This is the first draft of the new simplified TD approach based on JSON-LD 1.1. Some definitions are not finished yet and are still in progress. A stable Thing Description deliverable version based on JSON-LD 1.0 can be found here.

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.

Implementers need to be aware that this specification is considered unstable. Vendors interested in implementing this specification before it eventually reaches the Candidate Recommendation phase should subscribe to the repository and take part in the discussions.

Please contribute to this draft using the GitHub Issue feature of the WoT Thing Description repository. For feedback on security and privacy considerations, please use the WoT Security and Privacy Issues, as security and privacy is cross-cutting over all our documents.

Introduction

The 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). The TD consists of semantic metadata for the Thing itself, an interaction model based on WoT's `Properties`, `Actions`, and `Events` paradigm, a semantic schema to make data models machine-understandable, and features for Web Linking to express relations among Things.

Properties can be used for sensing and controlling parameters of a Thing, such as getting the current value or setting an operation state. *Actions* model invocation of physical (and hence time-consuming) processes, but can also be used to abstract RPC-like calls of existing platforms. *Events* are used for the push model of communication where notifications, discrete events, or streams of values are sent asynchronously to the receiver. In general, the TD provides metadata for different communication bindings identified by URI schemes (e.g., "http", "coap", "mqtt", etc.), media types (e.g., "application/json", "application/xml", "application/cbor", "application/exi" etc.), and security mechanisms (for authentication, authorization, confidentiality, etc.). Serialization of TD instances is based on JSON and includes at least the TD core vocabulary as JSON keys as defined in section 5 in this specification document.

Formatted: Emphasis, English (US)

Formatted: Emphasis, English (US)

Formatted: Emphasis, English (US)

Example 1 shows a simple TD instance in such a JSON serializiation and depicts WoT's `Properties`, `Actions`, and `Events` paradigm by describing a lamp Thing with the name *MyLampThing*.

```
 {     "id": "urn:dev:wot:com:example:servient:lamp",     "name":
"MyLampThing",     "security": [{"scheme": "basic"}],     "properties":
{         "status" : {             "type": "string",
"forms": [{"href": "https://mylamp.example.com/status"}]         }
},     "actions": {         "toggle" : {            "forms": [{"href":
"https://mylamp.example.com/toggle"}]         }     },     "events":{
"overheating":{             "type": "string",             "forms": [{
"href": "https://mylamp.example.com/oh",              "subProtocol":
"LongPoll"            }]          }     } }
```

Based on this content, we know there exists one `Property` interaction resource with the name *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` key), and will return a string 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` is specified to toggle the switch status using the POST method applied to the `https://mylamp.example.com/toggle` resource, where POST is again a default assumption for invoking Actions.

The `Event` pattern 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 the HTTP with its long polling sub-protocol at `https://mylamp.example.com/oh`.

This example also specifies the `basic` security scheme, requiring a username and password for access. In combination with the use of the HTTP protocol this indicates the use of HTTP Basic Authentication. Specification of a security scheme at the top level as in this example indicates that it is required for every resource. However, security schemes can also be specified per-interaction or per-form, with lower-level configurations overriding higher-level ones, allowing for the specification of fine-grained access control. Examples are provided later.

The TD in Example 1 reflects some additional defined default assumptions that are not explicitly described. For example, the media type of the exchange format of the interactions is assumed to be JSON (=`mediaType`) and the `Property` *status* resource is not writable as well as not observable. Specifically, the TD specification defines vocabulary terms (`writable`, `observable`, `mediaType`) that have default values. If these vocabulary terms are not explicitly used in a Thing Description instance, the Thing Description processor follows default assumptions for interpretation as defined in this specification.

The TD can be also processed as an RDF-based model. In that case, the Thing Description instance needs to be transformed into valid JSON-LD first. In terms of JSON-LD 1.1 serialization, the open-world assumption of RDF semantic processing requires vocabulary terms with default values to be always present explicitly in the instances. Example 2 shows the same TD in a JSON-LD 1.1 serializiation representing exactly the same information as in Example 1; however, default values have been filled in.

```
 {     "@context": "http://www.w3.org/ns/td",     "id":
"urn:dev:wot:com:example:servient:lamp",     "name": "MyLampThing",
"security": [{"scheme": "basic", "in": "header"}],     "properties": {
"status": {             "writable": false,             "observable":
false,           "type": "string",           "forms": [{
"href": "https://mylamp.example.com/status",
"http:methodName": "GET",                 "mediaType":
"application/json"             }]         }     },     "actions": {
"toggle": {             "forms": [{                 "href":
"https://mylamp.example.com/toggle",                 "http:methodName":
"POST",                 "mediaType": "application/json"             }]
}     },     "events": {         "overheating": {           "type":
"string",           "forms": [{                 "href":
"https://mylamp.example.com/oh",                 "subProtocol":
"LongPoll",                 "mediaType": "application/json"
}]         }     } }
```

For more examples, including the use of other protocols besides HTTP, see Section .

Terminology

Generic WoT terminology is defined in [[!WOT-ARCHITECTURE]]: *Thing*, *Thing Description* (in short *TD*), *Web of Things* (in short ***WoT***), *WoT Interface* etc.

Namespaces

The namespace for the W3C TD vocabulary as defined in this document is http://www.w3.org/ns/td.

Using content negotiation, this namespace serves either the TD ontology file (Turtle) or the TD context file (JSON-LD).

The suggested ML: is this normative? prefix for the TD namespace is `td`.

The security ontology that can be used with the TD is available at https://www.w3.org/ns/wot-security.

The data schema ontology that can be used with the TD is available at https://www.w3.org/ns/json-schema.

ML: This section is in an editors note, but the http namespace appears to be normative. Please clarify. The TD will reuse existing vocabulary definitions such as for http from the http://www.w3.org/2011/http# namespace. In that case the prefix `http` is used. However, there are prefixes such as `coap` and `mqtt` which have no namespace yet. Currently there are efforts to have such namespace representations which will be referenced in the TD specification in the future. In the meantime, the [[!WOT-PROTOCOL-BINDING]] provides the list of terms that can be used to specify the protocol metadata in the Thing Description.

