocabulary Term as name.

If required, form objects *MAY* be supplemented with protocol-specific Vocabulary Terms identified with a prefix. See also § 8.3 Protocol Bindings.

A TD snippet of a form object in the `forms` array is given below:

EXAMPLE 20: Sample of a Form serialization

```
...
"forms": [{
    "op": "writeproperty",
    "href" : "http://mytemp.example.com:5683/temp",
    "contentType": "application/json",
    "htv:methodName": "POST"
}]
...
```

`href` may also carry a URI that contains dynamic variables such as p and d in http://192.168.1.25/left?p=2&d=1. In that case the URI can be defined as template as defined in [RFC6570]: `http://192.168.1.25/left{?p,d}`.

In such a case, the URI Template variables *MUST* be collected in the JSON-object based `uriVariables` member with the associated (unique) variable names as JSON names.

The serialization of each value in the map assigned to `uriVariables` in an instance of `Form` *MUST* rely on the Class `DataSchema`, whose serialization is defined in § 6.3.10 Data Schemas.

A TD snippet using a URI Template and `uriVariables` is given below:

EXAMPLE 21

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        { "eg": "http://www.example.org/iot#" }
    ],
    ...
    "actions": {
        "LeftDown": {
            ...
            "uriVariables": {
                "p" : { "type": "integer", "minimum": 0, "maximum": 16, "
                "d" : { "type": "integer", "minimum": 0, "maximum": 1, "@
            },
            "forms": [{
              "href" : "http://192.168.1.25/left{?p,d}",
              "htv:methodName": "GET"
            }]
        },
        ...
    },
    ...
}
```

The `contentType` member is used to assign a media type [RFC2046] including media type parameters as attribute-value pairs separated by a `;` character. Example:

EXAMPLE 22

```
...
"contentType" : "text/plain; charset=utf-8",
...
```

In some use cases, the form metadata of the Interaction Affordance not only describes the request, but also provides metadata for the expected response. For instance, an Action `takePhoto` defines an `input` schema to submit parameter settings of a camera (aperture priority, timer, etc.) using JSON for the request payload (i.e., `"contentType": "application/json"`). The output of this action is the photo taken, which is available in JPEG format, for example. In such cases, the `response` member is used to indicate the representation format of the response payload (e.g., `"contentType": "image/jpeg"`). Here no `output` schema is required, as the content type fully specifies the representation format.

If present, the value assigned to `response` in an instance of `Form` *MUST* be a JSON object. If present, the response object *MUST* contain a `contentType` member as defined in the Class definition of `ExpectedResponse`.

A `form` snippet with the `response` member is shown below based on the `takePhoto` Action described above:

EXAMPLE 23

```
{
    ...
    "actions": {
        "takePhoto": {
            ...
            "forms": [{
                "op": "invokeaction",
                "href": "http://camera.example.com/api/snapshot",
                "contentType": "application/json",
                "response": {
                    "contentType": "image/jpeg"
                }
            }]
        }
    },
    ...
}
```

When `forms` is present at the top level, it can be used to describe meta interactions offered by a Thing. For example, the operation types "readallproperties" and "writeallproperties" are for meta interactions with a Thing by which Consumers can read and write all properties at once. In the example below, a `forms` member is included in the TD root object and the Consumer can use the submission target

`https://mylamp.example.com/allproperties` both to read or write all Properties (i.e., `on`, `brightness`, and `timer`) of the Thing in a single protocol transaction.

EXAMPLE 24

```
{
    ...
    "properties": {
        "on": {
            "type": "boolean",
            "forms": [...]
        },
        "brightness": {
            "type": "number",
            "forms": [...]
        },
        "timer": {
            "type": "integer",
            "forms": [...]
        }
    },
    ...
    "forms": [{
        "op": "readallproperties",
        "href": "https://mylamp.example.com/allproperties",
        "contentType": "application/json",
        "htv:methodName": "GET"
    }, {
        "op": "writeallproperties",
        "href": "https://mylamp.example.com/allproperties",
        "contentType": "application/json",
        "htv:methodName": "PUT"
    }]
}
```

In the case of operation type `writeallproperties`, it is expected that the Consumer provides all writable (non `readOnly`) properties and the (new) assigned values (e.g., within payload). Similarly, for the `writemultipleproperties` operation type, it is expected that the Consumer provides writable (non `readOnly`) properties. On the Thing side, Thing is expected to return readable (non `writeOnly`) properties in the case of `readmultipleproperties` and `readallproperties` operation types.

6.3.10 Data Schemas §

The data schemas of the WoT Thing Description defined through the `DataSchema` Class are based on a subset of the JSON Schema terms [JSON-SCHEMA]. Thus, serializations of the TD data schemas can be fed directly into JSON Schema validator implementations to validate the data exchanged with Things.

Data schema serialization applies to `PropertyAffordance` instances, the values assigned to `input` and `output` in `ActionAffordance` instances, the values assigned to `subscription`, `data`, and `cancellation` in `EventAffordance` instances, and the value assigned to `uriVariables` in instances of Subclasses of `InteractionAffordance` (when a form object uses a URI Template).

All name-value pairs of an instance of one of the Subclasses of `DataSchema`, where the name is a Vocabulary Term included in the Signature of that Subclass or in the Signature of `DataSchema`, *MUST* be serialized as members of the JSON object that results from serializing the `DataSchema` Subclass's instance, with the Vocabulary Term as name.

The value assigned to `properties` in an instance of `ObjectSchema` *MUST* be serialized as a JSON object.

The values assigned to `enum`, `required`, and `oneOf` in an instance of `DataSchema` *MUST* be serialized as a JSON array.

The value assigned to `items` in an instance of `ArraySchema` *MUST* be serialized as a JSON object or a JSON array containing JSON objects.

A TD snippet data schema members is given below. Note that the surrounding object may be a data schema object (e.g., for `input` and `output`) or a Property object, which would contain additional members.

EXAMPLE 25: Sample of a DataSchema serialization

```
...
"type": "object",
"properties": {
    "status": {
        "title": "Status",
        "type": "string",
        "enum": ["On", "Off", "Error"]
    },
    "brightness": {
        "title": "Brightness value",
        "type": "number",
        "minimum": 0.0,
        "maximum": 100.0
    },
    "rgb": {
        "title": "RGB color value",
        "type": "array",
        "items" : {
            "type" : "number",
            "minimum": 0,
            "maximum": 255
        },
        "minItems": 3,
        "maxItems": 3
    }
},
...
```

The terms `readOnly` and `writeOnly` can be used signal which data items are exchanged in read interactions (i.e., when reading a Property) and which in write interactions (i.e., when writing a Property). This can be used as workaround when Properties of an unconventional Thing exhibit different data for reading and writing, which can be the case when augmenting an existing device or service with a Thing Description.

A TD snippet with the usage of `readOnly` and `writeOnly` is given below:

EXAMPLE 26

```
...
"properties": {
    "status": {
        "description": "Read or write On/Off status.",
        "type": "object",
        "properties": {
            "latestStatus": {
                "type": "string",
                "enum": ["On", "Off"],
                "readOnly": true
            },
            "newStatusValue": {
                "type": "string",
                "enum": ["On", "Off"],
                "writeOnly": true
            }
        },
        forms: [...]
    }
}
...
```

When the `status` Property is read, the status data is returned using a `latestStatus` member in the payload. To update the `status` Property, the new value must be provided through a `newStatusValue` member in the payload.

As an additional feature, a Thing Description instance allows the usage of a `unit` member within data schemas. This can be used to associate a unit of measure to a data item. Its string value can be selected freely. However, it is recommended to select units defined in well-known Vocabularies. See § 7. TD Context Extensions for an example.

6.4 Identification §

The JSON-based serialization of Thing Descriptions is identified by the media type `application/td+json` or the CoAP Content-Format ID `432` (see § 10. IANA Considerations).

7. TD Context Extensions §

This section is non-normative.

In addition to the standard Vocabulary definitions in § 5. TD Information Model, the WoT Thing Description offers the possibility to add context knowledge from additional namespaces. This mechanism can be used to enrich the Thing Description instances with additional (e.g., domain-specific) semantics. It can also be used to import additional Protocol Bindings or new security schemes in the future.

For such TD Context Extensions, the Thing Descriptions use the `@context` mechanism known from JSON-LD [json-ld11]. When using TD Context Extensions, the value of `@context` of the Class `Thing` is an Array with additional elements of type `anyURI` identifying JSON-LD context files or Map containing namespace IRIs as defined in § 5.3.1.1 `Thing`.

The serialization rules for complex types in § 6.1 Mapping to JSON Types define the serialization of an extended `@context` name-value pair. A snippet with TD Context Extensions is given below:

EXAMPLE 27

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        {
            "eg": "http://example.org/iot#",
            "cov": "http://www.example.org/coap-binding#"
        },
        "https://schema.org/"
    ],
    ...
}
```

7.1 Semantic Annotations §

TD Context Extensions allow for additional Vocabulary Terms to a Thing Description instance. If the included namespaces are based on Class definitions such as those provided by the RDF Schema or OWL, they can be used to annotate any Class instance of a Thing Description semantically by associating the instance to a such an external Class definition. This is done by assigning a Class name to the `@type` name-value pair or including Class name in its Array value for multiple associations/annotations. Following the serialization rules in § 6.1 Mapping to JSON Types, `@type` is

either serialized as JSON string or as JSON array. `@type` is the JSON-LD keyword [json-ld11] used to set the type of a node.

