OpenAPI Thing Descriptions for the Web of Things

Aimilios Tzavaras  Nikolaos Mainas  Fotios Bouraimis  Euripides G.M. Petrakis

School of Electrical and Computer Engineering
Technical University of Crete (TUC)
Chania, Crete, Greece

{atzavaras, fbouraimis}@isc.tuc.gr, {nmainas, petrakis}@intelligence.tuc.gr

Abstract—The Web of Things (WoT) recommendation of W3C suggests a model for integrating Things (e.g. devices) in the Web. In addition to W3C, OpenAPI specification provides a method for documenting RESTful services so that a user or another service can comprehend their purpose and reuse them in applications. This work applies the OpenAPI service description framework to Web objects (i.e. Things). As a result, OpenAPI descriptions of Web Things provide a complete documentation of the services exposed by Things and of their capabilities. The resulting descriptions can be converted to an ontology in order to allow a machine to better understand the inherent meaning of Thing descriptions and interact with them. Then, Thing descriptions exposed in the Web can be easily discovered, queried by Semantic Web query languages (e.g. SPARQL) and checked by reasoners (e.g. Pellet) for consistency or, for inferencing hidden properties. The approach is compared to the Web of Things Description (TD) model of W3C in terms of completeness of the representation.

Keywords—Web of Things, Thing Description, OpenAPI

I. INTRODUCTION

The Web of Things (WoT) initiative [1] aims at unifying the world of interconnected devices (Things) over the Internet. Each device should be published in the Web, be discovered by Web search engines and be reused in applications. The term Thing may refer to any device: a temperature or proximity sensor, a window actuator, a coffee machine, a smart TV, a Wi-Fi connected garage door or, a smart car. The Web of Things (WoT) Architecture recommendation of W3C [2] defines an abstract architecture and sets the requirements for interacting with Things in the Web using the REST architectural style [3]. The WoT Thing Description (TD) of W3C is a JSON template representation of Thing properties (e.g. purpose, data types and operations). TDs are used to expose Thing metadata in the Web so that other Things or clients (i.e. services or users) can interact with them. The effort is supported by a set of developer tools and a list of candidate implementations.

Real-world devices may implement any of a wide range of application specific protocols (e.g. Bluetooth, MQTT, ZigBee etc.). However, the WoT Architecture recommendation of W3C does not bind to any application and, does not depend on the peculiarities of IoT protocols. A workaround to this problem is to deploy a Web proxy on a server (or on a gateway) that keeps a directory with all TDs. In this way, Things become part of the Web and can be accessed via their Web Proxy (i.e. they can be published, aggregated and searched for).

The syntactic binding of WoT TD to JSON constraints the interpretation of Thing information by machines. The WoT Architecture of W3C suggests that TDs should be extended with semantics in order to enhance their information content and make them machine understandable. The resulting representation is JSON-LD [4] or, it can be converted to an ontology [5]. The motivation for using ontologies is that they are closer to the way machines analyse and comprehend the content of TDs. It is therefore easier for a machine to discover similarities in TDs and search the Web for Things with the desirable properties (e.g. using ontology query languages such as SPARQL [6]) or, apply semantic inference in order to detect services with inconsistencies (e.g. using ontology reasoners such as Pellet [7]). Enabling automatic synthesis of Things in applications would be possible as well. This is a more ambitious goal and is outside the scope of this work.

OpenAPI Specification2 suggests a description format for REST APIs. It is a widely adopted industry standard endorsed by Linux Foundation and supported by large software vendors like Google, Microsoft, IBM, Oracle and many others. OpenAPI format is based on JSON (or YAML) and comprises a large set of properties for composing service descriptions. OpenAPI 3.0 is the first major update of the specification released in 2017. Version 3.1 (as of February 2021) provides full JSON Schema support (i.e. all keywords of JSON Schema vocabulary can be used in OpenAPI 3.1) while being fully compatible with version 3.0. OpenAPI can be enriched with text descriptions so that users can easily discover and understand the service and interact with it.