Conformance

As well as all 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 *MUST*, *MUST NOT*, *REQUIRED*, *SHOULD*, *SHOULD NOT*, *RECOMMENDED*, *MAY*, and *OPTIONAL* in this specification are to be interpreted as described in [[!RFC2119]].

A Thing Description instance complies with this specification if it follows the normative statements in Section and Section regarding Thing Description serialization.

A JSON Schema is provided in Annex to validate Thing Description instances based on JSON-LD 1.1.

In the future some information about RDF validation will be provided.

Information Model

Overview

The W3C Thing Description provides a set of vocabulary for describing physical and virtual Things. To increase interoperability the vocabulary terms are defined using the Resource Description Framework (RDF). All vocabulary restrictions noted in these tables MUST be followed,including mandatory items and default values.

In general, the Thing Description vocabulary set is grouped in three modules: the core Thing Description vocabulary reflecting WoT's paradigm of `Properties` , `Actions` , and `Events` (also see [[!WOT-ARCHITECTURE]]);the data schema vocabulary reflecting a subset of JSON Schema terms in a linked data representation; and the security vocabulary used to define security mechanismconfiguration requirements.

An overview of this vocabulary with its class context and class relation is given by the following three figures: the TD core model, the TD data schema model, and the TD security model. Please note that the figures reflect the vocabulary terms and structure as they would be used in a Thing Description instance (see Section).The full ontology definitions of the different modules can be viewed by following the namespaces as provided in Section .

ML: The diagram is missing the name and security fields TD core model TD data schema

model TD security model

In the figures above, and in the tables to follow, items which have default values are indicated as being optional. However, technically, these are optional *only in the JSON serialization*. They are actually mandatory parts of the information model and are also mandatory in the JSON-LD serialization. In addition, the security scheme configuration is not actually optional even in the JSON serialization. Security configuration is mandatory for at least one of the three levels at which it may be specified... so it can be omitted only if it is specified at a different level for each form. A future version of this document should better express the status of these attributes.

A detailed description of the vocabulary of the TD core model and TD data schema model is given in the next sub-section.

Core Vocabulary Definition

ML: The references for the types (anyURI etc.) don't point to a proper type definition

`Thing`

Describes a physical and/or virtual Thing (may represent one or more physical and/or virtual Things) in the Web of Things context.

Field Name	Description	Mandatory	Default value	Type
`id`	unique identifier of the Thing (URI, e.g. custom URN)	yes	.	`anyURI`
`support`	Provides information about the TD maintainer (e.g., author, link or telephone number to get support, etc).	no	.	`string`
`name`	Name of the Thing.	yes	.	`string`
`description`	Provides additional (human) readable information.	no	.	`string`
`base`	Define the base URI that is valid for all defined local interaction resources. All other URIs in the TD must then be resolved using the algorithm defined in [[!RFC3986]].	no	.	`anyURI`

`properties`	All Property-based interaction patterns of the Thing.	no	.	[Property]
`actions`	All Action-based interaction patterns of the Thing.	no	.	[Action]
`events`	All Event-based interaction patterns of the Thing.	no	.	[Event]
`links`	Provides Web links to arbitrary resources that relate to the specified Thing Description.	no	.	array of [Link]
`security`	Set of security configurations that must all be satisfied for access to resources at or below the current level, if not overridden at a lower level. ML: do really **ALL** configurations have to be satisifed?	no	.	array of [SecurityScheme]

ML: It should be explicitly stated, that these fields can be extended by proprietary additional fields

`InteractionPattern`

Three interaction patterns are defined as subclasses: Property, Action and Event. When a concrete Property, Action or Event is defined in a Thing Description, it is called an "interaction resource". Interactions between Things can be as simple as one Thing accessing another Thing's data to get or (in the case the data is also writable) change the value of data such as metadata, status or mode. A Thing may also be interested in getting asynchronously notified of future changes in another Thing, or may want to initiate an action served ML: I replaced process with action, since it is an implementation term. How can a thing monitor an action? in another Thing that may take some time to complete and monitor the progress. Interactions between Things may involve exchanges of data between them. This data can be either given as input by the client Thing, returned as output by the server Thing or both.

Deleted: representation

Deleted: a process served

Each instance of a `Property`, `Action`, and `Event` class *MUST* have a unique name. See Section [Representation Format] for more details about JSON-LD 1.1 identifiers.

Deleted: has

Deleted: (

Deleted:)

Field Name	Description	Mandatory	Default value	Type
`forms`	Indicates one or more endpoints from which an interaction pattern is accessible.	yes	.	array of [Form]
`label`	Provides a label (e.g., display a text for UI representation) of the interaction pattern.	no	.	[string]
`description`	Provides additional (human) readable information.	no	.	[string]
`security`	Set of security configurations that must all be satisfied for access to resources at or below the current level, if not overridden at a lower level.	no	.	array of [SecurityScheme]
`scopes`	Array of authorization scope identifiers. 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.	no	.	array of [string]

The class `InteractionPattern` has the following subclasses:

- Event
- Property
- Action

Property

Properties expose internal state of a Thing that can be directly retrieved (get) and optionally modified (set). In addition, Things can also choose to make Properties observable by pushing the new state (not an event) after a change; this must follow eventual consistency (also see CAP Theorem).

Field Name	Description	Mandatory	Default value	Type
`name`	Name of the Property.	yes	.	`string`
`observable`	Indicates whether a remote servient can subscribe to ("observe") the Property, to receive change notifications or periodic updates (true/false).	no	false	`boolean`
`writable`	Boolean value that indicates whether a property is writable (=true) or not (=false).	no	false	`boolean`

Property instances may also be instances of the class DataSchema and therefore can contain, among others, the `type` term.

Action

Actions offer a way to invoke functions of the Thing. These functions may manipulate the interal state of a Thing in a way that is not possible through setting Properties. Examples are changing internal state that is not exposed as a Property, changing multiple Properties, changing Properties over time or with a process that should not be disclosed. Actions may also be pure functions, that is, they may not use any internal state at all, and may simply process input data and return a result that directly depends only on the input given. ML: I believe the asynchronous behavior is the main differentiator between an action and a property. This should be made explicit in this section.