TD Context Extensions also allow the inclusion of additional name-value pairs and well-defined values within any Class instance of a Thing Description. These pairs and values are defined through the included Vocabulary Terms and are serialized as additional members in the corresponding JSON objects or values of existing members, respectively. Examples are additional version metadata for the Thing or units of measure for data items.

As an example, the TD snippet given below extends the version information container by adding version numbers for the hardware and firmware of the Thing, and uses values from external Vocabularies for the Thing and for the data schema unit: SAREF, also used in Example 2, and OM, the Ontology of Units of Measure [RIJGERSBERG]. These Vocabularies are used as examples—others may exist, in particular in the home automation domain.

EXAMPLE 28

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        {
            "v": "http://www.example.org/versioningTerms#",
            "saref": "https://w3id.org/saref#",
            "om": "http://www.ontology-of-units-of-measure.org/resource/c
        }
    ],
    "version": {
        "instance": "1.2.1",
        "v:firmware": "0.9.1",
        "v:hardware": "1.0"
    },
    ...
    "@type": "saref:TemperatureSensor",
    "properties": {
        "temperature": {
            "description": "Temperature value of the weather station",
            "type": "number",
            "minimum": -32.5,
            "maximum": 55.2,
            "unit": "om:degree_Celsius",
            "forms": [...]
        },
        ...
    },
    ...
}
```

In many cases, TD Context Extensions may be used to annotate pieces of a data schema, to be able to semantically process the state information of the physical world object, which is represented by the data exchanged during an interaction (e.g., in the payload of a response). For example, a semantic description of this state information in RDF can be embedded in the TD Document and pieces of a data schema can be individually annotated as referring to specific parts of that RDF-modeled state of the physical world object.

The TD snippet below uses SAREF to describe the state of a lamp. The external Vocabulary Term `ssn:forProperty`, taken from SSN, the Semantic Sensor Network Ontology [VOCAB-SSN], is

being used to link the data schema of the `status` Property with the actual on/off state of the physical world object.

EXAMPLE 29

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        {
            "saref": "https://w3id.org/saref#",
            "ssn": "http://www.w3.org/ns/ssn/"
        }
    ],
    "id": "urn:dev:ops:32473-WoTLamp-1234",
    "@type": "saref:LightSwitch",
    "saref:hasState": {
        "@id": "urn:dev:ops:32473-WoTLamp-1234/state",
        "@type": "saref:OnOffState"
    },
    ...
    "properties": {
        "status": {
            "ssn:forProperty": "urn:dev:ops:32473-WoTLamp-1234/state",
            "type": "string",
            "forms": [{"href": "https://mylamp.example.com/status"}]
        },
        "fullStatus": {
            "ssn:forProperty": "urn:dev:ops:32473-WoTLamp-1234/state",
            "type": "object",
            "properties": {
                "statusString": { "type": "string" },
                "statusCode": { "type": "number" },
                "statusDescription": { "type": "string" }
            },
            "forms": [{"href": "https://mylamp.example.com/status?full=tr
        },
        ...
    },
    ...
}
```

In Example 2, the state of the Thing is given by the `status` affordance itself and possible state changes are given by the `toggle` affordance. In other words, the state of the physical world object directly provides the Interaction Affordances of the Thing. This design is satisfactory for simple cases. In more elaborate cases, however, several affordances may be available for the same physical state. In the example above, the `fullStatus` Property provides an alternative, more verbose representation for the state of the lamp.

7.2 Adding Protocol Bindings §

With the TD Context Extensions in a Thing Description, the communication metadata can be supplemented or new Protocol Bindings added through additional Vocabulary Terms serialized into JSON objects representing a `Form` instance. (see also § 8.3 Protocol Bindings).

The following TD example uses a fictional CoAP Protocol Binding, as no such Protocol Binding is available at the time of writing this specification. This TD Context Extension assumes that there is a *CoAP in RDF vocabulary* similar to *HTTP Vocabulary in RDF 1.0* [HTTP-in-RDF10] that is accessible via an example namespace `http://www.example.org/coap-binding#`. The supplemented `cov:methodName` member instructs the Consumer which CoAP method has to be applied (e.g., `GET` for the CoAP Method Code 0.01, `POST` for the CoAP Method Code 0.02, or `iPATCH` for CoAP Method Code 0.07).

EXAMPLE 30: Specialization of forms through TD Context Extension

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1",
        { "cov": "http://www.example.org/coap-binding#" }
    ],
    ...
    "properties": {
        "brightness": {
            "description": "The current brightness setting",
            "type": "integer",
            "minimum": -64,
            "maximum": 64,
            "forms": [{
                "op": "readproperty",
                "href": "coap://example.org:61616/api/brightness",
                "cov:methodName": "GET"
            }, {
                "op": "writeproperty",
                "href": "coap://example.org:61616/api/brightness",
                "cov:methodName": "POST"
            }]
        },
        ...
    },
    ...
}
```

7.3 Adding Security Schemes §

Finally, new security schemes that are not included in § 5.3.3 Security Vocabulary Definitions can be imported using the TD Context Extension mechanism. This example uses a fictional ACE security scheme based on [ACE] that is, for this example, defined by the namespace at `http://www.example.org/ace-security#`. Note that such additional security schemes must be Subclasses of the Class `SecurityScheme`.

EXAMPLE 31

```
{
    @context: [
        "https://www.w3.org/2019/wot/td/v1",
        {
            "cov": "http://www.example.org/coap-binding#",
            "ace": "http://www.example.org/ace-security#"
        }
    ],
    ...
    "securityDefinitions": {
        "ace_sc": {
            "scheme": "ace:ACESecurityScheme",
            ...
            "ace:as": "coaps://as.example.com/token",
            "ace:audience": "coaps://rs.example.com",
            "ace:scopes": ["limited", "special"],
            "ace:cnonce": true
        }
    },
    "security": ["ace_sc"],
    "properties": {
        "status": {
            ...
            "forms": [{
                "op": "readproperty",
                "href": "coaps://rs.example.com/status",
                "contentType": "application/cbor",
                "cov:methodName": "GET",
                "ace:scopes": ["limited"]
            }]
        }
    },
    "actions": {
        "configure": {
            ...
            "forms": [{
                "op": "invokeaction",
                "href": "coaps://rs.example.com/configure",
                "contentType": "application/cbor",
                "cov:methodName": "POST",
                "ace:scopes": ["special"]
            }]
```