In a previous work [8] we analysed the reasons that cause ambiguities in OpenAPI descriptions (e.g. similar to TD, the same property may appear with different names within the same OpenAPI document or, its meaning may not be defined at all) and showed that, in order to eliminate ambiguities, each ambiguous property must be semantically annotated and mapped to a semantic model (e.g. a vocabulary or ontology). Alongside, it is plausible to convert an OpenAPI description to an ontology [9]. OpenAPI ontology incorporates features of Hydra [10] for modeling service operations and, SHACL [11] for validating Schema descriptions against the ontology. The

1https://www.w3.org/WoT/developers/
2https://www.openapis.org
algorithm for mapping service descriptions to the OpenAPI ontology is available as a Web Application\(^1\) for testing. The idea of using ontologies is not new. Existing ontologies fit well the needs or remote procedure call technologies such as SOAP \(^{12}\). However, the emergence of REST, generated new difficulties in the representation of hypermedia-driven APIs (such as REST) that call for the dynamic discovery of resources at run-time (referred to as HATEOAS\(^3\)). This feature, is neither supported by known service ontologies (such as OWL-S \(^{13}\) for SOAP services) nor by the TD model of the WoT Architecture of W3C.

The proposed Things description format suggests a lightweight version of the general purpose OpenAPI description for REST services specialized for Web Things. It is an alternative to the TD of the W3C Web Architecture and offers a more informative and elaborate mechanism for the description of Things exposing their functionality in the Web as RESTful services. However, both representations share common features and serve the same purpose (i.e. discovering Things in the Web of Things) which are reviewed in this work.

Compared to the TD of W3C Architecture, OpenAPI is a mature framework providing both, human and machine readable descriptions of Web services: OpenAPI describes the paths, operations, responses, parameters and schemas used by a service while, TD is merely a short and abstract definition of the Things interactions with a client. Given an OpenAPI service description, a client can easily understand and discover the functionality of a Thing and how to interact with it with minimum implementation logic. It provides the needed information about service endpoints, service operations, the exchanged message formats and the conditions which need to be fulfilled before invoking the service. Finally, OpenAPI is supported by a complete tools pallet\(^5\) (e.g. editors, description validators and client SDK generators).

This work introduces a mechanism for generating OpenAPI Thing Descriptions from user input. A template description for Things is proposed based on the idea of mapping Thing properties to OpenAPI properties. The template is extensible (i.e. more properties can be added). If the meaning of Thing properties is ambiguous, they are mapped to equivalent properties in semantic models (i.e. vocabularies or ontologies) using OpenAPI external properties (i.e. x-properties) \(^8\). To show proof of concept, the OpenAPI thing approach is applied on a realistic smart door device with a lock-unlock actuator and a DHT22 sensor\(^6\).

The TD and OpenAPI approaches for Thing description are discussed in Sec. II and Sec. III respectively; they are compared in Sec. V. The OpenAPI template for Thing description is presented in Sec. IV. The process for generating OpenAPI descriptions for Things is described in Sec. VI followed by conclusions and directions for research in Sec. VII.

---

\(^1\)http://www.intelligence.tuc.gr/semantic-open-api/
\(^2\)https://restfulapi.net/hateoas/
\(^3\)https://openapi.tools
\(^4\)https://www.adafruit.com/product/385
\(^5\)https://github.com/eclipse/thingweb.node-wot

---

\(^6\)https://github.com/eclipse/thingweb.node-wot

---

II. WoT THING DESCRIPTION (TD)

Thing Description (TD) is a central building block of the WoT Architecture of W3C. It provides the entry point for discovering services and resources related with a Thing. The WoT Architecture suggests that TDs are hosted in a directory service (on a gateway or the cloud) providing a Web interface for registering and searching for Things. Thingweb node-wot\(^7\) is an implementation of TD along with an implementation of Thing operations using a JavaScript API similar to the Web browser APIs. It provides an API Interface that allows scripts to interact with Things using Web protocols such as HTTP, HTTPS, CoAP, MQTT and Websockets.