Field Name	Description	Mandatory	Default value	Type
`name`	Name of the Action.	yes	.	`string`
`output`	Link to the n-ary class that allows the declaration of the data type returned by an action.	no	.	`DataSchema`
`input`	Link to the n-ary class that allows the declaration of the accepted data type of an action.	no	.	`DataSchema`

Event

The Event Interaction Pattern describes event sources that asynchronously push messages. Here not state, but state transitions (events) are communicated (e.g., "clicked"). Events may be triggered by internal state changes that are not exposed as Properties. Events usually follow strong consistency, where messages need to be queued to ensure eventual delivery of all events that have occurred.

Field Name	**Description**	**Mandatory**	**Default value**	**Type**
`name`	Name of the Event.	yes	.	`string`

Instances of the `Event` class contains the term definitions of the class InteractionPattern and DataSchema.

Form

Communication protocol metadata indicating where a service can be accessed by a client application. An interaction might have more than one form.

Field Name	Description	Mandatory	Default value	Type
`href`	URI of the endpoint where an interaction pattern is provided.	yes	.	anyURI
`mediaType`	Assign the underlying media type of an interaction pattern based on IANA (https://www.iana.org/assignments/media-types/media-types.xhtml).	no	application/json	string
`rel`	Indicates the expected result of performing the operation 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 rel attribute indicates which form is which and allows the client to select the correct form for the operation required. The value of the `rel` attribute of the form must be one of readproperty, writeproperty, observeproperty for properties, invokeaction for actions, and subscribeevent or unsubscribeevent for events.	no	.	string (one of `"readproperty"`, `"writeproperty"`, `"observeproperty"`, `"invokeaction"`, `"subscribeevent"`, or `"unsubscribeevent"`)
`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.	no	.	string (one of `"LongPoll"`)
`security`	Set of security configurations that must all be satisfied for access to resources at or below the current level, if not overridden at a lower level. ML: must **ALL** be satisfied?	no	.	array of SecurityScheme
`scopes`	Array of authorization scope identifiers. 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.	no	.	string

Link

A Web link, as specified by [[!RFC8288]] (https://tools.ietf.org/html/rfc8288).

Deleted: IETF RFC 8288

Field Name	Description	Mandatory	Default value	Type
`href`	URI of the endpoint where an interaction pattern is provided.	yes	.	anyURI
`mediaType`	Assign the underlying media type of an interaction	no	application/json	string

Field Name	Description	Mandatory	Default value	Type
	pattern based on IANA (https://www.iana.org/assignments/media-types/media-types.xhtml). ML: move to normative reference section			
rel	Indicates the expected result of performing the operation 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 rel attribute indicates which form is which and allows the client to select the correct form for the operation required. ML: who defines the permitted values and the mapping / meaning?	no	.	string
anchor	By default, the context of a link is the URL of the representation it is associated with ML: does this sentence mean, that a link is local to the TD, if no anchor is present? , and is serialised as a URI. When present, the anchor parameter overrides this with another URI, such as a fragment of this resource, or a third resource (i.e., when the anchor value is an absolute URI).	no	.	anyURI

Data Schema Vocabulary Definition

DataSchema

Field Name	Description	Mandatory	Default value	Type
description	Provides additional (human) readable information.	no	.	string
type	Assignment of JSON-based data types compatible with JSON Schema ML: add normative reference to JSON schema, the null value should be clarified (one of boolean, integer, number, string, object, array, or null).	no	.	string (one of "boolean" , "integer" , "number" , "string" , "object" , "array" , or "null")
const	Provides a constant value.	no	.	anyType
enum	Restricted set of values provided as an array.	no	.	array of anyType

The class DataSchema has the following subclasses:

- ArraySchema
- BooleanSchema
- IntegerSchema
- NumberSchema
- ObjectSchema
- StringSchema

`ArraySchema`

A JSON array specification ("type": "array").

Field Name	Description	Mandatory	Default value	Type
`items`	Used to define the characteristics of an array.	no	.	`DataSchema`
`minItems`	Defines the minimum number of items that have to be in the array.	no	.	`unsignedInt`
`maxItems`	Defines the maximum number of items that have to be in the array.	no	.	`unsignedInt`

`ObjectSchema`

A JSON object specification ("type": "object").

Field Name	Description	Mandatory	Default value	Type
`properties`	Data schema nested definitions.	no	.	`DataSchema`
`required`	Defines which members of an object type are mandatory. Defines which members of the object type are mandatory.	no	.	array of `string`

`BooleanSchema`

A JSON boolean value specification ("type": "boolean").

`NumberSchema`

A JSON number value specification ("type": "number").

Field Name	Description	Mandatory	Default value	Type
`minimum`	Specifies a minimum numeric value	no	.	`double`
`maximum`	Specifies a maximum numeric value	no	.	`double`

Deleted: Only applicable for associated number or integer types.

Deleted: Only applicable for associated number or integer types.

`StringSchema`

A JSON string value specification ("type": "string").

`IntegerSchema`

A JSON integer value specification, that is, numbers without a fractional part ("type": "integer").

Field Name	Description	Mandatory	Default value	Type
`minimum`	Specifies a minimum numeric value	no	.	`integer`

Deleted: Only applicable for associated number or integer types.

`maximum`	Specifies a maximum numeric value	no	.	integer

Deleted: Only applicable for associated number or integer types.

Security Vocabulary Definition

The set of security metadata supported by the core TD vocabulary is still under discussion. For the core vocabulary the focus is on well-established security mechanisms, such as those built into protocols supported by WoT or already in wide use with those protocols. Note that the vocabulary extension mechanism of the WoT Thing Description allows for additional security schemes if needed. The current set of security schemes is partly based on OpenAPI 3.0.1. ML: Add to normative references. It would be useful to explain the difference between what we do here and Open API. Are we picking a subset or are we defining extensions? The security schemes, vocabulary and syntax given in this specification share many similarities with OpenAPI; however, it is not fully compatible. Also, as OpenAPI targets only web services built around HTTP it does not cover the full set of use cases required for the IoT. Therefore, we are currently discussing metadata supporting additional security mechanisms for IoT-centered protocols such as CoAP and MQTT. For more information about what security schemes are under discussion (and to file issues if you have a request) please visit the WoT Security TF repository.