```
            }
        },
        ...
}
```

Note that all security schemes defined in § 5.3.3 Security Vocabulary Definitions are already part of the TD context and need not to be included through a TD Context Extension.

8. Behavioral Assertions §

The following assertions relate to the behavior of components of a WoT system, as opposed to the representation or information model of the TD. However, note that TDs are descriptive, and may in particular be used to describe pre-existing network interfaces. In these cases, assertions cannot be made that constrain the behavior of such pre-existing interfaces. Instead, the assertions must be interpreted as constraints on the TD to accurately represent such interfaces.

8.1 Security Configurations §

To enable secure interoperation, security configurations must accurately reflect the requirements of the Thing:

- If a Thing requires a specific access mechanism for an interaction, that mechanism *MUST* be specified in the security configuration of the Thing Description.
- If a Thing does not require a specific access mechanism for an interaction, that mechanism *MUST NOT* be specified in the security configuration of the Thing Description.

Some security protocols may ask for authentication information dynamically, including required encoding or encryption schemes. One consequence of the above is that if a protocol asks for a form of security credentials or an encoding or encryption scheme not declared in the Thing Description then the Thing Description is to be considered invalid.

8.2 Data Schemas §

The data schemas provided in the TD should accurately represent the data payloads returned and accepted by the described Thing in the interactions specified in the TD. In general, Consumers should follow the data schemas strictly, not generating anything not given in the WoT Thing Description, but

should accept additional data from the Thing not given explicitly in the WoT Thing Description. In general, Things are *described* by WoT Thing Descriptions, but Consumers are *constrained* to follow WoT Thing Descriptions when interacting with Things.

- A Thing acting as a Consumer when interacting with another target Thing described in a WoT Thing Description *MUST* generate data organized according to the data schemas given in the corresponding interactions.
- A WoT Thing Description *MUST* accurately describe the data returned and accepted by each interaction.
- A Thing *MAY* return additional data from an interaction even when such data is not described in the data schemas given in its WoT Thing Description. This applies to `ObjectSchema` and `ArraySchema` (when `items` is an Array of `DataSchema`) where there can be additional properties or items in the data returned. This behaves as if `"additionalProperties":true` or `"additionalItems":true` as defined in [JSON-SCHEMA].
- A Thing acting as a Consumer when interacting with another Thing *MUST* accept without error any additional data not described in the data schemas given in the Thing Description of the target Thing. This applies to `ObjectSchema` and `ArraySchema` (when `items` is an Array of `DataSchema`) where there can be additional properties or items in the data returned. This behaves as if `"additionalProperties":true` or `"additionalItems":true` as defined in [JSON-SCHEMA].
- A Thing acting as a Consumer when interacting with another Thing *MUST NOT* generate data not described in the data schemas given in the Thing Description of that Thing.
- A Thing acting as a Consumer when interacting with another Thing *MUST* generate URIs according to the URI Templates, base URIs, and form href parameters given in the Thing Description of the target Thing.
- URI Templates, base URIs, and href members in a WoT Thing Description *MUST* accurately describe the WoT Interface of the Thing.

8.3 Protocol Bindings §

A Protocol Binding is the mapping from an Interaction Affordance to concrete messages of a specific protocol such as HTTP [RFC7231], CoAP [RFC7252], or MQTT [MQTT]. Protocol Bindings of Interaction Affordances are serialized as `forms` as defined in § 6.3.9 `forms`.

Every form in a WoT Thing Description must have a submission target, given by the `href` member. The URI scheme of this submission target indicates what Protocol Binding the Thing implements [WOT-ARCHITECTURE]. For instance, if the target starts with `http` or `https`, a Consumer can then

infer the Thing implements the Protocol Binding based on HTTP and it should expect HTTP-specific terms in the form instance (see next section, § 8.3.1 Protocol Binding based on HTTP).

- Every form in a WoT Thing Description *MUST* follow the requirements of the Protocol Binding indicated by the URI scheme of its `href` member.
- Every form in a WoT Thing Description *MUST* accurately describe requests (including request headers, if present) accepted by the Thing in an interaction.

8.3.1 Protocol Binding based on HTTP §

Per default the Thing Description supports the Protocol Binding based on HTTP by including the HTTP RDF vocabulary definitions from *HTTP Vocabulary in RDF 1.0* [HTTP-in-RDF10]. This vocabulary can be directly used within TD instances by the usage of the prefix `htv`, which points to `http://www.w3.org/2011/http#`. Further details of Protocol Binding based on HTTP can be found in [WOT-BINDING-TEMPLATES].

To interact with a Thing that implements the Protocol Binding based on HTTP, a Consumer needs to know what HTTP method to use when submitting a form. In the general case, a Thing Description can explicitly include a term indicating the method, i.e., `htv:methodName`. For the sake of conciseness, the Protocol Binding based on HTTP defines Default Values for the operation types listed below, which also aims at convergence of the methods expected by Things (e.g., GET to read, PUT to write). When no method is indicated in a form representing an Protocol Binding based on HTTP, a Default Value *MUST* be assumed as shown in the following table.

Vocabulary term	*Default value*	*Context*
`htv:methodName`	`GET`	`Form` with operation type `readproperty`, `readallproperties`, `readmultipleproperties`
`htv:methodName`	`PUT`	`Form` with operation type `writeproperty`, `writeallproperties`, `writemultipleproperties`
`htv:methodName`	`POST`	`Form` with operation type `invokeaction`

For example, the Example 1 in § 1. Introduction does not contain operation types and HTTP methods in the forms. The following Default Values should be assumed for the forms in the Example 1:

EXAMPLE 32

☐ *with default values for Protocol Binding based on HTTP*

8.3.2 Other Protocol Bindings §

The number of Protocol Bindings a Thing can implement is not restricted. Other Protocol Bindings (e.g., for CoAP, MQTT, or OPC UA) are intended to be standardized in separate documents such as a protocol Vocabulary similar to *HTTP Vocabulary in RDF 1.0* [HTTP-in-RDF10] or specifications including Default Value definitions. Such protocols can be simply integrated into the TD by the usage of the TD Context Extension mechanism (see § 7. TD Context Extensions).

Please refer to [WOT-BINDING-TEMPLATES] for information on how to describe IoT platforms and ecosystems.

9. Security and Privacy Considerations §

This section is non-normative.

In general the security measures taken to protect a WoT system will depend on the threats and attackers that system may face and the value of the assets needs to protect. In addition privacy risks will depend on the association of Things with identifiable people and both the direct information and the inferred information available from such an association. A detailed discussion of security and privacy considerations for the Web of Things, including a threat model that can be adapted to various circumstances, is presented in the informative document [WOT-SECURITY-GUIDELINES]. This section discusses only security and privacy risks and possible mitigations directly relevant to the WoT Thing Description.

A WoT Thing Description can describe both secure and insecure network interfaces. When a Thing Description is retro-fitted to an existing network interface, no change in the security status of the network interface is to be expected.

The use of a WoT Thing Description introduces the security and privacy risks given in the following sections. After each risk, we suggest some possible mitigations.

9.1 Context Fetching Privacy Risk §

Fetching the vocabulary files given in the `@context` member of any JSON-LD [json-ld11] document can be a privacy risk. In the case of the WoT, an attacker can observe the network traffic produced by such fetches and can use the metadata of the fetch, such as the destination IP address, to infer information about the device especially if domain-specific vocabularies are used. This is a risk even if the connection is encrypted, and is related to DNS privacy leaks.

Mitigation:

Avoid actual fetching of vocabulary files. Vocabulary files should be cached whenever possible. Ideally they would be made immutable, built into the interpreting device, and not fetched at all, with the URI in the `@context` member serving only as an identifier of the (known) vocabulary. This requires the use of strict version control, as updates should use a new URI to ensure that existing URIs can refer to immutable data. Use well-known standard vocabulary files whenever possible to improve the chances that the context file will be available locally to systems interpreting the metadata in a Thing Description.

9.2 Immutable Identifiers Privacy Risk §

A Thing Description containing an identifier (`id`) may describe a Thing that is associated with an identifiable person. Such identifiers pose various risks including tracking. However, if the identifier is also immutable, then the tracking risk is amplified, since a device may be sold or given to another person and the known ID used to track that person.

Mitigation:

All identifiers should be mutable, and there should be a mechanism to update the `id` of a Thing. Specifically, the `id` of a Thing should not be fixed in hardware. This does, however, conflict with the Linked Data ideal that identifiers are fixed URIs. In many circumstances it will be acceptable to only allow updates to identifiers if a Thing is reinitialized. In this case as a software entity the old Thing ceases to exist and a new Thing is created. This can be sufficient to break a tracking chain when, for example, a device is sold to a new owner. Alternatively, if more frequent changes are desired during the operational phase of a device, a mechanism can be put into place to notify only authorized users of the change in identifier when a change is made. Note however that some classes of devices, e.g., medical devices, may require immutable IDs by law in some jurisdictions. In this case extra attention should be paid to secure access to files, such as Thing Descriptions, containing such immutable identifiers. It may also be desirable to not share the "true" immutable identifier in such a case in the TD whenever possible.

9.3 Fingerprinting Privacy Risk §

As noted above, the `id` member in a TD can pose a privacy risk. However, even if the `id` is updated as described to mitigate its tracking risk, it may still be possible to associate a TD with a particular physical device, and from there to an identifiable person, through fingerprinting.

Even if a specific device instance cannot be identified through fingerprinting, it may be possible to infer the type of a device from the information in the TD, such as the set of interactions, and use this type to infer private information about an identifiable person, such as a medical condition.

Mitigation:

Only authorized users should be provided access to the Thing Description for a Thing, and only the amount of information needed for the level of authorization and the use case should be provided. If the TD is only distributed to authorized users through secure and confidential channels, for example through a directory service that requires authentication, then external unauthorized parties will not have access to the TD to fingerprint it. To further mitigate this risk, information not necessary for a particular use case of a TD should be omitted whenever possible. For example, for an ad-hoc connection to a device where the Consumer does not store state about the Thing, the `id` can be omitted. If the Consumer does not need certain interactions for its use case, they can be omitted. If the Consumer is not authorized to use certain interactions, they can likewise be omitted. If the Consumer does not have any capability to display human-readable information such as titles or descriptions, they can be omitted or replaced with zero-length strings.

9.4 Globally Unique Identifier Privacy Risk §

Globally unique identifiers pose a privacy risk if a centralized authority is needed to create and distribute them, since then a third party has knowledge of the identifiers.

Mitigation:

The `id` field in TDs are intentionally not required to be globally unique. There are several cryptographic mechanisms available to generate suitable IDs in a distributed fashion that do not require a central registry. These mechanisms typically have a very low probability of generating duplicate identifiers, and this needs to be taken into account in the system design; for example, by detecting duplicates and regenerating IDs when necessary. The scope of IDs also does not need to be global: it is acceptable to use identifiers that only distinguish Things in a certain context, such as within a home or factory.

9.5 TD Interception and Tampering Security Risk §

Intercepting and tampering with TDs can be used to launch man-in-the-middle attacks, for example by rewriting URLs in TDs to redirect accesses to a malicious intermediary that can capture or manipulate data.

Mitigation:

Obtain Thing Descriptions only through mutually authenticated channels. This ensures that the Consumer and the server are both sure of the identity of the other party to the communication. This is also necessary in order to deliver TDs only to authorized users.

9.6 Context Interception and Tampering Security Risk §

Intercepting and tampering with context files can be used to facilitate attacks by modifying the interpretation of vocabulary.

Mitigation:

Ideally context files would only be obtained through authenticated channels but it is notable (and unfortunate) that many contexts are indicated using HTTP URLs, which are vulnerable to interception and modification if dereferenced. However, if context files are immutable and cached, and dereferencing is avoided whenever possible, then this risk can be reduced.

9.7 Inferencing of Personally Identifiable Information Privacy Risk §

In many locales, in order to protect the privacy of users, there are legal requirements for the handling of personally identifiable information, that is, information that can be associated with a particular person. Such information can of course be generated by IoT devices directly. However, the existence and metadata of IoT devices (the kind of data stored in a Thing Description) can also contain or be used to infer personally identifiable information. This information can be as simple as the fact that a certain person owns a certain type of device, which can lead to additional inferences about that person.

Mitigation:

Treat a Thing Description associated with a personal device as if it contained personally identifiable information. As an example application of this principle, consider how to obtain user consent. Consent for usage of personally identifiable data generated by a Thing is often obtained when a Thing is paired with system consuming the data, which is frequently also when the Thing Description is registered with a local directory or the system consuming the Thing Description in order to access the device. In this case, consent for using data from a Thing can be combined with consent for accessing the Thing Description of the Thing. As a second example, if we consider a TD to contain personally identifiable information, then it should not be retained indefinitely or used for purposes other than those for which consent was given.

10. IANA Considerations §

10.1 `application/td+json` Media Type Registration §

Type name:
application

Subtype name:
td+json

Required parameters:
None

Optional parameters:
None

Encoding considerations:
See RFC 6839, section 3.1.

Security considerations:
See RFC 8259, section 12.

Since WoT Thing Description is intended to be a pure data exchange format for Thing metadata, the serialization *SHOULD NOT* be passed through a code execution mechanism such as JavaScript's `eval()` function to be parsed. An (invalid) document may contain code that, when executed, could lead to unexpected side effects compromising the security of a system.

WoT Thing Descriptions can be evaluated with a JSON-LD 1.1 processor, which typically follows links to remote contexts (i.e., TD context extensions, see W3C WoT Thing Description, section 7) automatically, resulting in the transfer of files without the explicit request of the Consumer for each one. If remote contexts are served by third parties, it may allow them to gather usage patterns or similar information leading to privacy concerns. While implementations on resource-constrained devices are expected to perform raw JSON processing (as opposed to JSON-LD processing), implementations in general *SHOULD* statically cache vetted versions of their supported context extensions and not to follow links to remote contexts. Supported context extensions can be managed through a secure software update mechanism instead.

Context Extensions (see W3C WoT Thing Description, section 7) that are loaded from the Web over non-secure connections, such as HTTP, run the risk of being altered by an attacker such that they may modify the TD Information Model in a way that could compromise security. For this reason, Consumer again *SHOULD* vet and cache remote contexts before allowing the system to use it.

Given that JSON-LD processing usually includes the substitution of long IRIs [RFC3987] with short terms, WoT Thing Descriptions may expand considerably when processed using a JSON-LD 1.1 processor and, in the worst case, the resulting data might consume all of the recipient's resources. Consumers *SHOULD* treat any TD metadata with due skepticism.

Interoperability considerations:
See RFC 8259.

Rules for processing both conforming and non-conforming content are defined in this specification.

Published specification:
https://w3c.github.io/wot-thing-description

Applications that use this media type:
All participating entities in the W3C Web of Things, that is, Things, Consumers, and Intermediaries as defined in the Web of Things (WoT) Architecture.

Fragment identifier considerations:
See RFC 6839, section 3.1.

Additional information:

Magic number(s):
Not Applicable

File extension(s):
.jsontd

Macintosh file type code(s):
TEXT

Person & email address to contact for further information:
Matthias Kovatsch <w3c@kovatsch.net>

Intended usage:
COMMON

Restrictions on usage:
None

Author(s):
The WoT Thing Description specification is a product of the Web of Things Working Group.

Change controller:
W3C

10.2 CoAP Content-Format Registration §

IANA assigns compact CoAP Content-Format IDs for media types in the CoAP Content-Formats subregistry within the Constrained RESTful Environments (CoRE) Parameters registry [RFC7252]. The Content-Format ID for WoT Thing Description is 432.

Media Type:
application/td+json

Encoding:
-

ID:
432

Reference:
["Web of Things (WoT) Thing Description", May 2019]

A. Example Thing Description Instances §

This section is non-normative.

A.1 MyLampThing Example with CoAP Protocol Binding §

Feature list of the Thing:

- Title: MyLampThing
- Context Extensions: none
- Offered affordances: 1 Property, 1 Action, 1 Event
- Security: PSKSecurityScheme
- Protocol Binding: CoAP [RFC7252] over TLS
- Comment: Also see § 7.2 Adding Protocol Bindings.

EXAMPLE 33: MyLampThing with CoAP Protocol Binding