Fig. 1 illustrates the structure of a TD. Compared to OpenAPI, TD is a more abstract description of a Thing. In fact, the description is a much shorter document (which is desirable if TDs are hosted onboard, especially in cases of resource constraint devices) that describes the Interactions, Data Schemas, Security Configuration and Protocol Binding of a Web Thing. It includes the Thing’s name, its unique identifier, its security requirements, a title, an optional human readable description, and all the possible interactions supported by the Thing. Similar to OpenAPI, TD adopts a JSON serialization format which can be enhanced with a context field for converting the JSON format to JSON-LD (OpenAPI resorts to JSON or YAML and does not support JSON-LD) or, it can be converted to an ontology \(^5\).

![Fig. 1. Thing Description (TD) document structure.](image)

Listing 1 is an example TD for a smart door that contains (a) a context attribute which extends the definition with additional vocabulary terms, (b) the identifier of the device, (c) an indicative title, (d) the security configuration of the service (Basic Authentication in this example), (e) interactions supported by the smart door; the state property, the lock and unlock actions, the door open event (i.e. the state property of the door turning to open) and, (g) the forms field that describes how each interaction can be performed; it specifies the protocol that should be used (i.e. HTTPS) and the operation endpoint. The endpoint to retrieve the last state value of the smart door is specified in the Properties object (i.e. in the forms array). The protocols and the endpoints used to execute the lock and the unlock actions are specified by an Actions object; the protocol, the endpoint and the sub-protocol (e.g. the exact mechanism used for asynchronous notifications) for
III. OPENAPI

Fig. 2 illustrates the structure of an OpenAPI service description. It comprises many parts (objects). Each object specifies a list of properties which can be objects as well. Objects and properties defined under the Components unit of an OpenAPI document can be reused by other objects or they can be linked to each other (e.g. using keyword $ref). However, these links are not always explicitly expressed (e.g. there can be properties with the same name with no reference to one another or to an external model). The Info object provides non-functional information such as the name of the service, service provider, license information and terms of the service. The Servers object provides information about where the API servers are located. Servers can be defined for different operations (locally declared servers override global servers). The service description contains an Info object with some non-functional information for the service, an External Documentation object and all possible Tag objects: a Web Thing tag, a Properties tag, an Actions tag and a Subscriptions tag are used to group properties by type or resource.

The description includes a Paths object that holds all the available service paths (i.e. endpoints) and their operations, which may also specify Parameter objects. The Paths object provides information about expressing HTTP requests to the service and about the responses of the service. It describes the supported HTTP methods (e.g. get, put, post etc.) and defines the relative paths of the service endpoints (which is appended to a server URL in order to construct the full URL of an operation). The Responses object describes the responses of an operation, its message content and the HTTP headers that a response may contain. The Parameters object describes parameters that operations use (i.e. path, query, header and cookie parameters).

The Components object lists re-usable objects. That includes (among others) definitions of schemas, responses, headers, parameters and security schemes. The Security object lists the security schemes of the service. The specification supports HTTP authentication, API keys, OAuth2 common flows or grants (i.e. ways of retrieving an access token) and OpenID Connect.

The Schemas object describes the request and response messages based on JSON Schema. A Schema object can be a primitive (string, integer), an array or a model or an XML data type and may also have properties of its own accord (i.e. externalDocs). New data types can be defined as a composition or a specialization of existing ones using properties allOf, oneOf, anyOf and not. Schema properties do not have a semantic meaning and, (as a result) their meaning can be vague. In addition, there can be Schema properties with different names that share the same meaning. A human might easily resolve ambiguities either by the element names or by the description that may be provided but, a machine cannot. The problem is solved by associating each Schema object with a semantic model [8]. OpenAPI properties are semantically annotated and associated to entities of a semantic model using the x-refersTo extension property. The x-kindOf extension property defines a specialization between an OpenAPI property and a semantic model (e.g. a class). The x-mapsTo extension property denotes that a Schema property is semantically equivalent with another property in the same document. Additional extension properties are defined to clarify the meaning of the members in a collection of objects (x-collectionOn), for grouping Schema objects by type (x-

---

3

onResource) and for clarifying the meaning of operations (x-operationType). Detailed vocabularies (e.g. www.schema.org) can be referred to using x-properties and define Things for an application domain. The semantic meaning of the service is captured by the OpenAPI ontology [8].