`SecurityScheme`

Field Name	Description	Mandatory	Default value	Type
`scheme`	Identification of security mechanism being configured.	yes	.	string (one of "nosec" "basic" "cert" "digest" "bearer" "pop" "psk" "public" "oauth2" or "apikey"
`description`	Provides additional (human) readable information.	no	.	string
`proxyUrl`	URI of the proxy server this security configuration provides access to. If not given, the corresponding security configuration is for the endpoint.	no	.	anyURI

Deleted: ",
Deleted: ",
Deleted: ",
Deleted: ",
Deleted: ",
Deleted: ",
Deleted: ",
Deleted: ",
Deleted: ",
Deleted: ")

The class `SecurityScheme` has the following subclasses:

- `CertSecurityScheme`
- `PoPSecurityScheme`
- `APIKeySecurityScheme`
- `BasicSecurityScheme`
- `BearerSecurityScheme`
- `DigestSecurityScheme`
- `NoSecurityScheme`
- `OAuth2SecurityScheme`
- `PSKSecurityScheme`
- `PublicSecurityScheme`

NoSecurityScheme

A security configuration corresponding to ("scheme": "nosec"), indicating there is no authentication or other mechanism required to access the resource.

BasicSecurityScheme

Basic authentication security configuration ("scheme": "basic"), using an unencrypted username and password. This scheme should be used with some other security mechanism providing confidentiality, for example, TLS.

Field Name	Description	Mandatory	Default value	Type
`name`	Name for query, header, or cookie parameters.	no	.	string
`in`	Specifies the location of security authentication information (one of header, query, body, or cookie).	no	header	string

CertSecurityScheme

Certificate-base asymmetric key security configuration ("scheme": "cert").

Field Name	Description	Mandatory	Default value	Type
`identity`	Pre-shared key identity.	no	.	string

DigestSecurityScheme

Digest authentication security configuration ("scheme": "digest"). This scheme is similar to basic authentication but with added features to avoid man-in-the-middle attacks.

Field Name	Description	Mandatory	Default value	Type
`qop`	Quality of protection (one of auth or auth-int).	no	auth	string
`in`	Specifies the location of security authentication information (one of header, query, body, or cookie).	no	header	string
`name`	Name for query, header, or cookie parameters.	no	.	string

BearerSecurityScheme

Bearer token authentication security configuration ("scheme": "bearer"). This scheme is intended 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.

Field Name	Description	Mandatory	Default value	Type
`authorizationUrl`	URI of the authorization server.	no	.	anyURI
`alg`	Encoding, encryption, or digest algorithm (one of MD5, ES256, or ES512-256).	no	ES256	string
`format`	Specifies format of security authentication information (one of jwt, jwe, or jws).	no	jwt	string

Field Name	Description	Mandatory	Default value	Type
`in`	Specifies the location of security authentication information (one of header, query, body, or cookie).	no	header	[`string`]()
`name`	Name for query, header, or cookie parameters.	no	.	[`string`]()

`PSKSecurityScheme`

Pre-shared key authentication security configuration ("scheme": "psk").

Field Name	Description	Mandatory	Default value	Type
`identity`	Pre-shared key identity.	no	.	`string`

`PublicSecurityScheme`

Raw public key asymmetric key security configuration ("scheme": "public").

Field Name	Description	Mandatory	Default value	Type
`identity`	Pre-shared key identity.	no	.	`string`

`OAuth2SecurityScheme`

OAuth2 authentication security configuration ("scheme": "oauth2"). For the implicit flow the authorizationUrl and scopes are *REQUIRED*. For the password and client flows both tokenUrl and scopes are *REQUIRED*. For the code flow authorizationUrl, tokenUrl, and scopes are *REQUIRED*.

Deleted: required

Deleted: required

Deleted: required

Field Name	Description	Mandatory	Default value	Type
`authorizationUrl`	URI of the authorization server.	no	.	`anyURI`
`tokenUrl`	URI of the token server.	no	.	`anyURI`
`refreshUrl`	URI of the refresh server.	no	.	`anyURI`
`scopes`	Array of authorization scope identifiers. 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. ML: we need an example or more descriptive text to understand this paragraph.	no	.	array of `string`
`flow`	Authorization flow (one of implicit, password, client, or code).	no	implicit	`string`

`APIKeySecurityScheme`

API key authentication security configuration ("scheme": "apikey"). This is for the case where the access token is opaque and is not using a standard token format.

Field Name	Description	Mandatory	Default value	Type
`in`	Specifies the location of security authentication information	no	query	`string`

Field Name	Description	Mandatory	Default value	Type
	(one of header, query, body, or cookie).			
`name`	Name for query, header, or cookie parameters.	no	.	[`string`]()

`PoPSecurityScheme`

Proof-of-possession token authentication security configuration ("scheme": "pop").

Field Name	Description	Mandatory	Default value	Type
`alg`	Encoding, encryption, or digest algorithm (one of MD5, ES256, or ES512-256).	no	ES256	[`string`]()
`authorizationUrl`	URI of the authorization server.	no	.	[`anyURI`]()
`format`	Specifies format of security authentication information (one of jwt, jwe, or jws).	no	jwt	[`string`]()
`in`	Specifies the location of security authentication information (one of header, query, body, or cookie).	no	header	[`string`]()
`name`	Name for query, header, or cookie parameters.	no	.	[`string`]()

Thing Description Serialization

This is the first draft that uses JSON-LD 1.1 as a serialization format of the Thing Description. As that is work in progress, working assumptions are based on the latest Community Draft of JSON-LD 1.1. ML: add to normative reference section A new JSON-LD Working Group has already been chartered. It is planned that this section will conform to the latest working draft of their JSON-LD 1.1 deliverable and only use stable elements.

Thing Description instances are modeled and structured based on Section . This section defines a TD serialization based on JSON [[!RFC8259]].

Representation Format

The JSON serialization of TDs follows the syntax of JSON-LD 1.1 in order to streamline semantic evaluation. Hence, serializations can be parsed either as raw JSON or with a JSON-LD 1.1 processor.

In order to enable this convergence, all vocabulary terms (field names) defined in Section have a JSON key representation.