```
{
   "@context": [
     "https://www.w3.org/2019/wot/td/v1",
     {
     "cov": "http://www.example.org/coap-binding#"
     }
    ],
    "id": "urn:dev:ops:32473-WoTLamp-1234",
    "title": "MyLampThing",
    "description" : "MyLampThing uses JSON serialization",
    "securityDefinitions": {"psk_sc":{"scheme": "psk"}},
    "security": ["psk_sc"],
    "properties": {
        "status": {
            "description" : "Shows the current status of the lamp",
            "type": "string",
            "forms": [{
                "op": "readproperty",
                "href": "coaps://mylamp.example.com/status",
                "cov:methodName" : "GET"
            }]
        }
    },
    "actions": {
        "toggle": {
            "description" : "Turn on or off the lamp",
            "forms": [{
                "href": "coaps://mylamp.example.com/toggle",
                "cov:methodName" : "POST"
            }]
        }
    },
    "events": {
        "overheating": {
            "description" : "Lamp reaches a critical temperature (overhea
            "data": {"type": "string"},
            "forms": [{
                "href": "coaps://mylamp.example.com/oh",
                "cov:methodName" : "GET",
                "subprotocol" : "cov:observe"
            }]
```

```
            }
        }
    }
```

A.2 MyIlluminanceSensor Example with MQTT Protocol Binding §

Feature list of the Thing:

- Title: MyIlluminanceSensor
- Context Extensions: none
- Offered affordances: 1 Event
- Security: none
- Protocol Binding: MQTT [MQTT]
- Comment: An MQTT client frequently publishes the illuminance data (number is serialized in text format) to the topic `/illuminance` by the MQTT broker running behind the address 192.168.1.187:1883.

EXAMPLE 34: MyIlluminanceSensor with MQTT Protocol Binding

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "title": "MyIlluminanceSensor",
    "id": "urn:dev:ops:32473-WoTIlluminanceSensor-1234",
    "securityDefinitions": {"nosec_sc": {"scheme": "nosec"}},
    "security": ["nosec_sc"],
    "events": {
        "illuminance": {
            "data":{"type": "integer"},
            "forms": [
                {
                    "href": "mqtt://192.168.1.187:1883/illuminance",
                    "contentType" : "text/plain",
                    "op" : "subscribeevent"
                }
            ]
        }
    }
}
```

A.3 Webhook Event Example §

Feature list of the Thing:

- Title: WebhookThing
- Context Extensions: use HTTP Protocol Binding supplements (htv prefix already included in TD context)
- Offered affordances: 1 Event
- Security: none
- Protocol Binding: HTTP
- Comment: *WebhookThing* provides an Event affordance `temperature` which periodically pushes the latest temperature value to the Consumer using a Webhook mechanism, where the Thing sends POST requests to a callback URI provided by the Consumer. To describe this, the `subscription` member defines a write-only parameter `callbackURL`, which must be submitted through the `subscribeevent` form. The read-only parameter `subscriptionID` is returned by the subscription. The *WebhookThing* will then periodically POST to this callback URI

with a payload defined by `data`. To unsubscribe, the Consumer has to submit the `unsubscribeevent` form, which makes use of a URI Template. The `uriVariables` member informs the Consumer to include the `subscriptionID` string. This can be further automated by using a TD Context Extension to include proper semantic annotations. Alternatively, one can imagine unsubscribing using the `cancellation` member similarly to `subscription` and combine this with a `unsubscribeevent` form that describes a POST request with payload to unsubscribe.

EXAMPLE 35: Temperature Event with subscription and cancellation

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "id": "urn:dev:ops:32473-Thing-1234",
    "title": "WebhookThing",
    "description": "Webhook-based Event with subscription and unsubscribe
    "securityDefinitions": {"nosec_sc": {"scheme": "nosec"}},
    "security": ["nosec_sc"],
    "events": {
        "temperature": {
            "description": "Provides periodic temperature value updates."
            "subscription": {
                "type": "object",
                "properties": {
                    "callbackURL": {
                        "type": "string",
                        "format": "uri",
                        "description": "Callback URL provided by subscrib
                        "writeOnly": true
                    },
                    "subscriptionID": {
                        "type": "string",
                        "description": "Unique subscription ID for cancel
                        "readOnly": true
                    }
                }
            },
            "data": {
                "type": "number",
                "description": "Latest temperature value that is sent to
            },
            "cancellation": {
                "type": "object",
                "properties": {
                    "subscriptionID": {
                        "type": "integer",
                        "description": "Required subscription ID to cance
                        "writeOnly": true
                    }
                }
            },
            "uriVariables": {
                "subscriptionID": { "type": "string" }
```

```
            },
            "forms": [
                {
                    "op": "subscribeevent",
                    "href": "http://192.168.0.124:8080/events/temp/subscr
                    "contentType": "application/json",
                    "htv:methodName": "POST"
                },
                {
                    "op": "unsubscribeevent",
                    "href": "http://192.168.0.124:8080/events/temp/{subsc
                    "htv:methodName": "DELETE"
                }
            ]
        }
    }
}
```

B. JSON Schema for TD Instance Validation §

This section is non-normative.

Below is a JSON Schema [JSON-SCHEMA] document for syntactically validating Thing Description instances serialized in JSON based format.

NOTE

The Thing Description defined by this document allows for adding external vocabularies by using `@context` mechanism known from JSON-LD [json-ld11], and the terms in those external vocabularies can be used in addition to the terms defined in § 5. TD Information Model. For this reason, the below JSON schema is intentionally non-strict in that regard. You can replace the value of `additionalProperties` schema property `true` with `false` in different scopes/levels in order to perform a stricter validation in case no external vocabularies are used.

NOTE

Please note that some JSON Schema validation tools do not support the `iri` string format.

The following JSON Schema for validating TD instances does not require the terms with Default Values to be present. Thus the terms with Default Values are optional. (see also § 5.4 Default Value Definitions)