Apart from properties and actions, a Thing may also support subscriptions. A subscription is the result of subscribing to a specific resource of a Thing (e.g. a particular property or action) in order to get notified on changes of the Things state information (e.g. new temperature value). The subscriptions are stored in a storage structure so that they can be retrieved by a subscription identifier (TD supports subscriptions to events which are not stored).

Listing 2 shows how x-refersTo is used to semantically associate the Actuator type of the smart door to the Sosa ontology [14]. The x-kindOf extension property is used to semantically annotate the Thing properties (i.e. id, name) with concepts in www.schema.org vocabulary.

Listing 2. OpenAPI Thing Description for the smart door.

```json
schemata:
  Webthing:
    - id
    - name
    type: object
  x-refersTo: 'http://www.w3.org/ns/sosa/Actuator
properties:
  id:
    type: string
    default: SmartDoor
  x-kindOf: 'http://schema.org/identifier'
  name:
    type: string
    example: IoTSmartDoor
  x-kindOf: 'http://schema.org/name'
```

Listing 3 shows a delete operation on a subscription using its subscription identifier. A human may refer to the description of the operation in order to understand its intended purpose, but a machine needs additional information which is provided by the x-operationType extension property. The value of the property is a URL pointing to the concept that semantically describes the operation type. The Action type of the www.schema.org vocabulary provides a detailed hierarchy of Action sub-types that can be used by the property.


```json
paths:
  /subscriptions/{subscriptionID}:
    delete:
      tags:
        - Subscriptions
      summary: Delete a subscription
      description: reject the request with an appropriate status code or remove (unsubscribe) the subscription and return a 200 OK status code.
      operationId: deleteSubscription
      x-operationType: 'https://schema.org/DeleteAction'
      parameters
        - name: subscriptionID
      in: path
      description: The id of the specific subscription
      required: true
      style: simple
      explode: true
      schema:
        type: string
        example: 5fd23faccde6be05da68bcf6
      responses:
        '200':
          description: OK
        '404':
          description: Not found
```

IV. OPENAPI WEB THING TEMPLATE

The OpenAPI Web Thing template employs a JSON (or YAML) description format which is common to all Things. It is a valid OpenAPI document which can be handled by all known OpenAPI tools (e.g. Swagger editor, code generator etc.). All Things expose properties and, depending on their type, they may also support actions. The building blocks of OpenAPI Thing template are its resources. Each resource of a Thing supports operations. For example, the Properties resource supports operations for each property and, the Actions resource supports operations for each action that a Thing can perform. Thing resources and their supported operations are described below.

a) Thing resource: It is an abstract description of a Web Thing. The operation retrieves the description of the Thing by issuing an HTTP GET request to the root URL of the Thing that is, its IP address and the default port that follows the IP (e.g. http://34.122.93.207:5001/MySmartDoor). The operation (i.e. get, put, post etc.) and its corresponding path are specified in the Paths object of the OpenAPI description. The Schema object used to describe a Thing payload (returned in the response body) is included in the Components object. The Things identifier is an indicative name (string), a description of the device (string), the date it was created or updated or registered to an application (date-time string) and, tags for all devices.

b) Properties resource: It defines the properties of a Thing (e.g. temperature, humidity) and describes measurements of properties (e.g. temperature and humidity values for DHT22 sensor) or, the internal state of a Thing (e.g. the state of the smart door). This resource applies to all devices.