Deleted: will

In addition, Thing Description instances MAY contain JSON-LD 1.1 keywords such as `@context` and `@type`.

ML: are additional JSON-LD keywords valid?

The data types of the vocabulary as defined in Section will be transformed to JSON-based types. The following rules are used for vocabulary terms based on some simple type definitions:

ML: Suggest to use the term "fields" instead of "vocabulary term" all over the spec to improve readability

- Vocabulary terms that use the simple types string and anyURI MUST be serialized as JSON string.
- Vocabulary terms that use the simple types integer and unsignedInt MUST be serialized as JSON integer.
- Vocabulary terms that use the simple type double MUST be serialized as JSON number.

Deleted: type

All vocabulary terms in Section associated with more complex class-based types are defined separately for structured JSON type transformation in the following subsections.

Thing as a whole

The root object of a Thing Description instance MAY include the `@context` key from JSON-LD 1.1 with the value URI of the Thing Description context file `http://www.w3.org/ns/td`.

Deleted: description

```
{       "@context": "http://www.w3.org/ns/td",      ... }
```

ML: This paragraph appears to be normative and not an editors note. Section 3 mentions RDF as alternative format - what accept header is required ? `http://www.w3.org/ns/td` uses content negotiation to return the context file. Thus, it must be fetched with an `Accept` header set to `application/ld+json`.

ML: The next paragraph occurs several times in similar form - the spec should not consider how the TD file is processed, but which fields are reuired/permitted in a well-formed TD. When a Thing Description instance is processed and interpreted by a JSON-LD 1.1 processor the `@context` field MUST be present (see also Section JSON-LD 1.1 Processing).

When a single Thing Description instance involves several contexts, additional namespaces with prefixes MUST be appended to the `@context` array structure. This option proves relevant if one wants to extend the existing Thing Description context without modifying it. ML: I'm not sure I understand the previous sentence. Do we want to express, that additional semantic types can be added by including additional contexts? For instance:

```
{     "@context": ["http://www.w3.org/ns/td",                  {"iot": "http://iotschema.org/"}],      ... }
```

Each mandatory and optional field name as defined in the class `Thing` MUST be serialized as a JSON key in the root object of the Thing Description instance.

The type of the fields `properties`, `actions`, and `events` MUST be a JSON object.

The type of the fields `links` and `security` MUST be a JSON array.

A TD snippet based on the fields of the class `Thing` without the optional field `@context` is given below:

Deleted: defined

```
{     "id": "urn:dev:wot:com:example:servient:myThing",     "name": "MyThing",     "description": "Additional (human) readable information of the Thing.",     "support": "https://servient.example.com/contact",     "security": [...],     "base": "https://servient.example.com/",     "properties": {...},     "actions": {...},     "events": {...},     "links": [...] }
```

Alternatively, the same example can be written instead to explicitly include the (semantic) keys used by JSON-LD 1.1 (`@context` and `@type`):

```
 {     "@context": "http://www.w3.org/ns/td",     "@type": "Thing",
"id": "urn:dev:wot:com:example:servient:myThing",     "name":
"MyThing",     "description": "Additional (human) readable information
of the Thing.",     "support": "https://servient.example.com/contact",
"security": [...],     "base": "https://servient.example.com/",
"properties": {...},     "actions": {...},     "events": {...},
"links": [...] }
```

`properties`

Properties (and sub-properties) offered by a Thing MUST be collected in the JSON-object based `properties` field with unique names as JSON keys. ML: What are sub-properties? The language can be read as if there is one hierarchy-level.

Each mandatory and optional vocabulary term as defined in the class [`Property`](), as well as its two superclasses [`InteractionPattern`]() and [`DataSchema`](), MUST be serialized as a JSON key within a Property object.

The type of the fields `properties` and `items` MUST be serialized as a JSON object.

The type of the fields `forms`, `required`, and `enum`, and potentially `security`, MUST be serialized as a JSON array. ML: *required* is not defined in the property section of chapter 5. Why is security "potentially"?

A TD snippet based on the defined fields is given below:

```
 {     ...     "properties": {         "on": {             "label":
"On/Off",             "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": [...]         }
}     ... }
```

Similar to the case at the `Thing` level, properties MAY have additional semantic annotations based on JSON-LD 1.1 keywords.

ML: implementation detail vs. normative requirement. I think the normative requirement is "is a valid JSON-LD 1.1 TD", since this requirement only applies for this case.

When a Thing Description instance is processed and interpreted by a JSON-LD 1.1 processor, each `property` MUST contain the vocabulary terms `observable` and `writable` due to the [open-world assumption]() of Linked Data. This assumption means that if a Linked Data model of a Thing Description instance were to omit these vocabulary terms, then the interpreter would not be able to make any assumptions about their actual value.

Deleted: (

Deleted:) Property

A snippet of a JSON-LD 1.1 processable TD serialization including semantic annotations and the default values of `observable` and `writable` based on the class `Property` is given as follows:

```
     ...      "properties": {          "on": {              "@type":
"iot:SwitchToggle",             "label": "On/Off",
"writable": false,             "observable": false,              "type":
"boolean",              "forms": [...]          },          "status": {
"writable": false,             "observable": false,              "type":
"object",              "properties": {                "brightness": {
"@type": "iot:CurrentLevel",                   "type": "number",
"minimum": 0.0,                     "maximum": 100.0                 },
"rgb": {                     "@type": "iot:rgbData",
"type": "array",                     "items" : {
"type" : "number",                         "minimum": 0,
"maximum": 255                     },                      "minItems":
3,                      "maxItems": 3                  }              },
"required": ["brightness","rgb"],             "forms": [...]          }
}      ...
```

actions

Actions offered by a Thing MUST be collected in the JSON-object based `actions` field with (unique) Action names as JSON keys.

Each optional vocabulary term as defined in the class `Action` and its superclass `InteractionPattern` MUST be serialized as a JSON key within an Action object.

The type of the fields `input` and `output` MUST be serialized as a JSON object.

The keys of `input` and `output` rely on the the class `DataSchema`.

The type of the field `forms`, and potentially `security`, MUST be serialized as a JSON array.