```
{
"title": "WoT TD Schema - 16 October 2019",
"description": "JSON Schema for validating TD instances against the
"$schema ": "http://json-schema.org/draft-07/schema#",
"definitions": {
    "anyUri": {
        "type": "string",
        "format": "iri-reference"
    },
    "description": {
        "type": "string"
    },
    "descriptions": {
        "type": "object",
        "additionalProperties": {
            "type": "string"
        }
    },
    "title": {
        "type": "string"
    },
    "titles": {
        "type": "object",
        "additionalProperties": {
            "type": "string"
        }
    },
    "security": {
        "oneOf": [{
                "type": "array",
                "items": {
                    "type": "string"
                }
            },
            {
                "type": "string"
            }
        ]
    },
    "scopes": {
        "oneOf": [{
                "type": "array",
                "items": {
                    "type": "string"
```

```
                    }
                },
                {
                    "type": "string"
                }
            ]
        },
        "subprotocol": {
            "type": "string",
            "enum": [
                "longpoll",
                "websub",
                "sse"
            ]
        },
        "thing-context-w3c-uri": {
            "type": "string",
            "enum": [
                "https://www.w3.org/2019/wot/td/v1"
            ]
        },
        "thing-context": {
            "oneOf": [{
                    "type": "array",
                    "items": [{
                        "$ref": "#/definitions/thing-context-w3c-uri"
                    }],
                    "additionalItems": {
                        "anyOf": [{
                                "$ref": "#/definitions/anyUri"
                            },
                            {
                                "type": "object"
                            }
                        ]
                    }
                },
                {
                    "$ref": "#/definitions/thing-context-w3c-uri"
                }
            ]
        },
        "type_declaration": {
            "oneOf": [{
```

```
                    "type": "string"
                },
                {
                    "type": "array",
                    "items": {
                        "type": "string"
                    }
                }
            ]
        },
        "dataSchema": {
            "type": "object",
            "properties": {
                "@type": {
                    "$ref": "#/definitions/type_declaration"
                },
                "description": {
                    "$ref": "#/definitions/description"
                },
                "title": {
                    "$ref": "#/definitions/title"
                },
                "descriptions": {
                    "$ref": "#/definitions/descriptions"
                },
                "titles": {
                    "$ref": "#/definitions/titles"
                },
                "writeOnly": {
                    "type": "boolean"
                },
                "readOnly": {
                    "type": "boolean"
                },
                "oneOf": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/dataSchema"
                    }
                },
                "unit": {
                    "type": "string"
                },
                "enum": {
```

```
                    "type": "array",
                    "minItems": 1,
                    "uniqueItems": true
                },
                "format": {
                    "type": "string"
                },
                "const": {},
                "type": {
                    "type": "string",
                    "enum": [
                        "boolean",
                        "integer",
                        "number",
                        "string",
                        "object",
                        "array",
                        "null"
                    ]
                },
                "items": {
                    "oneOf": [{
                            "$ref": "#/definitions/dataSchema"
                        },
                        {
                            "type": "array",
                            "items": {
                                "$ref": "#/definitions/dataSchema"
                            }
                        }
                    ]
                },
                "maxItems": {
                    "type": "integer",
                    "minimum": 0
                },
                "minItems": {
                    "type": "integer",
                    "minimum": 0
                },
                "minimum": {
                    "type": "number"
                },
                "maximum": {
```

```
                "type": "number"
            },
            "properties": {
                "additionalProperties": {
                    "$ref": "#/definitions/dataSchema"
                }
            },
            "required": {
                "type": "array",
                "items": {
                    "type": "string"
                }
            }
        }
    },
    "form_element_property": {
        "type": "object",
        "properties": {
            "op": {
                "oneOf": [{
                        "type": "string",
                        "enum": [
                            "readproperty",
                            "writeproperty",
                            "observeproperty",
                            "unobserveproperty"
                        ]
                    },
                    {
                        "type": "array",
                        "items": {
                            "type": "string",
                            "enum": [
                                "readproperty",
                                "writeproperty",
                                "observeproperty",
                                "unobserveproperty"
                            ]
                        }
                    }
                ]
            },
            "href": {
                "$ref": "#/definitions/anyUri"
```

```
                },
                "contentType": {
                    "type": "string"
                },
                "contentCoding": {
                    "type": "string"
                },
                "subprotocol": {
                    "$ref": "#/definitions/subprotocol"
                },
                "security": {
                    "$ref": "#/definitions/security"
                },
                "scopes": {
                    "$ref": "#/definitions/scopes"
                },
                "response": {
                    "type": "object",
                    "properties": {
                        "contentType": {
                            "type": "string"
                        }
                    }
                }
            },
            "required": [
                "href"
            ],
            "additionalProperties": true
        },
        "form_element_action": {
            "type": "object",
            "properties": {
                "op": {
                    "oneOf": [{
                            "type": "string",
                            "enum": [
                                "invokeaction"
                            ]
                        },
                        {
                            "type": "array",
                            "items": {
                                "type": "string",
```

```
                            "enum": [
                                "invokeaction"
                            ]
                        }
                    }
                ]
            },
            "href": {
                "$ref": "#/definitions/anyUri"
            },
            "contentType": {
                "type": "string"
            },
            "contentCoding": {
                "type": "string"
            },
            "subprotocol": {
                "$ref": "#/definitions/subprotocol"
            },
            "security": {
                "$ref": "#/definitions/security"
            },
            "scopes": {
                "$ref": "#/definitions/scopes"
            },
            "response": {
                "type": "object",
                "properties": {
                    "contentType": {
                        "type": "string"
                    }
                }
            }
        },
        "required": [
            "href"
        ],
        "additionalProperties": true
    },
    "form_element_event": {
        "type": "object",
        "properties": {
            "op": {
                "oneOf": [{
```

```
                    "type": "string",
                    "enum": [
                        "subscribeevent",
                        "unsubscribeevent"
                    ]
                },
                {
                    "type": "array",
                    "items": {
                        "type": "string",
                        "enum": [
                            "subscribeevent",
                            "unsubscribeevent"
                        ]
                    }
                }
            ]
        },
        "href": {
            "$ref": "#/definitions/anyUri"
        },
        "contentType": {
            "type": "string"
        },
        "contentCoding": {
            "type": "string"
        },
        "subprotocol": {
            "$ref": "#/definitions/subprotocol"
        },
        "security": {
            "$ref": "#/definitions/security"
        },
        "scopes": {
            "$ref": "#/definitions/scopes"
        },
        "response": {
            "type": "object",
            "properties": {
                "contentType": {
                    "type": "string"
                }
            }
        }
```

```
            },
            "required": [
                "href"
            ],
            "additionalProperties": true
        },
        "form_element_root": {
            "type": "object",
            "properties": {
                "op": {
                    "oneOf": [{
                            "type": "string",
                            "enum": [
                                "readallproperties",
                                "writeallproperties",
                                "readmultipleproperties",
                                "writemultipleproperties"
                            ]
                        },
                        {
                            "type": "array",
                            "items": {
                                "type": "string",
                                "enum": [
                                    "readallproperties",
                                    "writeallproperties",
                                    "readmultipleproperties",
                                    "writemultipleproperties"
                                ]
                            }
                        }
                    ]
                },
                "href": {
                    "$ref": "#/definitions/anyUri"
                },
                "contentType": {
                    "type": "string"
                },
                "contentCoding": {
                    "type": "string"
                },
                "subprotocol": {
                    "$ref": "#/definitions/subprotocol"
```

```
                },
                "security": {
                    "$ref": "#/definitions/security"
                },
                "scopes": {
                    "$ref": "#/definitions/scopes"
                },
                "response": {
                    "type": "object",
                    "properties": {
                        "contentType": {
                            "type": "string"
                        }
                    }
                }
            },
            "required": [
                "href"
            ],
            "additionalProperties": true
        },
        "property_element": {
            "type": "object",
            "properties": {
                "@type": {
                    "$ref": "#/definitions/type_declaration"
                },
                "description": {
                    "$ref": "#/definitions/description"
                },
                "descriptions": {
                    "$ref": "#/definitions/descriptions"
                },
                "title": {
                    "$ref": "#/definitions/title"
                },
                "titles": {
                    "$ref": "#/definitions/titles"
                },
                "forms": {
                    "type": "array",
                    "minItems": 1,
                    "items": {
                        "$ref": "#/definitions/form_element_property"
```

```
            }
        },
        "uriVariables": {
            "type": "object",
            "additionalProperties": {
                "$ref": "#/definitions/dataSchema"
            }
        },
        "observable": {
            "type": "boolean"
        },
        "writeOnly": {
            "type": "boolean"
        },
        "readOnly": {
            "type": "boolean"
        },
        "oneOf": {
            "type": "array",
            "items": {
                "$ref": "#/definitions/dataSchema"
            }
        },
        "unit": {
            "type": "string"
        },
        "enum": {
            "type": "array",
            "minItems": 1,
            "uniqueItems": true
        },
        "format": {
            "type": "string"
        },
        "const": {},
        "type": {
            "type": "string",
            "enum": [
                "boolean",
                "integer",
                "number",
                "string",
                "object",
                "array",
```