The first operation retrieves a list of properties. It is realized by issuing an HTTP GET request to the /properties endpoint of the root URL of the device. The endpoint (i.e. /properties) and operation (i.e. get, put, post etc.) are specified in the Paths object. A JSON array describing the Thing properties is returned in response to this operation. The array contains a JSON object for each property of the Thing. This object includes a short description of the property (e.g. identifier, name and any other attribute set by the Thing owner) and the current (i.e. last) measurement value of the Thing (e.g. last humidity value of DHT22) or, the internal state of the Thing (e.g. the state of a smart door). This operation returns also the description of each property. In the case of a sensor that observes more than one property (e.g. pressure, temperature),...
information about properties is included in the response array. The Schema object that describes a property of the device is defined in the Components object. If the Thing accepts one or more from a list of alternative properties (like the DHT22 sensor), the response JSON array is implemented using polymorphism (i.e. using property anyOf). OpenAPI supports combination of model definitions using the properties allOf, oneOf and anyOf of JSON Schema.

The second operation on Properties Resource retrieves the current value (or the most recent values) of a property. It requires sending an HTTP GET request to the /properties endpoint of the root URL of the Thing followed by the specific Thing property name as a path parameter (e.g. rootURL/properties/state for a smart door). In case there is more than one state value, an array of JSON objects is returned in the response body. The particular endpoint (i.e. /properties/state) and operation are also specified in the Paths object of the service description. The Schema object that describes the payload (i.e. temperature of humidity) is defined in the Components object.

c) Actions resource: It defines the allowed actions on a Thing, such as execution commands (e.g. a command sent by a client to a window actuator to open). In all cases below, the Schema object that describes the execution information of an action is returned in the response body and it is defined in the Components object.

The first operation retrieves a list of actions by issuing an HTTP GET request to the /actions endpoint of the root URL of the Thing. This operation returns an array of descriptions for the actions that the Thing may perform (e.g. locking and unlocking for a smart door). The response is in the form of a JSON array which contains an identifier and a name for each possible action. The particular endpoint (i.e. /actions) and operation are specified in the Paths object. An action identifier and an action name are properties of the corresponding Schema object.

The second operation retrieves all recent executions of a specific action and issues an HTTP GET request to the /actions endpoint of the root URL of the Thing followed by the specific action name as a path parameter. For example, a GET rootURL/actions/{actionName} (e.g. GET rootURL/actions/lock) returns a JSON array of the recent executions of an action including information about the status of the action execution and a timestamp. A time delay value might have been sent in the request body in order to schedule the execution of the action at a later time.

The next operation executes an action. An HTTP POST request is sent to the /actions endpoint of the root URL of the Thing followed by an action name as a path parameter (e.g. POST rootURL/actions/lock in order to lock a smart door). The service generates a unique identifier for each execution of an action which is stored in a database and can be used to retrieve the action. In fact, a unique integer number is generated and its value is used to identify the action. The operation is specified in the /actions/{actionName} path (e.g. /actions/lock) in the Paths object of the OpenAPI description.

The last operation retrieves the status of an action using its execution identifier (i.e. executionId): an HTTP GET request is sent to the /actions endpoint of the root URL of a Thing followed by the name of the action and the execution identifier as a path parameter (e.g. GET rootURL/actions/lock/156). The endpoint for the particular action (i.e. /actions/lock/{executionId}) and the respective operations are specified in the Paths object of the service description. The execution information are the same as those used in the second operation.

d) Subscriptions resource: It describes subscriptions to Web Things (e.g. to their actions and properties). For example, users and services can subscribe to the humidity property of a specific sensor in order to get notified on changes of humidity.

The first operation creates a new subscription (i.e. a user or service may subscribe to a specific Web Thing resource). Subscriptions are ideally supported using custom callbacks (i.e. Webhooks) which are naturally supported by Websocket protocol. An HTTP POST request is required in order to create and store a new subscription. The particular endpoint (i.e. /subscriptions) and operation are specified in the Paths object. Although the subscription information is defined in the request body by the subscriber (i.e. client or service), the Schema object that describes the payload can be a reusable objects which is defined in the Components object. The object contains an indicative name, a description, the subscription type (e.g. webhook), the callback URL, an object containing (i.e. as object properties) the type and the name of the resource to which the subscription is made, the expiration date of the subscription and, a throttling parameter which is used to specify a minimum inter-notification arrival time for the subscription. A response header containing the subscription identifier is returned as long as the operation is successful (i.e. a 200 OK response is returned).