A TD snippet based on the defined fields is given below:

```
     ...      "actions": {          "fade" {              "label": "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": [...]          }          ...      }
...
```

Definitions within the `actions` field MAY have additional semantic annotations based on JSON-LD 1.1 keywords.

events

Events offered by a Thing MUST be collected in the JSON-object based `events` field with (unique) Event names as JSON keys.

Each optional vocabulary term as defined in the class `Event`, as well as its two superclasses `InteractionPattern` and `DataSchema`, MUST be serialized as a JSON key within an Event object.

The type of the fields `properties` and `items` MUST be serialized as a JSON object.

The type of the fields `forms`, `required`, and `enum`, and potentially `security`, MUST be serialized as a JSON array.

A TD snippet based on the defined fields is given below:

```
     ...      "event": {          "overheated": {              "type":
"object",          "properties": {              "temperature": {
"type": "number" }              },              "forms": [...]          }
...      }      ...
```

Definitions within the `events` field MAY have additional semantic anotations based on JSON-LD 1.1 keywords.

`forms`

Each mandatory and optional vocabulary term as defined in the class `Form`, MUST be serialized as a JSON key.

If required, `forms` MAY be supplemented with protocol-specific vocabulary terms identified with a prefix. See also [[!WOT-PROTOCOL-BINDING]].

When a Thing Description instance is processed and interpreted by a JSON-LD 1.1 processor, each `forms` (array) entry MUST contain a `mediaType` due to the open-world assumption of Linked Data. This assumption means that if a Linked Data model of a Thing Description instance were to omit these vocabulary terms, then the interpreter would not be able to make any assumptions about their actual value.

A TD snippet based on the defined fields is given below:

```
     ...      "forms": [{          "href" :
"http://mytemp.example.com:5683/temp",          "mediaType":
"application/json",          "http:methodName": "POST",          "rel":
"writeProperty",          "security": [{"scheme":"basic",
"in":"header"}]      }]      ...
```

`links`

Each mandatory and optional vocabulary term as defined in the class `Link`, MUST be serialized as a JSON key.

A TD snippet based on the defined fields is given below:

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

`security`

Each mandatory and optional vocabulary term as defined in the class `SecurityScheme`, MUST be serialized as a JSON key.

The following TD snippet shows a simple security configuration specifying basic username/password authentication in the header. The value of `in` given is actually the default value of `header`.

```
    ...    "security": [{        "scheme": "basic",        "in":
"header"    }]    ...
```

Here is a more complex example: a TD snippet showing digest authentication on a proxy combined with bearer token authentication on an endpoint. Here the default value of `in` in the `digest` scheme, `header`, is implied.

```
     ...     "security": [         {            "scheme": "digest",
"proxyUrl": "https://portal.example.com/"          },         {
"scheme": "bearer",              "format": "jwt",            "alg":
"ES256",            "authorizationUrl":
"https://servient.example.com:8443/"         }     ]     ...
```

Security definitions can be given at more than one level. In this case, definitions at the lower levels override (completely replace) the definitions at the higher level.

Security configuration is mandatory. Every form in a Thing MUST have a security configuration either provided in the form itself, at the interaction level directly above it (if security is not configured in the form), or at the Thing level (if security is not configured in either the form or at the interaction level). In the vocabulary defined above, note that `security` is marked as non-mandatory. However, this is only true locally (at a specific level), and only if the security is configured at a higher or lower level. In other words, a security configuration must be provided at *some* level for each form.

Security configuration is considered binding. The security configuration metadata provided in a Thing Description MUST accurately reflect the security requirements of the Thing. Some protocols can ask for authentication dynamically. If a protocol asks for a form of security credentials not declared in the Thing Description then the Thing Description is to be considered invalid.

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

```
 {     "id": "urn:dev:wot:com:example:servient:myThing",     "name":
"MyThing",     "description": "Additional (human) readable information
of the Thing.",     "support": "https://servient.example.com/contact",
"security": [{"scheme": "nosec"}],     "properties": {...},
"actions": {...},     "events": {...},     "links": [...] }
```

To give a more complex example, suppose we have a Thing where all interactions require basic authentication except for one interaction for which no authentication is required. In the following, the `nosec` scheme for the security configuration in the `overheating` event to indicate no authentication is required. For the `status` property and the `toggle` action, however, `basic` authentication is required as defined at the top level of the Thing.

```
 {     ...     "security": [{"scheme": "basic"}],     "properties": {
"status": {             ...             "forms": [{
```

```
"href": "https://mylamp.example.com/status",
"mediaType": "application/json",              }]          }      },
"actions": {          "toggle": {              ...              "forms":
[{              "href": "https://mylamp.example.com/toggle",
"mediaType": "application/json"              }]          }      },
"events": {          "overheating": {              ...
"forms": [{                  "href": "https://mylamp.example.com/oh",
"mediaType": "application/json",                  "security":
[{"scheme": "nosec"}]              }]          }      } }
```

Security definitions can also can be given for different elements at the same level. This may be required for devices that support multiple protocols, for example CoAP and HTTP, with support for different security mechanisms. This is also useful when alternative authentication mechanisms are allowed. Here is a TD snippet demonstrating three possible ways to access a resource: via HTTPS with basic authentication, via HTTPS via digest authentication, or via CoAPS with an API key. In other words, the use of multiple security configurations at the same level provides a way to combine security mechanisms an in "OR" fashion. In contrast, putting multiple security configurations in the same `security` field combines them in an "AND" fashion, since in that case they would all need to be satisfied to allow access to the resource.

```
     ...      "properties": {          "status": {              ...
"forms": [                  {                      "href":
"https://mylamp.example.com/status",                      "mediaType":
"application/json",                      "security": [{"scheme":
"basic"}]                  },                  {
"href": "https://mylamp.example.com/status",
"mediaType": "application/json",                      "security":
[{"scheme": "digest"}]                  },                  {
"href": "coaps://mylamp.example.com:5683/status",
"mediaType": "application/json",                      "security":
[{"scheme": "apikey"}]                  }              ]          }      },
...
```

Media Type

The JSON-based serialization of the TD is identified by the media type `application/td+json`.

CoAP-based WoT implementations can use the experimental Content-Format `65100` until a proper identifier has been registered.