```
                    "null"
                ]
            },
            "items": {
                "oneOf": [{
                        "$ref": "#/definitions/dataSchema"
                    },
                    {
                        "type": "array",
                        "items": {
                            "$ref": "#/definitions/dataSchema"
                        }
                    }
                ]
            },
            "maxItems": {
                "type": "integer",
                "minimum": 0
            },
            "minItems": {
                "type": "integer",
                "minimum": 0
            },
            "minimum": {
                "type": "number"
            },
            "maximum": {
                "type": "number"
            },
            "properties": {
                "additionalProperties": {
                    "$ref": "#/definitions/dataSchema"
                }
            },
            "required": {
                "type": "array",
                "items": {
                    "type": "string"
                }
            }
        },
        "required": [
            "forms"
        ],
```

```
            "additionalProperties": true
        },
        "action_element": {
            "type": "object",
            "properties": {
                "@type": {
                    "$ref": "#/definitions/type_declaration"
                },
                "description": {
                    "$ref": "#/definitions/description"
                },
                "descriptions": {
                    "$ref": "#/definitions/descriptions"
                },
                "title": {
                    "$ref": "#/definitions/title"
                },
                "titles": {
                    "$ref": "#/definitions/titles"
                },
                "forms": {
                    "type": "array",
                    "minItems": 1,
                    "items": {
                        "$ref": "#/definitions/form_element_action"
                    }
                },
                "uriVariables": {
                    "type": "object",
                    "additionalProperties": {
                        "$ref": "#/definitions/dataSchema"
                    }
                },
                "input": {
                    "$ref": "#/definitions/dataSchema"
                },
                "output": {
                    "$ref": "#/definitions/dataSchema"
                },
                "safe": {
                    "type": "boolean"
                },
                "idempotent": {
                    "type": "boolean"
```

```
                }
            },
            "required": [
                "forms"
            ],
            "additionalProperties": true
        },
        "event_element": {
            "type": "object",
            "properties": {
                "@type": {
                    "$ref": "#/definitions/type_declaration"
                },
                "description": {
                    "$ref": "#/definitions/description"
                },
                "descriptions": {
                    "$ref": "#/definitions/descriptions"
                },
                "title": {
                    "$ref": "#/definitions/title"
                },
                "titles": {
                    "$ref": "#/definitions/titles"
                },
                "forms": {
                    "type": "array",
                    "minItems": 1,
                    "items": {
                        "$ref": "#/definitions/form_element_event"
                    }
                },
                "uriVariables": {
                    "type": "object",
                    "additionalProperties": {
                        "$ref": "#/definitions/dataSchema"
                    }
                },
                "subscription": {
                    "$ref": "#/definitions/dataSchema"
                },
                "data": {
                    "$ref": "#/definitions/dataSchema"
                },
```

```
                "cancellation": {
                    "$ref": "#/definitions/dataSchema"
                }
            },
            "required": [
                "forms"
            ],
            "additionalProperties": true
        },
        "link_element": {
            "type": "object",
            "properties": {
                "href": {
                    "$ref": "#/definitions/anyUri"
                },
                "type": {
                    "type": "string"
                },
                "rel": {
                    "type": "string"
                },
                "anchor": {
                    "$ref": "#/definitions/anyUri"
                }
            },
            "required": [
                "href"
            ],
            "additionalProperties": true
        },
        "securityScheme": {
            "oneOf": [{
                    "type": "object",
                    "properties": {
                        "@type": {
                            "$ref": "#/definitions/type_declaration"
                        },
                        "description": {
                            "$ref": "#/definitions/description"
                        },
                        "descriptions": {
                            "$ref": "#/definitions/descriptions"
                        },
                        "proxy": {
```

```
                        "$ref": "#/definitions/anyUri"
                    },
                    "scheme": {
                        "type": "string",
                        "enum": [
                            "nosec"
                        ]
                    }
                },
                "required": [
                    "scheme"
                ]
            },
            {
                "type": "object",
                "properties": {
                    "@type": {
                        "$ref": "#/definitions/type_declaration"
                    },
                    "description": {
                        "$ref": "#/definitions/description"
                    },
                    "descriptions": {
                        "$ref": "#/definitions/descriptions"
                    },
                    "proxy": {
                        "$ref": "#/definitions/anyUri"
                    },
                    "scheme": {
                        "type": "string",
                        "enum": [
                            "basic"
                        ]
                    },
                    "in": {
                        "type": "string",
                        "enum": [
                            "header",
                            "query",
                            "body",
                            "cookie"
                        ]
                    },
                    "name": {
```

```
                        "type": "string"
                    }
                },
                "required": [
                    "scheme"
                ]
            },
            {
                "type": "object",
                "properties": {
                    "@type": {
                        "$ref": "#/definitions/type_declaration"
                    },
                    "description": {
                        "$ref": "#/definitions/description"
                    },
                    "descriptions": {
                        "$ref": "#/definitions/descriptions"
                    },
                    "proxy": {
                        "$ref": "#/definitions/anyUri"
                    },
                    "scheme": {
                        "type": "string",
                        "enum": [
                            "digest"
                        ]
                    },
                    "qop": {
                        "type": "string",
                        "enum": [
                            "auth",
                            "auth-int"
                        ]
                    },
                    "in": {
                        "type": "string",
                        "enum": [
                            "header",
                            "query",
                            "body",
                            "cookie"
                        ]
                    },
```

```
                    "name": {
                        "type": "string"
                    }
                },
                "required": [
                    "scheme"
                ]
            },
            {
                "type": "object",
                "properties": {
                    "@type": {
                        "$ref": "#/definitions/type_declaration"
                    },
                    "description": {
                        "$ref": "#/definitions/description"
                    },
                    "descriptions": {
                        "$ref": "#/definitions/descriptions"
                    },
                    "proxy": {
                        "$ref": "#/definitions/anyUri"
                    },
                    "scheme": {
                        "type": "string",
                        "enum": [
                            "apikey"
                        ]
                    },
                    "in": {
                        "type": "string",
                        "enum": [
                            "header",
                            "query",
                            "body",
                            "cookie"
                        ]
                    },
                    "name": {
                        "type": "string"
                    }
                },
                "required": [
                    "scheme"
```

```
                ]
            },
            {
                "type": "object",
                "properties": {
                    "@type": {
                        "$ref": "#/definitions/type_declaration"
                    },
                    "description": {
                        "$ref": "#/definitions/description"
                    },
                    "descriptions": {
                        "$ref": "#/definitions/descriptions"
                    },
                    "proxy": {
                        "$ref": "#/definitions/anyUri"
                    },
                    "scheme": {
                        "type": "string",
                        "enum": [
                            "bearer"
                        ]
                    },
                    "authorization": {
                        "$ref": "#/definitions/anyUri"
                    },
                    "alg": {
                        "type": "string"
                    },
                    "format": {
                        "type": "string"
                    },
                    "in": {
                        "type": "string",
                        "enum": [
                            "header",
                            "query",
                            "body",
                            "cookie"
                        ]
                    },
                    "name": {
                        "type": "string"
                    }
```

```
                    },
                    "required": [
                        "scheme"
                    ]
                },
                {
                    "type": "object",
                    "properties": {
                        "@type": {
                            "$ref": "#/definitions/type_declaration"
                        },
                        "description": {
                            "$ref": "#/definitions/description"
                        },
                        "descriptions": {
                            "$ref": "#/definitions/descriptions"
                        },
                        "proxy": {
                            "$ref": "#/definitions/anyUri"
                        },
                        "scheme": {
                            "type": "string",
                            "enum": [
                                "psk"
                            ]
                        },
                        "identity": {
                            "type": "string"
                        }
                    },
                    "required": [
                        "scheme"
                    ]
                },
                {
                    "type": "object",
                    "properties": {
                        "@type": {
                            "$ref": "#/definitions/type_declaration"
                        },
                        "description": {
                            "$ref": "#/definitions/description"
                        },
                        "descriptions": {
```

```
                        "$ref": "#/definitions/descriptions"
                    },
                    "proxy": {
                        "$ref": "#/definitions/anyUri"
                    },
                    "scheme": {
                        "type": "string",
                        "enum": [
                            "oauth2"
                        ]
                    },
                    "authorization": {
                        "$ref": "#/definitions/anyUri"
                    },
                    "token": {
                        "$ref": "#/definitions/anyUri"
                    },
                    "refresh": {
                        "$ref": "#/definitions/anyUri"
                    },
                    "scopes": {
                        "oneOf": [{
                                "type": "array",
                                "items": {
                                    "type": "string"
                                }
                            },
                            {
                                "type": "string"
                            }
                        ]
                    },
                    "flow": {
                        "type": "string",
                        "enum": [
                            "code"
                        ]
                    }
                },
                "required": [
                    "scheme"
                ]
            }
        ]
```