The second operation retrieves a list of subscriptions made to a specific Thing or Web Thing resource. The operation issues an HTTP GET request to the /subscriptions endpoint of the root URL of the Thing. A JSON array containing the subscriptions made to a specific Web Thing resource is returned. For example, in order to retrieve the subscriptions made to the smart door, an HTTP GET request is issued (i.e. GET http://34.122.93.207:5001/subscriptions). The operation is specified in the /subscriptions path in the Paths object. The Schema object describing the payload for each subscription included in the JSON array is defined in the Components object.

The next operation retrieves a subscription by its subscription identifier as a path parameter. The operation requires sending an HTTP GET request on the /subscriptions endpoint of the root URL of the Thing followed by subscription identifier as a path parameter (e.g. GET rootURL/subscriptions/5a82be4d093a1f1b95ac0f730). A JSON representation of the subscription is returned in response. The particular endpoint (i.e. /subscriptions/{subscriptionId}) and operation are
included in the Paths object of the smart door. The subscription identifier (string) is defined as a Path parameter under Paths object. The Schema object that describes the subscription (i.e. in the response body) is also a reusable object which is defined in the Components object.

The last operation deletes a subscription (using its subscription identifier) by issuing an HTTP DELETE request to the /subscriptions endpoint of the root URL of the Thing followed by the subscription identifier as a path parameter. If successful, the subscription is removed and a 200 OK response header is returned. The operation is included in the relative path (i.e. /subscriptions/{subscriptionID}) in the Paths object of the service description. The subscription identifier (string) is defined as a Path parameter in the Paths object.

The template is common to all devices. For example, the Web Thing resource, the Web Thing Tag and the operation for retrieving the Web Thing description apply to all Things and cannot be changed. The same applies to the Properties resource and to the Properties Tag. All request and response bodies must be defined. The payload attributes of devices, properties and actions are set by the device owners and are defined in the Components object. The Parameter object specifies all operation parameters.

A. OpenAPI Web Thing Ontology

In order to facilitate the search for Things in the Web, OpenAPI Thing Descriptions are translated to an ontology [8], [9]. The ontology translation process has been incorporated into an application in the Web9. The main functionality of the application supports uploading OpenAPI descriptions of Things (in YAML or JSON) and their instantiation to the ontology. The ontology can be downloaded in TTL (turtle) format, can be searched using SPARQL or checked using Pellet. Listing 4 is the OpenAPI description of an action that creates a subscription to a DHT22 or a smart door resource. The x-operationType property states that the type of the post operation is clarified by https://schema.org/CreateAction.

Listing 4. OpenAPI description of a subscription to a resource.

```json
"post": {  
"tags": [  
"Subscriptions"  
],  
"summary": "Create a subscription",  
"description": "Subscription to a resource (DHT22 or smart door).",  
"operationId": "createSubscription",  
"x-operationType": "https://schema.org/CreateAction",  
"requestBody": {  
"description": "Create new subscription",  
"content": {  
"application/json": {  
"schema": {  
"$ref": "#/components/schemas/SubscriptionRequestBody"  
}  
}  
},  
"required": true
}
```

The SPARQL query of Listing 5 on the ontology would respond with the names of all operations (on any Thing) with type defined by https://schema.org/CreateAction. The response would include subscriptions (and other actions) to the smart door or DHT22. In this example, the query responds with the name of the operation (i.e. createSubscription) and its text description.

Listing 5. Retrieve Things that support Actions.

```sparql
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api#>
SELECT ?graph ?operName ?operDescription WHERE {  
GRAPH ?graph {  
?operation openapi:name ?operName .  
?operation openapi:description ?operDescription .}
}
```

V. Comparison

In the following, the TD of the WoT Architecture of W3C and the OpenAPI approach are compared based on their capacity to describe Web Things and their functionality (i.e. services exposed by Things). TD and OpenAPI present remarkable similarities. The particular properties and actions supported by Things are actually defined in both approaches, while the operations, specific endpoints and protocols used to interact with Things are provided by both, TD and OpenAPI descriptions. Both, TD and OpenAPI make services offered by Things machine-understandable and discoverable by Semantic Web tools (by translating the respective descriptions to ontologies). TD adopts JSON-LD format which is more powerful than JSON or YAML. A TD may also refer to extra IoT protocols (e.g. CoAP, MQTT), while OpenAPI only supports HTTP(S) and Webhooks. For example, the TD of a sensor may specify an MQTT Protocol Binding. A TD is a much shorter description than an OpenAPI document. This is helpful in case of a resource-constrained devices (e.g. with limited memory).