The media type `application/td+json` MUST be also associated with the JSON-LD context `http://www.w3.org/ns/td`. That means that this media type can also be used for contextual identification of the vocabulary within a (simplified) TD instance that may omit the `@context` key term.

Neither the `application/td+json` media type nor a CoAP Content-Format identifier have been registered with IANA yet.

Implementation Notes

ML: Implementation notes are informative. This section contains several normative requirements, which should be moved somewhere else.

JSON Processing

The minimum requirement to read the content of a Thing Description instance is a (simple) JSON parser.

If the key terms `writable` and/or `observable` are not present within a `properties` definition, the default value defined in MUST be assumed.

If the `mediaType` key term is not present within a `forms` definition, the default value as defined in MUST be assumed.

To validate the semantic meaning and follow references to external context vocabulary terms (e.g., iot.schema.org), use of JSON-LD or RDF-based tools and libraries is highly recommended as explained in the next sub-section.

JSON-LD 1.1 Processing

To interpret the semantic meaning of a Thing Description in terms of RDF triples, a Thing Description instance first requires a valid JSON-LD 1.1 representation based on this Thing Description specification. Then this representation can be passed to a JSON-LD 1.1 processor.

The following pre-processing steps of a Thing Description instance must be executed before starting JSON-LD 1.1 processing:

- Before starting JSON-LD 1.1 processing of a Thing Description instance, there MUST be a `@context` key from JSON-LD 1.1 as defined in Section .
- Before starting JSON-LD 1.1 processing of a Thing Description instance, all external vocabulary terms used in the Thing Description MUST provide their context URIs as prefix or within the `@context` field.
- Before starting JSON-LD 1.1 processing of a Thing Description instance, all mandatory vocabulary terms as defined in Section that are missing from the instance MUST be inserted explicitly with their default value.

Section shows an example of how the input and output of such preprocessing would appear.

Example Thing Description Instances

MyLampThing Example

```
 {     "id": "urn:dev:wot:com:example:servient:lamp",     "name":
"MyLampThing",     "description" : "MyLampThing uses JSON-LD 1.1
serialization",     "security": [{"scheme": "psk"}],     "properties":
{         "status": {             "description" : "Shows the current
status of the lamp",             "type": "string",             "forms":
[{                 "href": "coaps://mylamp.example.com/status"
}]         }     },     "actions": {         "toggle": {
"description" : "Turn on or off the lamp",             "forms": [{
"href": "coaps://mylamp.example.com/toggle"             }]         }
},     "events": {         "overheating": {             "description" :
"Lamp reaches a critical temperature (overheating)",
"type": "string",             "forms": [{                 "href":
"coaps://mylamp.example.com/oh"             }]         }     } }
```

```
 {     "@context": ["http://www.w3.org/ns/td",               {"iot":
"http://iotschema.org/"}],     "@type" : "Thing",     "id":
"urn:dev:wot:com:example:servient:lamp",      "name": "MyLampThing",
"description" : "MyLampThing uses JSON-LD 1.1 serialization",
"security": [{"scheme": "psk"}],     "properties": {         "status":
{              "@type" : "iot:SwitchStatus",            "description" :
"Shows the current status of the lamp",            "writable": false,
"observable": false,             "type": "string",            "forms":
[{                   "href": "coaps://mylamp.example.com/status",
"mediaType": "application/json"             }]          }     },
"actions": {          "toggle": {             "@type" :
"iot:SwitchStatus",            "description" : "Turn on or off the
lamp",               "forms": [{                 "href":
"coaps://mylamp.example.com/toggle",                   "mediaType":
"application/json"             }]          }     },     "events": {
"overheating": {             "@type" : "iot:TemperatureAlarm",
"description" : "Lamp reaches a critical temperature (overheating)",
"type": "string",             "forms": [{                "href":
"coaps://mylamp.example.com/oh",                "mediaType":
"application/json"             }]          }     } }
```

Security and Privacy Considerations

Please see the WoT Security and Privacy repository for work in progress regarding threat models, assets, risks, recommended mitigations, and best practices for security and privacy for systems using the Web of Things. Once complete, security and privacy considerations relevant to Thing Descriptions will be summarized in this section.

JSON Schema for TD Instance Validation (normative)

Below is a JSON Schema for syntactically validating Thing Description instances serialized in JSON-LD 1.1.