```
        }
    },
    "type": "object",
    "properties": {
        "id": {
            "type": "string",
            "format": "uri"
        },
        "title": {
            "$ref": "#/definitions/title"
        },
        "titles": {
            "$ref": "#/definitions/titles"
        },
        "properties": {
            "type": "object",
            "additionalProperties": {
                "$ref": "#/definitions/property_element"
            }
        },
        "actions": {
            "type": "object",
            "additionalProperties": {
                "$ref": "#/definitions/action_element"
            }
        },
        "events": {
            "type": "object",
            "additionalProperties": {
                "$ref": "#/definitions/event_element"
            }
        },
        "description": {
            "$ref": "#/definitions/description"
        },
        "descriptions": {
            "$ref": "#/definitions/descriptions"
        },
        "version": {
            "type": "object",
            "properties": {
                "instance": {
                    "type": "string"
                }
```

```
            },
            "required": [
                "instance"
            ]
        },
        "links": {
            "type": "array",
            "items": {
                "$ref": "#/definitions/link_element"
            }
        },
        "forms": {
            "type": "array",
            "minItems": 1,
            "items": {
                "$ref": "#/definitions/form_element_root"
            }
        },
        "base": {
            "$ref": "#/definitions/anyUri"
        },
        "securityDefinitions": {
            "type": "object",
            "minProperties": 1,
            "additionalProperties": {
                "$ref": "#/definitions/securityScheme"
            }
        },
        "support": {
            "$ref": "#/definitions/anyUri"
        },
        "created": {
            "type": "string",
            "format": "date-time"
        },
        "modified": {
            "type": "string",
            "format": "date-time"
        },
        "security": {
            "oneOf": [{
                    "type": "string"
                },
                {
```

```
                    "type": "array",
                    "minItems": 1,
                    "items": {
                        "type": "string"
                    }
                }
            ]
        },
        "@type": {
            "$ref": "#/definitions/type_declaration"
        },
        "@context": {
            "$ref": "#/definitions/thing-context"
        }
    },
    "required": [
        "title",
        "security",
        "securityDefinitions",
        "@context"
    ],
    "additionalProperties": true
}
```

C. Thing Description Templates §

This section is non-normative.

A *Thing Description Template* is a description for a *class* of Things. It describes the properties, actions, events and common metadata that are shared for an entire group of Things, to enable the common handling of thousands of devices by a cloud server, which is not practical on a per-Thing basis. The Thing Description Template uses the same core vocabulary and information model from § 5. TD Information Model.

The Thing Description Template enables:

- management of multiple Things by a cloud service.
- simulation of devices/Things that have not yet been developed.
- common applications across devices from different manufacturers that share a common Thing model.

- combining multiple models into a Thing.

The Thing Description Template is a logical description of the interface and possible interaction with devices (properties, actions and events), however it does not contain device-specific information, such as a serial number, GPS location, security information or concrete protocol endpoints.

Since a Thing Description Template does not contain a Protocol Binding to specific endpoints and does not define a specific security mechanism, the *forms* and *securityDefinitions* and *security* keys must not be present.

The same Thing Description Template can be implemented by Things from multiple vendors, a Thing can implement multiple Thing Description Templates, define additional metadata (vendor, location, security) and define bindings to concrete protocols. To avoid conflicts between properties, actions and events from different Thing Description Templates that are combined into a common Thing, all these identifiers must be unique within a Thing.

A common Thing Description Template for a class of devices enables writing applications across vendors and creates a more attractive market for application developers. A concrete Thing Description can implement multiple Thing Description Templates and thus can aggregate function blocks into a combined device.

The business models of cloud vendors are typically built on managing thousands of identical devices. All devices with the same Thing Description Template can be managed in the same way by cloud applications. It is easy to create multiple simulated devices, if the interface and the instance are treated separately.

Since a Thing Description Template is a subset of the Thing Description in which some optional and mandatory Vocabulary Terms do not exist, however, it can be serialized in the same way and in the same formats as a Thing Description. Note that Thing Template instances cannot be validated in the same way as Thing Description instances due to some missing mandatory terms.

C.1 Thing Description Template Examples §

This section shows a Thing Description Template for a lamp and a Thing Description Template for a buzzer.

C.1.1 Thing Description Template: Lamp §

EXAMPLE 36: MyLampThingTemplate serialized in JSON

```
{
    "@context": ["https://www.w3.org/2019/wot/td/v1"],
    "@type" : "ThingTemplate",
    "title": "Lamp Thing Description Template",
    "description" : "Lamp Thing Description Template",
    "properties": {
        "status": {
            "description" : "current status of the lamp (on|off)",
            "type": "string",
            "readOnly": true
        }
    },
    "actions": {
        "toggle": {
            "description" : "Turn the lamp on or off"
        }
    },
    "events": {
        "overheating": {
            "description" : "Lamp reaches a critical temperature (overhea
            "data": {"type": "string"}
        }
    }
}
```

C.1.2 Thing Description Template: Buzzer §

EXAMPLE 37: MyBuzzerThingTemplate serialized in JSON

```
{
    "@context": ["https://www.w3.org/2019/wot/td/v1"],
    "@type" : "ThingTemplate",
    "title": "Buzzer Thing Description Template",
    "description" : "Thing Description Template of a buzzer that makes nc
     "actions": {
        "buzz": {
            "description" : "buzz for 10 seconds"
        }
     }
}
```

D. JSON-LD Context Usage §

This section is non-normative.

The present specification introduces the TD Information Model as a set of constraints over different Vocabularies, i.e. sets of Vocabulary Terms. This section briefly explains how a machine-readable definition of these constraints can be integrated into client applications, by making use of the mandatory `@context` of a TD document.

Accessing the TD Information Model from a TD document is done in two steps. First, clients must retrieve a mapping from JSON strings to IRIs. This mapping is defined as a JSON-LD context, as explained later. Second, clients can access the constraints defined on these IRIs by dereferencing them. Constraints are defined as logical axioms in the RDF format, readily interpretable by client programs.

All Vocabulary Terms referenced in § 5. TD Information Model are serialized as (compact) JSON strings in a TD document. However, each of these terms is unambiguously identified by a full IRI, as per the first Linked Data principle [LINKED-DATA]. The mappings from JSON keys to IRIs is what the `@context` value of a TD points to. For instance, the file at

`https://www.w3.org/2019/wot/td/v1`

includes the following mappings (among others):

`properties` → `https://www.w3.org/2019/wot/td#hasPropertyAffordance`
`object` → `https://www.w3.org/2019/wot/json-schema#ObjectSchema`

```
basic      → https://www.w3.org/2019/wot/security#BasicSecurityScheme
href       → https://www.w3.org/2019/wot/hypermedia#hasTarget
...
```

This JSON file follows the JSON-LD 1.1 syntax [JSON-LD11]. Numerous JSON-LD libraries can automatically process the `@context` of a TD and expand all the JSON strings it includes.

Once every Vocabulary Term of a TD is expanded to a IRI, the second step consists in dereferencing this IRI to get fragments of the TD Information Model that refer to that Vocabulary Term. For instance, dereferencing the IRI

```
https://www.w3.org/2019/wot/json-schema#ObjectSchema
```

results in an RDF document stating that the term `ObjectSchema` is a Class and more precisely, a sub-class of `DataSchema`. Such logical axioms are represented in RDF using formalisms of various complexity: here, sub-class relations are expressed as RDF Schema axioms [RDF-SCHEMA]. Moreover, these axioms may be serialized in various formats. Here, they are serialized in the Turtle format [TURTLE]:

```
<https://www.w3.org/2019/wot/json-schema#ObjectSchema>
    a rdfs:Class .
<https://www.w3.org/2019/wot/json-schema#ObjectSchema>
    rdfs:subClassOf <https://www.w3.org/2019/wot/json-schema#DataSchema> .
```

By default, if a user agent does not perform any content negotiation, a human-readable HTML documentation is returned instead of the RDF document. To negotiate content, clients must include the HTTP header `Accept: text/turtle` in their request.

E. Recent Specification Changes §

E.1 Changes from Proposed Recommendation §

- CoAP Content-Format ID 432 was assigned by IANA. (See section § 6.4 Identification and § 10.2 CoAP Content-Format Registration).
- In section § 8.1 Security Configurations, clarified the interaction between the dynamic authentication information asked by some security protocols and the security configurations information declared in the Thing Description.

- Several instances of a typo was fixed in appendix section § B. JSON Schema for TD Instance Validation

E.2 Changes from Second Candidate Recommendation §

- At-risk features `CertSecurityScheme`, `PublicSecurityScheme`, `PoPSecurityScheme` as well as `implicit`, `password` and `client` flows in `OAuth2SecurityScheme` were removed.
- In section § 5.3.2 Data Schema Vocabulary Definitions, clarified the relationship between data schemas and content types.
- In section § 6.3.9 `forms`, clarified the expectations of Consumers and Things for operation types `writemultipleproperties`, `readmultipleproperties` and `readallproperties`.
- In section § 8.3.1 Protocol Binding based on HTTP, the contexts in which default values `GET` or `PUT` are used for vocabulary term `htv:methodName` were clarified.
- The example Thing Description in appendix section § A.2 MyIlluminanceSensor Example with MQTT Protocol Binding was improved to make a better example using MQTT.

E.3 Changes from First Candidate Recommendation §

Changes from First Candidate Recommendation are described in the second Candidate Recommendation

F. Acknowledgements §

The editors would like to thank Michael Koster, Michael Lagally, Kazuyuki Ashimura, Ege Korkan, Daniel Peintner, Toru Kawaguchi, María Poveda, Dave Raggett, Kunihiko Toumura, Takeshi Yamada, Ben Francis, Manu Sporny, Klaus Hartke, Addison Phillips, Jose M. Cantera, Tomoaki Mizushima, Soumya Kanti Datta and Benjamin Klotz for providing contributions, guidance and expertise.

Also, many thanks to the W3C staff and all other current and former active Participants of the W3C Web of Things Interest Group (WoT IG) and Working Group (WoT WG) for their support, technical input and suggestions that led to improvements to this document.

Finally, special thanks to Joerg Heuer for leading the WoT IG for 2 years from its inception and guiding the group to come up with the concept of WoT building blocks including the Thing Description.

G. References §

G.1 Normative references §

[BCP47]

Tags for Identifying Languages. A. Phillips; M. Davis. IETF. September 2009. IETF Best Current Practice. URL: https://tools.ietf.org/html/bcp47

[html]

HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[RFC2046]

Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. N. Freed; N. Borenstein. IETF. November 1996. Draft Standard. URL: https://tools.ietf.org/html/rfc2046

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC3339]

Date and Time on the Internet: Timestamps. G. Klyne; C. Newman. IETF. July 2002. Proposed Standard. URL: https://tools.ietf.org/html/rfc3339

[RFC3629]

UTF-8, a transformation format of ISO 10646. F. Yergeau. IETF. November 2003. Internet Standard. URL: https://tools.ietf.org/html/rfc3629

[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

[RFC3987]

Internationalized Resource Identifiers (IRIs). M. Duerst; M. Suignard. IETF. January 2005. Proposed Standard. URL: https://tools.ietf.org/html/rfc3987

[RFC6570]

URI Template. J. Gregorio; R. Fielding; M. Hadley; M. Nottingham; D. Orchard. IETF. March 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6570

[RFC6749]

The OAuth 2.0 Authorization Framework. D. Hardt, Ed.. IETF. October 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6749

[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

[RFC7516]

JSON Web Encryption (JWE). M. Jones; J. Hildebrand. IETF. May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7516

[RFC7519]

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

[RFC7616]

HTTP Digest Access Authentication. R. Shekh-Yusef, Ed.; D. Ahrens; S. Bremer. IETF. September 2015. Proposed Standard. URL: https://httpwg.org/specs/rfc7616.html

[RFC7617]

The 'Basic' HTTP Authentication Scheme. J. Reschke. IETF. September 2015. Proposed Standard. URL: https://httpwg.org/specs/rfc7617.html

[RFC7797]

JSON Web Signature (JWS) Unencoded Payload Option. M. Jones. IETF. February 2016. Proposed Standard. URL: https://tools.ietf.org/html/rfc7797

[RFC8174]

Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://tools.ietf.org/html/rfc8174

[RFC8252]

OAuth 2.0 for Native Apps. W. Denniss; J. Bradley. IETF. October 2017. Best Current Practice. URL: https://tools.ietf.org/html/rfc8252

[RFC8259]

The JavaScript Object Notation (JSON) Data Interchange Format. T. Bray, Ed.. IETF. December 2017. Internet Standard. URL: https://tools.ietf.org/html/rfc8259

[RFC8288]

Web Linking. M. Nottingham. IETF. October 2017. Proposed Standard. URL: https://httpwg.org/specs/rfc8288.html

[RFC8392]

CBOR Web Token (CWT). M. Jones; E. Wahlstroem; S. Erdtman; H. Tschofenig. IETF. May 2018. Proposed Standard. URL: https://tools.ietf.org/html/rfc8392

[websub]

WebSub. Julien Genestoux; Aaron Parecki. W3C. 23 January 2018. W3C Recommendation. URL: https://www.w3.org/TR/websub/

[XMLSCHEMA11-2-20120405]

W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. David Peterson; Sandy Gao; Ashok Malhotra; Michael Sperberg-McQueen; Henry Thompson; Paul V. Biron et al. W3C. 5 April 2012. W3C Recommendation. URL: https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/

G.2 Informative references §

[ACE]

Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth). L. Seitz; G. Selander; E. Wahlstroem; S. Erdtman; H. Tschofenig. IETF. 27 March 2019. Internet-Draft. URL: https://tools.ietf.org/html/draft-ietf-ace-oauth-authz-24

[HTTP-in-RDF10]

HTTP Vocabulary in RDF 1.0. Johannes Koch; Carlos A. Velasco; Philip Ackermann. W3C. 2 February 2017. W3C Note. URL: https://www.w3.org/TR/HTTP-in-RDF10/

[IANA-MEDIA-TYPES]

Media Types. IANA. URL: https://www.iana.org/assignments/media-types/

[IANA-URI-SCHEMES]

Uniform Resource Identifier (URI) Schemes. IANA. URL: https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml

[JSON-LD11]

JSON-LD 1.1. Gregg Kellogg; Pierre-Antoine Champin; Dave Longley. W3C. 5 March 2020. W3C Candidate Recommendation. URL: https://www.w3.org/TR/json-ld11/

[JSON-SCHEMA]

JSON Schema Validation: A Vocabulary for Structural Validation of JSON. Austin Wright; Henry Andrews; Geraint Luff. IETF. 19 March 2018. Internet-Draft. URL: https://tools.ietf.org/html/draft-handrews-json-schema-validation-01

[LDML]

Unicode Technical Standard #35: Unicode Locale Data Markup Language (LDML). Mark Davis; CLDR Contributors. URL: https://unicode.org/reports/tr35/

[LINKED-DATA]

Linked Data Design Issues. Tim Berners-Lee. W3C. 27 July 2006. W3C-Internal Document. URL: https://www.w3.org/DesignIssues/LinkedData.html

[MQTT]

MQTT Version 3.1.1. Andrew Banks; Rahul Gupta. OASIS. 10 December 2015. OASIS Standard Incorporating Approved Errata 01. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

[OPENAPI]

OpenAPI Specification: Version 3.0.1. Darrel Miller; Jason Harmon; Jeremy Whitlock; Kris Hahn; Marsh Gardiner; Mike Ralphson; Rob Dolin; Ron Ratovsky; Tony Tam. OpenAPI Initiative, Linux Foundation. 7 December 2017. URL: https://swagger.io/specification/

[RDF-SCHEMA]

RDF Schema 1.1. Dan Brickley; Ramanathan Guha. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/rdf-schema/

[RFC3966]

The tel URI for Telephone Numbers. H. Schulzrinne. IETF. December 2004. Proposed Standard. URL: https://tools.ietf.org/html/rfc3966

[RFC6068]

The 'mailto' URI Scheme. M. Duerst; L. Masinter; J. Zawinski. IETF. October 2010. Proposed Standard. URL: https://tools.ietf.org/html/rfc6068

[RFC7231]

Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. R. Fielding, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: https://httpwg.org/specs/rfc7231.html

[RIJGERSBERG]

Ontology of Units of Measure and Related Concepts. Hajo Rijgersberg; Mark van Assem; Jan Top. Semantic Web journal, IOS Press. 2013. URL: http://www.semantic-web-journal.net/content/ontology-units-measure-and-related-concepts

[SEMVER]

Semantic Versioning 2.0.0. Tom Preston-Werner. 26 December 2017. URL: https://semver.org/

[SMARTM2M]

ETSI TS 103 264 V2.1.1 (2017-03): SmartM2M; Smart Appliances; Reference Ontology and oneM2M Mapping. ETSI. March 2017. Published. URL: http://www.etsi.org/deliver/etsi_ts/103200_103299/103264/02.01.01_60/ts_103264v020101p.pdf

[string-meta]

Strings on the Web: Language and Direction Metadata. Addison Phillips; Richard Ishida. W3C. 11 June 2019. W3C Working Draft. URL: https://www.w3.org/TR/string-meta/

[TURTLE]

RDF 1.1 Turtle. Eric Prud'hommeaux; Gavin Carothers. W3C. 25 February 2014. W3C Recommendation. URL: https://www.w3.org/TR/turtle/

[VOCAB-SSN]

Semantic Sensor Network Ontology. Armin Haller; Krzysztof Janowicz; Simon Cox; Danh Le Phuoc; Kerry Taylor; Maxime Lefrançois. W3C. 19 October 2017. W3C Recommendation. URL: https://www.w3.org/TR/vocab-ssn/

[WOT-ARCHITECTURE]

Web of Things (WoT) Architecture. Matthias Kovatsch; Ryuichi Matsukura; Michael Lagally; Toru Kawaguchi; Kunihiko Toumura; Kazuo Kajimoto. W3C. May 2019. URL: https://www.w3.org/TR/wot-architecture/

[WOT-BINDING-TEMPLATES]

Web of Things (WoT) Binding Templates. Michael Koster; Ege Korkan. W3C. 30 January 2020. W3C Note. URL: https://www.w3.org/TR/wot-binding-templates/

[WOT-SECURITY-GUIDELINES]

Web of Things (WoT) Security and Privacy Guidelines. ; Michael McCool; Elena Reshetova. W3C. March 2019. URL: https://w3c.github.io/wot-security/

[xml]

Extensible Markup Language (XML) 1.0 (Fifth Edition). Tim Bray; Jean Paoli; Michael Sperberg-McQueen; Eve Maler; François Yergeau et al. W3C. 26 November 2008. W3C Recommendation. URL: https://www.w3.org/TR/xml/

↥