TDs can describe events, while OpenAPI documents can describe subscription operations using Callbacks or Webhooks properties added to a Path object (at the same level as parameters, responses, etc.). Compared to subscriptions, events is a narrower concept. Events actually represent Thing state transitions. A subscription is the result of subscribing to a specific resource of a Thing (e.g. a particular property or action and not merely an event). As a result, subscriptions can describe any number of HTTP requests that may arrive in response to an earlier HTTP request. For example, clients (i.e. users or services) can subscribe to Thing resources using
a publish/subscribe pattern implemented using the WebSocket protocol, in order to get notified on changes of Things state information (e.g., new temperature value). An event is defined as an interaction affordance that describes an event source, which asynchronously pushes event data from a Thing to the subscribed client (e.g., overheating alerts). Clients can only subscribe or unsubscribe to an event and receive asynchronous notifications (alerts) when the event occurs. The WoT Architecture of W3C defines operations for subscribing and unsubscribing to events (e.g., overheating of a device).

Both TDs and OpenAPI describe the security requirements of services exposed by Things. The HTTP security schemes, vocabulary, and syntax in the WoT Architecture of W3C share many similarities with OpenAPI v3.0.1: TD supports security configuration for Basic Authentication, Digest Access Authentication, API key authentication, Bearer Token, Pre-shared key authentication (i.e., using pre-shared keys such as TLS-PSK) and OAuth 2.0 authentication. A combination of security schemes can also be defined in the TD security definition (CommoSecurityScheme), as long as the security schemes can be combined. Apparently, when no authentication or other mechanism is required to access the resources of a Thing, this is declared in the TD security definition (NoSecurityScheme). OpenAPI describes HTTP authentication schemes (i.e., Basic Authentication, Bearer Token and HTTP schemes defined by RFC 7235 and HTTP Authentication Scheme Registry) that use the Authorization header, API key authentication (i.e., in headers, query strings or cookies), OAuth2.0 common flows and OpenID Connect.

An OpenAPI service document defines all service request and response bodies (schemas), operations, HTTP status codes, headers, parameters (e.g., path parameters), security requirements, non-functional information, etc. OpenAPI also defines services in a way that eliminates ambiguities and provides Web Thing service descriptions which are uniquely defined and discoverable, (i.e., using semantic annotations). Documentation generation tools can use an OpenAPI description to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, etc.

Both approaches include the names of the Thing interactions (i.e., properties, actions) and describe the functionality (service) of the Thing. They do not describe the interactions in detail. A user can only find out which properties and actions the Thing actually supports, since they are indirectly indicated in the descriptions (in the form of paths, schemas, operations etc). The values of these properties and actions must be provided by the Thing owner (i.e., a user).

OpenAPI is a promising technology towards understanding and constructing Web services that meet the HATEOAS requirement of REST architectural style. Two new features are introduced in the latest OpenAPI v3.0 referred to as Links and Callbacks. Links are defined in the service response section to allow values returned by a service call to be used as input for a next call. This is an attempt to incorporate HATEOAS functionality in the specification. Callbacks is a feature for defining asynchronous APIs or Webhooks. Callbacks define the requests that the described service will send to another service in response to certain events. TD, on the other hand, does not support HATEOAS.

Overall, OpenAPI outweighs the TD of the W3C Architecture. TD is an abstract description of the device and of its functionality; OpenAPI is detailed and complete: It fully describes the functionality of a device and provides all the information a client needs to use the services it provides and not just interact with the device, as TD does.