This schema is known to be too permissive. This issue will be resolved in the next release of this document.

```
 {   "title": "WoT TD Schema for Bundang Plug Fest",   "description":
"JSON Schema representation of the TD serialisation format.",
"$schema ": "http://json-schema.org/draft-06/schema#",   "type":
"object",   "properties": {     "base": {       "$ref":
"#/definitions/url"     },     "@type": {       "$ref":
"#/definitions/type_declaration"     },     "@context": {       "$ref":
"#/definitions/context"     },     "name": {       "type": "string"
},     "id": {       "type": "string"     },     "description": {
"type": "string"     },     "properties": {       "type": "object",
"items": {         "$ref": "#/definitions/properties"       }     },
"actions": {         "type": "object",       "items": {       "$ref":
"#/definitions/actions"       }     },     "events": {       "type":
"object",       "items": {         "$ref": "#/definitions/events"
}     },     "links": {       "type": "array",       "items": {
"$ref": "#/definitions/links"       }     },     "support": {
"type": "string"     },     "security": {       "type": "array",
"items": {         "$ref": "#/definitions/securityScheme"       }     }
```

```
},   "required": [     "name",     "id"   ],   "additionalProperties":
true,   "definitions": {     "context": {       "oneOf": [         {
"type": "array",           "items": {             "anyOf": [
{                 "$ref": "#/definitions/url"               },
{                 "type": "object"               }             ]
},           "contains": {             "type": "string",
"enum": [               "https://w3c.github.io/wot-thing-
description/context/td-context.jsonld",
"http://www.w3.org/ns/td"             ]           }         },
{           "type": "string",           "enum": [
"https://w3c.github.io/wot-thing-description/context/td-
context.jsonld",             "http://www.w3.org/ns/td"           ]
}       ]     },     "type_declaration": {       "oneOf": [         {
"type": "string"         },         {           "type": "array"
}       ]     },     "form_declaration": {       "type": "array",
"items": {         "$ref": "#/definitions/form_element"       }     },
"form_element": {       "type": "object",       "properties": {
"href": {           "$ref": "#/definitions/url"         },
"rel": {           "type": "string"         },         "mediaType": {
"type": "string"         },         "subProtocol": {           "type":
"string"         },         "security": {           "type": "array",
"items": {             "$ref": "#/definitions/securityScheme"
}         },         "scopes": {           "type": "string"         }
},       "required": [         "href"       ],
"additionalProperties": true     },     "properties": {
"additionalProperties": {         "type": "object",         "items": {
"$ref": "#/definitions/property_element"         }       }     },
"actions": {       "additionalProperties": {         "type": "object",
"items": {           "$ref": "#/definitions/action_element"         }
}     },     "events": {       "additionalProperties": {
"type": "object",         "items": {           "$ref":
"#/definitions/event_element"         }       }     },
"property_element": {       "type": "object",       "properties": {
"description": {           "type": "string"         },         "@type":
{           "$ref": "#/definitions/type_declaration"         },
"label": {           "type": "string"         },         "writable": {
"type": "boolean"         },         "observable": {           "type":
"boolean"         },         "forms": {           "$ref":
"#/definitions/form_declaration"         },         "scopes": {
"type": "string"         },         "security": {           "type":
"array",           "items": {             "$ref":
"#/definitions/securityScheme"           }         }       },
"required": [],       "additionalProperties": true     },
"action_element": {       "type": "object",       "properties": {
"description": {           "type": "string"         },         "@type":
{           "$ref": "#/definitions/type_declaration"         },
"label": {           "type": "string"         },         "forms": {
"$ref": "#/definitions/form_declaration"         },         "input": {
"$ref": "#/definitions/data"         },         "output": {
"$ref": "#/definitions/data"         },         "scopes": {
"type": "string"         },         "security": {           "type":
"array",           "items": {             "$ref":
"#/definitions/securityScheme"           }         }       },
"required": [],       "additionalProperties": true     },
"event_element": {       "type": "object",       "properties": {
"description": {           "type": "string"         },         "@type":
{           "$ref": "#/definitions/type_declaration"         },
"label": {           "type": "string"         },         "forms": {
```

```
"$ref": "#/definitions/form_declaration"         },         "scopes": {
"type": "string"         },         "security": {           "type":
"array",           "items": {             "$ref":
"#/definitions/securityScheme"           }         }       },
"required": [],       "additionalProperties": true     },     "links":
{       "type": "object",       "properties": {         "anchor": {
"$ref": "#/definitions/url"         },         "href": {
"$ref": "#/definitions/url"         },         "rel": {
"type": "string"         },         "mediatype": {           "type":
"string"         }       },       "required": [         "href"       ],
"additionalProperties": true     },     "securityScheme": {
"type": "object"     },     "url": {       "type": "string",
"format": "uri",       "pattern":
"(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(([^#]*))?(#(.*))?"     },
"jsonld_url": {       "type": "string",       "format": "uri",
"pattern": "http://[^/?#]*|https://[^/?#]*"     },     "data": {
"title": "Data type",       "anyOf": [         {
"description": "URI of an XSD built-in type",           "type":
"string"         },         {           "description": "URI and media
type for a complex type (XSD, SenML...)",           "type": "object",
"properties": {             "name": {               "type": "string"
},             "href": {               "type": "string"             },
"mediatype": {               "type": "string"             }
},           "required": [             "name",             "href",
"mediatype"           ]         },         {           "$ref":
"http://json-schema.org/draft-06/schema#"         }       ]     },
"dataSchema": {       "type": "object",       "properties": {
"description": {         "type": "string"         },         "enum":
{           "type": "array",           "items": {             "anyOf":
[               {                 "type": "string"               },
{                 "type": "boolean"               },               {
"type": "object"               },               {
"type": "number"               },               {
"type": "array"               }             ]           }         },
"type": {           "type": "string",           "enum": [
"string",             "boolean",             "object",
"number",             "array"           ]         }       }     }   } }
```

Recent Specification Changes

Changes from Second Public Working Draft

- The default serialization format for Thing Description is now plain JSON. Thing Description instances still MAY contain JSON-LD keys such as `@context` and `@type` for facilitating semantic evaluation of TDs. (see Section)
- With regards to JSON-LD, this Thing Description draft uses JSON-LD 1.1 instead of JSON-LD 1.0. With this transition, all vocabulary terms defined in vocabulary definition sections now have a JSON key representation. Previously, there were vocabulary terms represented as key values in TDs.
- In addition to Thing Description core vocabulary, Thing Description now has the data schema vocabulary reflecting a subset of JSON Schema terms as linked data representation (see Section) and the security vocabulary (see Section) for defining security based conditions.
- A security field is available at three levels (Thing, Interaction Patterns and Forms) for providing security requirements for accessing resources. Security configuration is mandatory for each `form`.

- Some vocabulary terms (e.g., `writable`, `observable`, `mediaType`) now have default values that are used by Thing Description processors when they are absent in TD instances (see Section). Note that some security vocabulary terms can also have such values (see Section).
- Things now have a mandatory field `id` for uniquely identifying a Thing, and an optional field `support` to provide information about Thing Description maintainer.
- Interaction patterns have an optional `label` field providing text for such uses as user interfaces.
- The `rel` attribute for `form` now has enumerated allowable values specified.
- JSON-LD @context key value to represent the Thing description context file now is `http://www.w3.org/ns/td`.
- Added a section about media types used for exchanging TDs. (see Section)
- Added a section about implementation notes that provide useful clarifications to implementors. (see Section)
- A JSON Schema is provided in an appendix section to facilitate the validation of Thing Description instances. (see Section)

Changes from First Public Working Draft

Changes from First Public Working Draft are described in the Second Public Working Draft

Acknowledgements

The editors would like to thank Dave Raggett, Matthias Kovatsch, Michael McCool, Michael Koster, Victor Charpenay, Kawaguchi Toru, Michael Lagally, Kazuyuki Ashimura, María Poveda, Daniel Peintner, Ben Francis for their contributions, comments, and guidance.