VI. Generating OpenAPI Web Thing Descriptions

The flow-chart of Fig. 3 summarizes the mechanism that generates the OpenAPI description of a Thing from user input. The input comprises: a) the standard OpenAPI Thing Description template of Sec. IV that applies to all Things and, b) a payload in JSON with the user settings (e.g. security settings) and the Thing characteristics that will be instantiated to the template. The user specifies also the necessary information that characterizes the device and the functionality it supports (e.g. the properties it provides, the actions it performs, etc.). The output of this mechanism is the OpenAPI description (in YAML or JSON) of the Thing. The mechanism is a RESTful service which is implemented in Python Flask and is available on Github\(^\text{10}\) for download and testing. It applies to any device as long as its functionality can be exposed using REST. As a use case, the complete OpenAPI Thing descriptions for a smart door and for a DHT22 sensor device (along with their corresponding JSON files given as input to the mechanism), can be found in the same Github address.

![](https://github.com/Emiltzav/wot)

Fig. 3. Generating an OpenAPI Thing Description.

Initially, the process creates the OpenAPI objects for the Web Thing description: Info, Security, Servers, Schema and (optionally) External document objects are created and appended in the OpenAPI Thing template. As long as the user has set external documentation information, the process creates an External Documentation object. Next, the process appends the Thing’s description payload (as a Schema object) under `Webthing model object` (as in Listing 2). This payload

\(^\text{10}\)https://github.com/Emiltzav/wot_openapi_generator
describes the device and its features. Basic payload attributes (e.g. identifier, name, description, etc.) are mandatory. Available, Security Requirement objects are set next (e.g. HTTP Authentication, OAuth2.0, OpenID Connect). The process reads a list of available servers as an array of Server objects. The values of all OpenAPI objects are defined and instantiated to the respective objects in the next stages.

Schemas, parameters, paths (i.e. endpoints), operations and security information are defined. For example, apart from the /properties path which is standard for all Things, a new path is appended to the service description for each particular property of the device. If the device supports actions, the mechanism appends a standard Action Tag. Input regarding the Actions resource and their security settings is provided. For example, the /actions path (standard for all Things that perform actions) is appended to the Paths object. All relative action execution operations and their response payload models (Schemas) are also defined in the input. If the user wishes to set a request body for the action execution operations (i.e. the commands to lock or unlock the smart door), this can be specified in the input as well. If the device supports subscriptions, a Subscriptions Tag is added to subscription objects (i.e. paths, operations, schemas, etc.) along with the security settings related to the Subscription resource. In fact, subscription paths, operations and Schemas are predefined in the OpenAPI Thing Description template (i.e. they are the same for all Things).

For the smart door in particular, the template contains an Info object (i.e. non-functional information), a Servers object (i.e. where the API servers are located) and an External Documentation object. The device exposes information about its current state (i.e. open, closed, locked), includes a lock-unlock actuator, and provides the possible resources for this (i.e. Thing, properties, Actions and Subscriptions resources). The document describes all subscription operations: the operation for subscribing to a smart door resource, the subscription retrieval operations and the subscription delete operation. The Paths object describes the relative paths for the service endpoints exposed for the smart door. The Components object specifies the schemas that describe request bodies, response bodies, etc. The description contains schemas describing the response payloads returned by the operations on the Properties and the Actions resources, and all the payloads of the Subscription operations. The schemas for the subscription operations are predefined in the template (additional schemas are defined by the user). The template is also enriched with semantic annotations using x-properties.

VII. CONCLUSIONS

Web Things can be fully described using OpenAPI similar to the way RESTful services are described. An OpenAPI template (i.e. a specialization of OpenAPI) and a mechanism for generating descriptions Thing descriptions are proposed and are available on Github for testing. Compared to Thing Description (TD) of W3C Architecture, OpenAPI is more detailed and complete. It provides all the information that a client (i.e. user or service) needs in order to use the service and not just interact with it (as TD does). Future work, will focus on Things discovery and composition. The work will resort to semantic descriptions of Things (i.e. ontologies) which can be derived from OpenAPI. A query language in the spirit of SOWL-QL [15] will be designed so that the user need not be familiar with the specifics of the Things representation.

REFERENCES